

source  
ing the

(1.1)

ram and  
logically  
, while,  
:=. The

ple, when  
generates a  
de, if it is  
ence of **id**

tial, and  
an identifi-  
fier. The  
y:

(1.2)

:= and the  
er that until

ve also been  
structure on  
§. 1.11(a). A  
ch an interior  
ds containing  
a record with  
the others to  
bout language  
or nodes. We  
tively.

n explicit inter-  
k of this inter-  
This intermedi-  
ould be easy to

s. In Chapter 8,

SYMBOL TABLE	
1	position . . .
2	initial . . .
3	rate . . .
4	

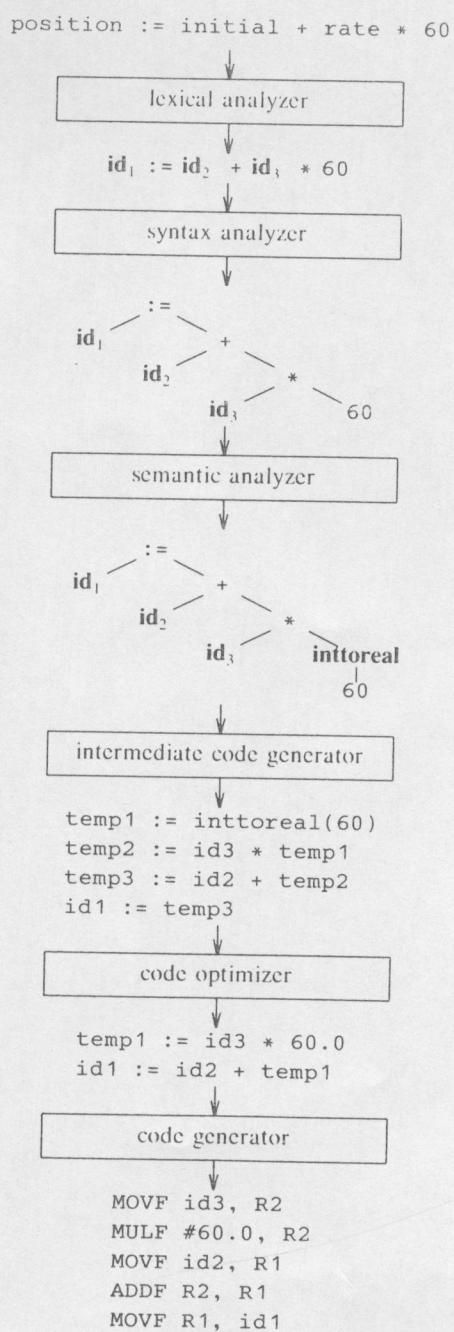


Fig. 1.10. Translation of a statement.

## ניתוח לקסיקלי

### תפקידו המנתח הלקסיקלי

זהוי אסימונים (tokens) בקלט והעברתם למנתח התחבירי העברת תכוונות של אסימונים לשלבים הבאים בקומפיאר טיפול בשגיאות לקסיקליות ועוד (למשל מעקב אחר המיקום בקלט).

### מושגים בסיסיים

עבור כל אסימון יש קבוצה של מחרוזות שכאשר הן מופיעות בקלט, האסימון הנთון מיוצר על ידי המנתח הלקסיקלי כדי ליצגן. קבוצה זו מותוארת על ידי תבנית (pattern). אומרים שהתבנית מתאימה לכל מחרוזת בקבוצה.

לקסמה (lexeme) היא סידרה של תווים בקלט שמתאימה לתבנית עבור אסימון.

תאור התבנית	דוגמאות לקסמות	אסימון
while	while	while
if	if	if
relop	<, >, >=, !=	< או > או >= או != או ==
id	foo, bar, count	אות וلاحכלה אותיות או ספרות
string	“hello world”	סידרת סימנים (שאינם גרשאים) המוקפת בגרשיים.

את התבניות נוח לתאר בעזרת ביטויים רגולריים. למשל התבנית עבור **id** עשויה להיות  $*[a-zA-Z][a-zA-Z0-9]*$  והtbנית עבור **relop** עשויה להיות  $< | > | <= | >= | == | !=$

דוגמה  
נתבונן בקלט הבא:

while (i < 10) /\* this is a comment \*/ j = i;

עבור קלט זה ייצור המנתח הלקסיקלי את האסימונים הבאים:  
**while leftpar id relop num rightpar id assignop id mulop id semicolon**

בנייה אוטומטית של מנתח לקסיקלי (דומה ל-lex)  
אנו מעוניינים לבנות מנתח לקסיקלי שידע זיהות התבניות לקבוצה נתונה של ביטויים רגולריים (תבניות – patterns) pn ... p2 , p1 . לכל אחת מהtbניות משוויכת פעולה (action). בכל פעם שהמנתח נקרא, עליו למצוא את הרישא הארוכה ביותר של (שארית) הקלט אשר מתאימה לאחת מהtbניות האלו ולבצע את הפעולה

המשויכת לאוֹתָה הַתְּבִנִית. אִם הַרְשָׁא הַאֲרוֹכָה בַּיּוֹתֶר מִתְאִימָה לִיוֹתֶר מִתְבִּנִית  
אַחַת, נִתְנַת עֲדִיףָת לְתַבִּנִית המופיעת קָודֵם בְּרִשְׁמַת הַתְּבִנִיות.

1. בּוֹנִים אֹטוֹמַט לֹא דְּטְרָמִינִיסְטִי עַבְורָ כָּל אַחַת מִתְבִּנִיות הַנְּטוּנוֹת.

2. מְשֻׂלְבִים אֶת הָאֹטוֹמַטִים הַנִּיל לֹאֹטוֹמַט אַחַד לֹא דְּטְרָמִינִיסְטִי (שָׁנַׁקְרָא לֹ  
N) בָּאוּפָן הַבָּא :

מוֹסִיפִים מִצְבַּח הַתְּחִלָּתִי חֲדַש וּמְחַבְּרִים אָתוֹ בְּקַשְׁתּוֹת הַמְסֻׂמְנוֹת  
בְּאָפְסִילּוֹן כָּל אַחַד מִהְמַצְבִּים הַתְּחִלָּתִים שֶׁל הָאֹטוֹמַטִים שֶׁבְנֵינוּ בְּסֻעִיף  
א'

בְּאֹטוֹמַט הַמְשׁוֹלֵב יִהְיֶה מִצְבַּח (או מִצְבִּים) מַקְבֵּל עַבְורָ כָּל אַחַת מִתְבִּנִיות.

3. בּוֹנִים אֹטוֹמַט דְּטְרָמִינִיסְטִי D הַשְׁקוֹל לְ— N בְּעֵזֶרֶת הַאלְגּוֹרִיתְם שֶׁרָאִינוּ (subset  
construction). כָּרְגִּיל הַמִּצְבִּים הַמְקַבְּלִים של D הַם אַלְוּ המִתְאִימִים לְתַת קְבוֹצָה  
שֶׁל מִצְבַּי N הַכּוֹלֶת לִפְחוֹת מִצְבַּח מַקְבֵּל אַחַד. כָּל מִצְבַּח מַקְבֵּל של D מְשֻׂשׁ לִזְיהָוִי  
שֶׁל תַּבִּנִית מְסֻׂוִימָת. אִם בְּתַת קְבוֹצָה שֶׁל מִצְבַּי N המִתְאִימָה לִמִּצְבַּח הַמַּקְבֵּל של D  
יֵשׁ יוֹתֶר מִצְבַּח מַקְבֵּל אַחַד אֲזַנְתָּה הַעֲדֵף לִמִּצְבַּח הַמַּקְבֵּל המִתְאִים לְתַבִּנִית  
שֶׁמְפַעַת קָודֵם בְּרִשְׁמַת הַתְּבִנִיות.

4. המנתח הַלְּקָסִיקְלִי יִפְעַל עַל יְדֵי כִּי שִׁיעָשָׂה סִימּוֹלָצִיהַ של אֹטוֹמַט D. כִּי  
לְמַצּוֹא אֶת הַהְתָּאִמָה הַאֲרוֹכָה בַּיּוֹתֶר, המנתח יִמְשִׁיךְ לְבָצָע מַעֲבָרִים (גַּם כִּי  
הָוָא יָכַנס לִמִּצְבַּח מַקְבֵּל) עַד אֲשֶׁר הָוָא "יִתְקַע" (יִגְעַל לְסוֹף הַקְּלָט אוֹ יִמְצָא בְּמִצְבַּח  
שֶׁאֵין מִמְנוּ מַעֲבָר עַל סִימּוֹן הַקְּלָט הַנוֹּכָחִי). אֶת הַלְּקָסִיקָה שַׁהְתָּאִמָה מַוְצָּאים עַל יְדֵי  
חַזְרָה אַחֲרָונית אֶל הַמָּקוֹם בְּקָלְטָה בְּוֹ הַיְיָנוּ בְּפָעַם הַאַחֲרָונה שַׁהְמַנְתָּחָה נִכְנַס  
לִמִּצְבַּח מַקְבֵּל. אֲזַנְתָּה גַּם הַפְּעָולָה (action) המִתְאִימָה.

דוגמא נְרָאָה כִּי־צִדְבָּנִים מַנְתָּחָה המזוהה אֶת שְׁלֹשָׁת הַתְּבִנִיות הַבָּאות :

$$\begin{array}{c} a+ \\ abb \\ a+b^* \end{array}$$

רָאשֵׁית נְבָנָה אֹטוֹמַטִים עַבְורָ כָּל אַחַת מִתְבִּנִיות בְּנִירְפֵּד (הַתְּבִנִיות כָּל־  
פְּשׁוֹטוֹת וְאֵין צָרֵךְ לְהַשְׁתֵּמֶשׁ בְּאַלְגּוֹרִיתְם מִיּוֹחֵד לְצָרֵךְ כֵּן).

טְבִלָת מַעֲבָרִים עַבְורָ אֹטוֹמַט המזוהה אֶת a+ :

מצב	a	b
1	2	—
2	2	—

הַמִּצְבַּח הַתְּחִلָּתִי הָוָא 1. 2 הוּא הַמִּצְבַּח הַמַּקְבֵּל הַיְיחִיד.

אֹטוֹמַט המזוהה אֶת abb :

מצב	a	b
3	4	—
4	—	5
5	—	6

המצב ההתחלתי הוא 3. 6 הוא המצב המתקבל היחיד.

אוטומט מקבל את  $a+b^*$  :

מצב	a	b
7	8	—
8	8	9
9	—	9

המצב ההתחלתי הוא 7. המצבים המתקבלים הם 8, 9.

הערה : שלושת האוטומטים הנ"ל הם דטרמיניסטיים. אבל לא היתה בעיה אילו הם לא היו דטרמיניסטיים. (זכור גם שפורמלית, כל אוטומט דטרמיניסטי נחשב גם לאוטומט לא דטרמיניסטי).

את האוטומט הלא דטרמיניסטי (הנקרא N למעלה) מקבלים על ידי יצרת מצב התחלתי חדש (נקרא לו מצב 0) וחיבורו בקשות אפסילון לכל אחד מהמצבים ההתחלתיים הנ"ל. באוטומט המתקבל יש רק מצב התחלתי אחד (מצב 0) ויש 4 מצבים מקבלים (2, 6, 8, 9).

הנה טבלת המעברים של אוטומט סופי דטרמיניסטי השקול לאוטומט N.

מצב	a	b	תבנית מוכרצת
0137	248	—	—
248	28	59	a+
28	28	9	a+
59	—	69	a+b*
9	—	9	a+b*
69	—	9	abb

במצב 248 לדוגמה, האוטומט מזזה התאמה גם לבניית  $a^+$  וגם לבניית  $b^*$ . בغالל ש-  $a^+$  מופיע קודם בראשית התבניות, ניתנת לו עדיפות. אילו  $a^+$  היה מופיע אחרי  $b^*$  לעולם לא היינו מכריזים על התאמה ל-  $a^+$  כי כל מחרוזת שמתאימה ל-  $a^+$  מתאימה גם ל-  $b^*$ .

הנה לדוגמה סידרת המצבים אותן יעבור המנתח עבור הקלט abbbbaa

סימוניים : החץ מצביע בכל שלב על סימן הקלט הראשון שהמנתח עדין לא קרא אותו. ההתאמה הארוכה ביותר שנמצאה בקלט (עד לאותו שלב) מסומנת בקו תחתון.

מספר	יתרთ הקלט	הערות
0137	abbbbaa ↑	
248	<u>abbbbaa</u> ↑	מצא התאמה ל- . a+
59	<u>a</u> bbbaa ↑	מצא התאמה ל- a+b*
69	<u>ab</u> bbbaa ↑	מצא התאמה ל- abb
9	<u>a</u> bbbaa ↑	מצא התאמה ל- a+b*
		"נתקע". מכריז על מציאת התאמה ל- a+b*. הלקסמה היא abbb (ארבעת הסימנים הראשונים בקלט).
0137	abbbbaa ↑	מתחילת חיפוש חדש. הלקסמה הבאה שתותאמ תתחיל בסימן הראשון שבא אחרי הלקסמה الأخيرة שהותאמ.
248	abbb <u>baa</u> ↑	מצא התאמה ל- a+
28	abbb <u>baa</u> ↑	מצא התאמה ל- a+
		הגיע לסוף הקלט. מכריז על התאמה ל- a+. הלקסמה היא aa (סימנים חמישי ושישי בקלט).

## תאור בסיסי של flex – כלי לייצור מנתחים לקסיקליים

תאור מלא של flex נמצא בקישור  
<http://flex.sourceforge.net/manual/>

ניתן להוריד קובץ הרצה של flex עבור WINDOWS מהקישור  
<http://gnuwin32.sourceforge.net/packages/flex.htm>

רץ גם על LINUX.

### כללי

על המשמש לספק ל- flex כללים (rules) שכל אחד מהם כולל ביטוי רגולרי וקוד בשפת C המשויך אליו. flex מקבל כקלט את הכללים האלה ומיציר קובץ בשפת C הנקרא c.y.lex. קובץ זה כולל הגדירה של פונקציה הנקראית () yylex. כאשר קוראים לפונקציה זו, היא מזזה בקלט שלא התאימה לביטויים הרגולריים אותם סייפק המשמש. כאשר () yylex מוצאת התאמה לביטוי רגולרי, היא מבצעת את הקוד המשויך אליו.

( ) yylex יכולה לשמש לצרכים שונים. אנו משתמש בה כמנתח לקסיקלי.

### קובץ הקלט של flex

קובץ הקלט ל-flex כולל שלושה חלקים :

```
definitions
%%
rules
%%
user code
```

#### החלק הראשון כולל :

-- הגדרות של שמות שמקילים על כתיבת הכללים (בחלק השני)

-- הכרזות על start conditions

-- קוד בשפת C שנכתב על ידי המשמש ומוועתק כפי שהוא לקובץ המוצע על ידי

flex.. קוד זה יופיע לפני הפונקציה () yylex. יש להזכיר קוד זה בצורה כזו :

```
% {
    user code
}%
```

(בנוסף לכך, כל שורה שמתחילה ב-whitespace מועתקת ללא שינוי לקובץ  
אותו מייצר flex).

בדרך כלל שמים בחלק זהה פקודות define, include ו命令ות שונות  
-- דברים שדרושים להופיע לפני () yylex. הקוד אותו כותב המשמש

בחלק מהכללים (בחלק השני של קובץ הקלט ל-`flex`) יופיע בתוך ()  
החלק השני כולל סידרה של כללים מהצורה  
 pattern action

התבנית (pattern) היא ביטוי רגולרי. action זה קוד בשפת C אותו מספק המשתמש.  
 קוד זה מבוצע כאשר מתגלה התאמה לתבנית.  
 התבנית חייבת להתחיל בתחילת השורה (לא whitespace לפניה). היא מסתירה את בתו הראשון שהוא whitespace (סימן whitespace מוקף בגרשיים). יתרת השורה מהוות את הוא מסומן באופן המתאים למשל הוא מוקף בסוגרים - . action אם ה- מתרחש על פני יותר משורה אחת יש להקיפו בסוגרים מסולסלות.

לפניהם הכלל הראשון ניתן כתוב קוד בשפת C שיעתק לתחילת הפונקציה ()`yylex`. לכן קוד זה יכול לשמש להגדרת משתנים לוקליים ל- ()`yylex` או להגדרת קוד שיתבצע כל פעם שנכנסים ל- ()`yylex`. את הקוד יש להקיף ב- { % } וב- { % } או להשאיר whitespace בתחילת כל שורת קוד (כמו במקרה הראשון).

החלק השלישי כולל קוד בשפת C שנכתב על ידי המשתמש. קוד זה מועתק כפי שהוא לסופו של הקובץ ש-`flex` מייצר (אחרי ()`yylex`).

tabniot b- flex  
 התבניות ב-`flex` מתוארות בעזרת קבוצה מורחבת של ביטויים רגולריים.

הנה רשימה (לא מלאה)

---

x	מתאים לתו 'x'
.	תו כלשהו בלבד newline
[xyz]	מתאים לכל אחד מהתווים הרשומים בתוך character class הסוגרים המרובעות (בדוגמה : מתאים ל-'x', ל-'y' או ל-'z')
[azc-fB]	מקף מצין טווח של תווים. (בדוגמה : מתאים ל-'a' או ל-'z', לכל אות בין 'c' ל-'f' וגם מתאים ל-'B').
[^A-Z]	מתאים לכל סימן בלבד המופיעים בקבוצה. (בדוגמה : מתאים לכל תו מלבד האותיות的大括號)
[^A-Z\n]	כל סימן בלבד אות גזולה ומלבד newline
r*	אפס או יותר z-ים (z ביטוי רגולרי כלשהו)
r+	r אחד או יותר
r?	z אופציוני (כלומר z אחד או אפס z-ים)
r{2, 5}	בין שניים לחמשה z-ים
r{2}	בדיקות שני z-ים
r{2,}	שני z-ים או יותר

---

---

{name}	name מוחלף בהגדתו (כפי שהוגדר בחלק הראשון של הקלט ל-
"[xy]\\"* foo"	מתאים למחוזות המוקפת בגרשיים (כדי שהגרשיים השניים יסמנו את התו גרשימים, מופיע ') לפנייהם). בדוגמה : מתאים למחוזות foo *[xy] (יש במחוזות 10 סימנים (הסימן השביעי הוא רווח)).
\X	אם X אחד מ- v a, b, f, n, r, t, oz המשמעות כמו ב-ANSI-C (לדוגמה '\n' זה newline). אחרת מתאים לו 'X'
(r)	מתאים למה שהビיטוי הרגולרי z מתאים
rs	שירשור של ביטויי רגולרי z עם ביטויי רגולרי s
r   s	r או s
r/s	r אבל רק אם בעקבותיו יש s
^r	z המופיע בתחילת שורה newline יש
r\$	z שלآخرיו יש
<s>r	כמו z אבל רק אם start condition שלו s
<<EOF>>	end of file

---

בתוך סוגרים מרובעים (character class), כל הסימנים שמייצגים אופרטורים מאבדים את משמעותם המינוחדת בלבד - '[', ']', '^' וכאשר הוא מופיע מיד אחר הסוגר המרובע הפתוח, גם '^'.

לכל אופרטור יש עדיפות. הנה חלק מהאופרטורים מסודרים לפי עדיפויותיהם, מהגבוהה לנמוכה. (לאופרטורים באותה השורה יש עדיפות זהה)

[], \*, +, ?, {}  
שרשור  
|  
/

### מה עושה הרוטינה ש-flex מייצר

flex מייצר רוטינה הנקראית () yylex(). כאשר () yylex המחזירה int. היא סורקת את הקלט שלה ומחפשת את המחרוזת הארוכה ביותר שהיא רישא של שאירית הקלט והמתאימה לאחת התבניות.

לדוגמה אם במקומות הנוכחי בקלט מופיעים התווים aabbbbac והתבניות הבאות מופיעות בקלט של flex .

a+  
a+b\*

או () yylex תקבע שההתאימה היא של מחרוזת הקלט aabbb למבנה השנייה. שימושו לב שתמצאה בקרה זה התאימות נוספת (a מתאימה לשתי התבניות,

aa מתאימה לשתי התבניות, aab מתאימה למבנה השנייה ו- aabb גם היא מתאימה למבנה השנייה (אבל נבחרה ההתאמה למחוזות הארכות ביותר).

אם יש יותר ממבנה אחד המתאימה למחוזות הארכות ביותר, () yylex בוחרת בתבנית שרשומה קודם בקובץ הקלט של flex.

לדוגמא אם הקלט שוב aabbbaac והמבנה הבאות מופיעות בקלט של flex

a+  
a+c\*bbb  
a+b\*

או () yylex תקבע שהמחוזות aabbbaac הותאמה למבנה השנייה בגין שזו רשומה לפני התבנית השלישייה (גם היא יכולה להתאים לאותה המחרוזת).

כשנקבעה ההתאמה, ה-action הצמוד למבנה שהותאמה מתבצע. ה-action יכול להשתמש במשתנים הגלובליים הבאים:  
yytext -- מצביע על טקסט שהותאם (yy יכול להיות מצביע ל-char או מערך של char -- לזכרינו זה לא משנה).  
yyleng – מכיל את האורך של הטקסט שהותאם.

אם לא נמצאה התאמה אז מבוצע כלל בירית המחדל:

. | \n ECHO;  
כלומר התו הבא בקלט נחשב כאילו הוא הותאם והוא נכתב לפט.  
ECHO כותב את yytext לפט (זה מקרו ש-flex דואג לכך שהגדתו תופיע בקובץ שהוא מיצר).

אם ה-action הוא ריק, אז האפקט הוא שהמחוזות שהותאמה "נזרקת".

לאחר ביצוע ה-action, () yylex תמשיך ותסrox את המשך הקלט כדי למצוא את ההתאמה הבאה. היא תמשיך ותמצא התאמות (ותבצע את actions המתאיםים) עד אשר היא הגיעו לסוף הקלט או עד אשר היא תבצע return action. במקרה האחרון היא תחזיר למי שקרה לה. בפעם הבאה שהיא תקרא היא תמשיך ותסrox את הקלט החל מהמקום הנוכחי שלה בו.

### פרטים נוספים שכדי לדעת

() yylex מחזירה אפס כאשר היא מגיעה לסוף הקלט. יש אפשרות שלא תפורט כאן לגרום לכך שהיא תמשיך ותקבל את הקלט שלה מקובץ נוסף (לכך משתמש הפונקציה yywrap -- היא אינה נחוצה לנו).

כדי לכלול את השורה %option noyywrap בחלק הראשון של קובץ הקלט ל- flex. אחריו בזמן שתקמפלו את התוכנית תקבלו הודעה שגיאת על כך שהפונקציה yywrap אינה מוגדרת.

() yylex ניגשת לקלט שלה דרך המשתנה הגלובלי yy (שטייפוסו

ברירת המחדל שלו היא `stdin`. אם רוצים שהקלט יגיע ממוקם אחר אפשר לעשות משהו כמו  
`yyin = fopen ("foo", "r");`  
`yylex ();`

הפלט של `ECHO` נכתב ל-`yyout`. ברירת המחדל היא `stdout` – ניתן לשנות את זה על ידי מתן ערך אחר ל-`yyout`.

`<foo>pattern action`      כל מהצורה start conditions  
יהיה פעיל רק כאשר נמצאים ב-`start condition` ששמו `foo`. (אין להשאיר רווח לפני התבנית).  
יש להזכיר על `start conditions` בחלק הראשון של קובץ הקלט ל-`flex`  
על ידי כתיבת `%` או `x %` ולאחריהם שמota של `start conditions`.

start condition מופעל על ידי פועלות `BEGIN` הבהא, עד לפועלות `BEGIN` הבאה, כללים עם ה-`start condition` שהופעל יהיו פעילים וככלים עם כללים לא יהיו פעילים. אם ה-`start condition` שהופעל הוכרז עם `x %` או גם כללים ללא `start condition` יהיו פעילים. אם הוא הוכרז עם `x %` אז רק הכללים עם ה-`start condition` שהופעל יהיו פעילים.

(0) `BEGIN` גורם לחזור למצב הרגיל שבו רק הכללים ללא `start conditions` הם פעילים. ניתן להתייחס למצב הרגיל כאל `start condition` ששמו `INITIAL` כך `BEGIN (INITIAL)` שקול ל-`(0)`.

הפעלת flex  
ניתן להפעיל את `flex` בשורת הפקודה של `cmd.exe` באופן הבא:  
`flex input-file-name`  
`lex.yy.c` ייצור את הקובץ `flex`

האופציה `-d` – טובה לצורך דיבוג:  
`flex -d input-file-name`

עכשו `yylex` תדפיס הודעה המתארות את התאמות שהיא מצאה (מה הביטוי הרגולרי ולאיזה מחרוזות בקלט הוא התאים).

כדי לקבל קובץ הרצה יש כמובן לкопרל את `lex.yy.c` (עם קבצי `C` נוספים בהתאם לצורך) בעזרת קומpileר לשפת `C`.  
אם משתמשים ב- `Visual C` אפשר ליצור פרויקט מסווג `.lex.yy.c` שמכיל את הקובץ `Win32 Console Application`

## פונקציה FIRST

שיטת לניתוח תחבירי אותו נלמד בהמשך עשוות שימוש בפונקציה זו.

הגדרה : FIRST זו פונקציה שפועלת על מחרוזות של סימני דקדוק. ( $\alpha$ ) FIRST ו- קבוצת כל הטרמינלים שמתחלילים מחרוזות שנגזרות מ- $\alpha$ . כלומר טרמינל  $a$  שייך FIRST אם ורק אם  $a\beta \Rightarrow^* \alpha$  עבור איזו שהיא מחרוזת  $\beta$ . אם  $\epsilon \Rightarrow^* \alpha$  או גם  $\epsilon$  שייכת ל- FIRST .  
הערה : הסימון  $*$   $\Rightarrow$  משמעותו "גוזר באפס או יותר צעדים".

נגידר גם שעבור טרמינל כלשהו  $a$   $\{ a \} = \text{FIRST}(a)$

נתבונן לדוגמא בדקדוק הבא (נשתמש כאן במוסכמה שהאותיות הקטנות (כולל אלו עם אינדסימים כמו  $a_1$ ) מייצגות טרמינלים. האותיות的大ות מיצגות משתנים) :

- (1)  $S \rightarrow ABCd$
- (2)  $A \rightarrow a_1 b_2$
- (3)  $A \rightarrow a_2 H$
- (4)  $A \rightarrow \epsilon$
- (5)  $B \rightarrow b_1 h$
- (6)  $B \rightarrow b_2 D H$
- (7)  $B \rightarrow G H$
- (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)  $H \rightarrow \epsilon$

דוגמא : נראה מה היא הקבוצה  $\text{FIRST}(ABCd)$ .

הגזירה הבאה מראה שהטרמינל  $a_1$  נמצא ב-  $\text{FIRST}(ABCd)$  :  $ABCd \Rightarrow a_1 b_2 B C d$

הגזירה הבאה מראה שהטרמינל  $a_2$  גם הוא נמצא ב-  $\text{FIRST}(ABCd)$  :  $ABCd \Rightarrow a_2 H B C d$

קל לראות שבאופן כללי כל טרמינל (להבדיל מאפסילון) שנמצא ב-  $\text{FIRST}(A)$  נמצא ב-  $\text{FIRST}(ABCd)$  וזאת משום ש-  $A$  הוא הסימן הראשון במחרוזת  $ABCd$ .

אבל בגלל שם-  $A$  (הסימן הראשון במחרוזת  $ABCd$ ) ניתן לגוזר  $\epsilon$ , כל טרמינל

שנמצא ב- FIRST של הסימן השני של ABCd (כלומר ב- (B) גם הוא נמצא ב- FIRST (ABCd). הנה לדוגמה גזירה המראה ש-  $b_1$  נמצא ב- FIRST (ABCd).

בגזרה זו קודם נגזר מ- A את ε ולאחר מכן נגזר מ- B מחרוזת המתחילה ב-  $b_1$ .  
 $ABCd \Rightarrow BCd \Rightarrow b_1 h C d$

הנה גזירה המראה ש-  $g$  גם הוא ב- FIRST (ABCd) (גוזרים ε מ- A ולאחר מכן גוזרים מ- B מחרוזת המתחילה ב-  $g$ ).

$ABCd \Rightarrow BCd \Rightarrow GHcd \Rightarrow gHcd$

לא קשה לראוץ שם- B ניתן גם לngזר מחרוזות המתחילות ב-  $b_2$  או ב-  $h$ .  
 לכן גם  $b_2$  ו-  $h$  בנוסף ל-  $b_1$  ו-  $g$  נמצאים ב- FIRST (B) וכך גם ב- FIRST (ABCd).

מ- B ניתן גם לngזר ε ( $\epsilon$ )  $\Rightarrow H \Rightarrow GH \Rightarrow (B)$ . יוצא שoczל אחד משני הסימנים הראשונים בחרוזת ABCd ניתן לngזר ε. לכן כל טרמינל שנמצא ב- FIRST של הסימן השלישי ב- ABCd (כלומר ב- (C)) גם הוא יהיה ב- FIRST (ABCd). הנה גזירה המראה ש-  $c$  נמצא גם הוא ב- FIRST (ABCd).  
 ב- FIRST (ABCd). הנה גזירה המראה ש-  $c$  נמצא גם הוא ב- FIRST (ABCd).  
 קודם נגזר מ- A את ε ולאחר מכן נגזר מ- B את ε ולבסוף נגזר מ- C מחרוזת המתחילה ב-  $c$ :

$ABCd \Rightarrow BCd \Rightarrow GHcd \Rightarrow HCd \Rightarrow Cd \Rightarrow cd$

גם מ- C ניתן לngזר ε. יוצא שoczל אחד משלושת הסימנים הראשונים בחרוזת ABCd ניתן לngזר ε. לכן גם כל טרמינל שנמצא ב- FIRST של הסימן הרביעי (כלומר ב- (d)) גם הוא יהיה ב- FIRST (ABCd). (לפי ההדרה,  
 $\{d\} = \{FIRST(d)\}$  כי ההדרה אומרת ש- FIRST של טרמינל כולל רק את הטרמינל עצמו).

הנה גזירה המראה ש-  $d$  נמצא גם הוא ב- FIRST (ABCd).  
 קודם נגזר מ- A את ε ולאחר מכן נגזר מ- B את ε, ולאחר מכן נגזר מ- C את ε  
 ואז קיבל מחרוזת שמתחליה ב-  $d$ :

$ABCd \Rightarrow BCd \Rightarrow GHcd \Rightarrow HCd \Rightarrow Cd \Rightarrow d$

שימוש לב שלא ניתן לngזר את ε מהחרוזת ABCd ולאחר ε איןנו נמצא ב- FIRST (ABCd). בסך הכל מתקיים:  
 $FIRST(ABCd) = \{a_1, a_2, b_1, b_2, g, h, c, d\}$

מהחרוזת ABC כן ניתן לngזר ε כי ראיינו כבר שניתנו לngזר ε מכל אחד מהסימנים המופיעים בה. לכן FIRST(ABC) כן יכול את ε. בדוקו הנטוון מתקיים:  
 $FIRST(ABC) = \{a_1, a_2, b_1, b_2, g, h, c, \epsilon\}$

הנה קבוצות ה- FIRST של כל משתני הדקדוק הנטוון:  
 $FIRST(S) = \{a_1, a_2, b_1, b_2, g, h, c, d\}$

$\text{FIRST}(A) = \{a_1, a_2, \epsilon\}$

$\text{FIRST}(B) = \{b_1, b_2, g, h, \epsilon\}$

$\text{FIRST}(C) = \{c, \epsilon\}$

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

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

$\text{FIRST}(H) = \{h, \epsilon\}$

שימוש לב שבאופן כללי, אם יש כלל גזירה מהצורה  $A \rightarrow \alpha$  → A (כאשר A משתנה ו-  $\alpha$  מחרוזת של סימני דקדוק (משתנים או טרמינלים)) אז כל אלמנט של הקבוצה  $(\alpha)$  FIRST נמצא גם ב-  $(A)$ .

אם למשל  $(\alpha)$  כולל את הטרמינל a אז טרמינל זה נמצא גם ב-  $(A)$  FIRST : מ- A ניתן לגזור את  $\alpha$  (לפי הכלל  $A \rightarrow \alpha$ ) ואו מ-  $\alpha$  ניתן לגזור מחרוזת המתחילה ב- a (שהרי הנחנו ש- a נמצא ב-  $(\alpha)$  FIRST). יוצא ש- A ניתן לגזור מחרוזת המתחילה ב- a כלומר a אכן נמצא ב-  $(A)$  FIRST.

בצורה דומה ניתן לראות שאם אפסילון נמצא ב-  $(\alpha)$  FIRST הוא נמצא גם ב-  $(A)$  FIRST. במקרים אחרים אם ניתן לגזור את אפסילון מ- A אז ניתן לגזור את אפסילון גם מ- A. (פשוט גוזרים מ- A את  $\alpha$  ומן  $\alpha$  גוזרים אפסילון).

### אלגוריתם לחישוב FIRST

הנה אלגוריתם לחישוב FIRST של מחרוזת כלשהי. לצורך כך יש לחשב קודם FIRST של כל אחד ממשתני הדקדוק.

שלב 1 תחילת נחשב את FIRST(X) עבור כל סימן דקדוקי X. אם X הוא טרמינל אז FIRST(X) יכיל את X בלבד.

נחשב את FIRST(X) עבור כל משתני הדקדוק באופן הבא.  
עבור כל משתנה X, נתחל את FIRST(X) לקבוצה הריקה.  
עבור כל גזירה מהצורה  $\epsilon \rightarrow X$  נוסיף את  $\epsilon$  ל- FIRST(X).

בוצע את הצעדים הבאים עד אשר אין שינוי באף אחת מקבוצות ה-FIRST.  
עבור כל גזירה מהצורה  $\epsilon \rightarrow Y_1 Y_2 \dots Y_k$  → X נוסיף את כל הטרמינלים ב- FIRST(Y<sub>1</sub>) ל- FIRST(X). (יתכן שה- FIRST(Y<sub>1</sub>) כולל גם את  $\epsilon$  אבל אותו לא מוסיפים).

עבור כל  $k \leq i < 1$  אם  $\epsilon \in \text{FIRST}(Y_1 \dots Y_{i-1})$  FIRST(Y<sub>1</sub>), FIRST(Y<sub>2</sub>) ... FIRST(Y<sub>i-1</sub>) ⇒ \* (כלומר  $\epsilon \Rightarrow^*$  FIRST(Y<sub>1</sub> ... Y<sub>i-1</sub>)).  
או נוסיף את כל הטרמינלים ב- FIRST(Y<sub>i</sub>) ל- FIRST(X).  
אם  $\epsilon \in \text{FIRST}(Y_i)$  אז נוסיף את  $\epsilon$  ל- FIRST(X).

כלומר בכל מקרה מוסיפים את כל הטרמינלים ב- FIRST (Y<sub>1</sub>) ל- (X).  
אם לא ניתן לגזור מ- Y<sub>1</sub> את ε אז לא מוסיפים ל- FIRST (X) שום דבר נוסף. אבל אם ε \* ⇒ Y<sub>1</sub> אז מוסיפים את הטרמינלים ב- FIRST (Y<sub>2</sub>). אם גם מ- Y<sub>1</sub> וגם מ- Y<sub>2</sub> ניתן לגזור את ε אז מוסיפים גם את הטרמינלים ב- FIRST (Y<sub>3</sub>) ל- (X) וכן הלאה.

בנוסף לכך, אם ניתן לגזור מכל ה- Y – ים את ε, אז מוסיפים את ε FIRST (X).

## שלב 2

לאחר חישוב ה- FIRST של כל משתני הדקדוק ניתן לחשב את FIRST עבור מחרוזת של סימני דקדוק X<sub>1</sub>X<sub>2</sub> ... X<sub>n</sub> באופן הבא.

נתחל את FIRST (X<sub>1</sub>X<sub>2</sub> ... X<sub>n</sub>) לקבוצה הריקה. נוסיף את כל הטרמינלים ב- FIRST (X<sub>1</sub>) ל- FIRST (X<sub>1</sub>X<sub>2</sub> ... X<sub>n</sub>). אם FIRST (X<sub>1</sub>X<sub>2</sub> ... X<sub>n</sub>) כולל את ε אז נוסיף גם את כל הטרמינלים ב- FIRST (X<sub>2</sub>) ל- FIRST (X<sub>1</sub>X<sub>2</sub> ... X<sub>n</sub>).  
אם ε נמצא גם ב- FIRST (X<sub>1</sub>) וגם ב- FIRST (X<sub>2</sub>) אז נוסיף את כל הטרמינלים ב- FIRST (X<sub>3</sub>) ל- FIRST (X<sub>1</sub>X<sub>2</sub> ... X<sub>n</sub>) וכן הלאה.  
לבסוף, נוסיף את ε ל- FIRST (X<sub>i</sub>) אם עבור כל i FIRST (X<sub>1</sub>X<sub>2</sub> ... X<sub>n</sub>) מכיל את ε.

## פונקציה FOLLOW

זו פונקציה שפועלת על משתנים של הדקדוק. FOLLOW

### הגדרה

(FOLLOW (A) – כאשר A זה משתנה, היא קבוצת הטרמינלים שיכולים להופיע מיד אחרי A (ימינה ממנו) באיזו שהייה תבנית פסוקית. כלומר טרמינל a שייך ל- FOLLOW (A) אם ורק אם  $\alpha A a \beta \Rightarrow^* S$  עבור איזה מהם מחרוזות של סימני דקדוק  $\alpha$  ו-  $\beta$ . אם A יכול להיות הסימן הימני ביותר באיזו שהייה תבנית פסוקית אז גם  $\$$  נמצא ב- FOLLOW (A). (\$) הוא סימן מיוחד שנשתמש בו לציוון סוף הקלט).

(תזכורת: **מבנה פסוקית** (sentential form) היא מחרוזת של משתנים וטרמינלים הניתנת לגזירה (לא משנה בכמה צעדים) מהמשתנה ההתחלתי).

נשתמש בדקדוק הבא בדוגמאות.

- (1)  $S \rightarrow DzDaG$
- (2)  $D \rightarrow ABG$
- (3)  $A \rightarrow Dh$
- (4)  $A \rightarrow a$
- (5)  $B \rightarrow b_1 h$
- (6)  $B \rightarrow b_2$
- (7)  $B \rightarrow \epsilon$
- (8)  $D \rightarrow d$
- (9)  $G \rightarrow g$
- (10)  $G \rightarrow B$

: בדוק זה מתקיים

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

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

$$\text{FIRST}(B) = \{ b_1, b_2, \epsilon \}$$

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

$$\text{FIRST}(G) = \{ b_1, b_2, g, \epsilon \}$$

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

$$\text{FOLLOW}(A) = \{ a, b_1, b_2, g, h, z \}$$

$$\text{FOLLOW}(B) = \{ a, b_1, b_2, g, h, z, \$ \}$$

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

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

: בדוק זה ישן אינסוף מבניות פסוקיות (sentential forms).

המחרוזת  $DzDaG$  היא אחת מהן (היא מבנית פסוקית כי היא ניתנת לגזירה

$$S \Rightarrow DzDaG$$

מהמשתנה ההתחלתי  $S$ ).

במבנה פסוקית זו,  $z$  מופיע מיד אחרי  $D$ . לכן  $z$  נמצא ב- (D) FOLLOW. גם  $a$  מופיע מיד אחרי  $D$  ולכן גם  $a$  נמצא ב- (D) FOLLOW. המשתנה  $G$  הוא הסימן הימני ביותר במבנה הפסוקית זו והוא נמצא \$. נמצא ב- (G) FOLLOW.

המחרוזת  $ABGzDaG$  היא דוגמא נוספת לבניה פסוקית כי גם היא ניתנת לגזירה מ-  $S$ :

$$S \Rightarrow DzDaG \Rightarrow ABGzDaG$$

מהמבנה הפסוקית  $ABGzDaG$  ניתן לראות ש-  $z$  נמצא ב- (G) FOLLOW, ש-  $a$ , ש-  $D$  ו- \$. נמצאים ב- (D) FOLLOW.

הערות: שימוש לב-  $S$  מוגדר רק על משתנים. זאת אומרת ש-  $(a)$  FOLLOW (כאשר  $a$  הוא טרמינל) אינו מוגדר.

לפי ההגדרה, (A) FOLLOW (כאשר A משתנה כלשהו) לעולם אינו כולל את אפסילון. זאת לעומת (α) FIRST (כאשר α מחרוזת כלשיי) שכן כולל את אפסילון כאשר ניתן לגזור את אפסילון מ- α. (A) FOLLOW עשוי לכלול את \$. לעומת זאת (α) FIRST, לפי הגדרה, לעולם אינו כולל את \$.

(S) FOLLOW כאשר S המשתנה הראשוני תמיד כולל את \$. זאת משום ש-  $S$  היא הסימן הימני ביותר (במקרה זה גם היחיד) במבנה הפסוקית  $S$ . שימוש לב-  $S$  היא תבנית פסוקית כי היא ניתנת לגזירה (באפס צעדים) מ-  $S$ . כמובן ש- (S) FOLLOW עשוי לכלול גם טרמינלים נוספים ל- \$.

### הסברים על האלגוריתם לחישוב FOLLOW (האלגוריתם מופיע בהמשך).

האלגוריתם מותבס על כך שoczם מופיע של משתנה באגף ימין של כלל גזירה ניתן להסיק שטרמינלים מסוימים (\$) נמצאים ב- FOLLOW של אותו המשתנה.

בהמשך נשתמש בסימונים הבאים בהתאם למוסכמות הרגילות שלנו:  $\gamma$ ,  $\beta$ ,  $\alpha$ , מצינים מחרוזות של סימני דקדוק (טרמינלים ומשתנים). (שימוש לב-  $S$  כולל גם את המחרוזת הריקה אפסילון).

(B) FOLLOW על סמך כלל גזירה שבו B מופיע באגף ימין.

מבחינים בין שלושה מקירים אפשריים.

מקרה ראשון במקרה זה B מופיעה באגף ימין של כלל גזירה והוא אינו הסימן הימני ביותר באגף ימין.

מכל גזירה מהצורה  $\alpha B \beta \rightarrow A$  ניתן להסיק שככל הטרמינלים (אבל לא ε) ב- (β) FIRST נמצאים ב- (B) FOLLOW.

הוכחה : נניח שהטרמינל  $\$$  נמצא ב- (β). נראה שהוא נמצא ב- (B) FOLLOW. התבנית הפסוקית האחורונה בגזירה הבאה מראה שאכן  $\$$  נמצא ב- (B) FOLLOW.

$$S \Rightarrow^* \gamma_1 A \gamma_2 \Rightarrow^* \gamma_1 \alpha B \beta \gamma_2 \Rightarrow^* \gamma_1 \alpha B \beta \gamma_3 \gamma_2$$

כאשר  $\gamma_3, \gamma_2, \gamma_1$  הן מחרוזות של סימני דקדוק.

הסבר : אנו מניחים שהדקוק עושה שימוש במשתנה A כולם ניתן לנזר מהמשתנה התחלתי מחרוזת המכילה את A (אחרת A אינו תורם דבר לדקדוק ונitin לוותר עליו מבלי לשנות את השפה שהדקוק מגדר). זו הצדקה לשלב הראשון בגזירה הניל. הצעד הבא בגזירה הוא פשוט גזירה של A לפי כלל הגזירה  $\alpha B \beta \rightarrow A$ .

הנחנו ש-  $\$$  נמצא ב- (β) FIRST. מכאן שקיימות גזירה כזו :  $\gamma_3 \Rightarrow^* b \gamma_2$ . וזה מסביר את השלב השני בגזירה הניל.

מקרה שני : במקרה זה B הוא הסימן הימני ביותר באגף ימין של כלל הגזירה. מכל גזירה מהצורה  $\alpha B \rightarrow A$  ניתן להסיק שככל האברים (טרמינלים או \$) הנמצאים ב- FOLLOW נמצאים גם ב- (B) FOLLOW.

הוכחה : נניח ש-  $\$$  הוא טרמינל ב- FOLLOW. נראה ש-  $\$$  נמצא גם ב- (B) FOLLOW. מהנחה ש-  $\$$  נמצא ב- FOLLOW ניתן להסיק שקיימות גזירה כזו :  $\gamma_2 \Rightarrow^* \gamma_1 A \gamma_3$  כאשר  $\gamma_2, \gamma_1$  הן מחרוזות של סימני דקדוק.inium

נשיך גזירה זו על ידי גזירת A לפי הכלל הנתון :

$$S \Rightarrow^* \gamma_1 A \gamma_2 \Rightarrow^* \gamma_1 \alpha B \beta \gamma_2$$

התבנית הפסוקית שקיבלנו מראה ש-  $\$$  אכן שייך גם ל- (B) FOLLOW.

קל לראות באופן דומה שבמקרה ש-  $\$$  נמצא ב- (A) הוא בהכרח נמצא גם ב- (B) FOLLOW :

מהנחה ש-  $\$$  נמצא ב- FOLLOW ניתן להסיק שקיימות גזירה כזו :  $\gamma \Rightarrow^* S$  כאשר  $\gamma$  מחרוזת של סימני דקדוק. נשיך גזירה זו על ידי גזירת A לפי הכלל הנתון :

$$S \Rightarrow^* \gamma A \Rightarrow^* \gamma \alpha B$$

התבנית הפסוקית שקיבלנו מראה ש-  $\$$  אכן שייך גם ל- (B) FOLLOW.

מקרה שלישי :

मכל גזירה מהצורה  $\alpha B \beta \rightarrow A$  כאשר  $\epsilon \Rightarrow^* \beta$  ניתן להסיק שככל האברים (טרמינלים או \$) הנמצאים ב- FOLLOW נמצאים גם ב- (B) FOLLOW. ההוכחה דומה להוכחה של המקרה השני.

הערה

שים לב שהאלגוריתם לשימוש FOLLOW אינו עושה שימוש ישיר בכללי גזירה שבהם לא מופיעים משתנים באגף ימין. (כללי גזירה כאלו כן משפיעים על ה- FOLLOW כי הם משפיעים על חישובי ה- FIRST שהאלגוריתם ל- FOLLOW עושה בהם שימוש).

הערה

כאשר משתנה מופיע כסימן הימני ביותר בכל גזירה, אין זה אומר בהכרח ש- \$ נמצא ב- FOLLOW של משתנה זה.  
לדוגמא נתבונן בדקדוק הבא (S הוא המשתנה הראשוני) :

$$S \rightarrow cAz$$

$$A \rightarrow aB$$

$$B \rightarrow b$$

בדקדוק זה מתקיים  $\text{FOLLOW}(B) = \{ z \mid z \text{ מופיע כסימן הימני ביותר בכל גזירה השני, } \text{FOLLOW}(B) \text{ אינו כולל את } \$ \text{.}$   
בגלל שהדקדוק הזה כל כך פשוט, לא קשה לראות שיש רק תבנית פסוקית אחת  
שהה מופיע B והוא  $caBz$ . בתבנית פסוקית זו z מופיע אחרי B.

אם נחליף את הכלל הראשון בכלל  $A \rightarrow cA$  נקבל  $\{\$ \mid \text{FOLLOW}(B) = \$ \text{ וגם הפעם יש רק תבנית פסוקית אחת שבה מופיע B אבל הפעם סימן הימני ביותר בתבנית זו : } caB \}$

### אלגוריתם לחישוב FOLLOW

איתחול: נתחל את (S) FOLLOW (כאשר S הוא המשתנה הראשוני) כך שיכיל רק את \$. עברו כל משתנה אחר A, נתחל את (A) FOLLOW לקובצת הריקה.

- נפעיל את הכללים הבאים עד אשר אין שינוי באף אחת מקבוצות ה-FOLLOW.
1. עברו לכל גזירה מהצורה  $\alpha B \beta \rightarrow A$  ווסף את כל הטרמינלים  $B - (\beta) \text{ FIRST } L - (\beta) \text{ FOLLOW}$ .
  2. עברו לכל גזירה מהצורה  $\alpha \rightarrow A$  או מהצורה  $\alpha B \beta \rightarrow A$  כאשר  $(\beta) \text{ FIRST } \epsilon \text{ מכיל את } \epsilon$  (כלומר  $\epsilon^* \Rightarrow \beta$ ) ווסף את כל מה שנמצא  $B - (\beta) \text{ FOLLOW } L - (\beta) \text{ FOLLOW } (A)$ .

### דוגמאות

נדגים חלק מהצעדים שיבצע האלגוריתם על הדקדוק שמופיע בהתחלה מסמך זה.

\$ יוכנס ל- (S) FOLLOW (זה בשלב האיתחול).

הטרמינל z יוכנס ל- (D) FOLLOW בעקבות הפעלת כלל מס' 1 באלגוריתם על כל גזירה הראשונית. (שםו לב ש-  $\{ z \mid \text{FIRST}(zDaG) = \}$ ).

הטרמינל a יוכנס ל- (D) FOLLOW בעקבות הפעלת כלל מס' 1 באלגוריתם על כל גזירה הראשונית. (שםו לב ש-  $\{ a \mid \text{FIRST}(aG) = \}$ ).

\$ יוכנס ל- (G) FOLLOW בעקבות הפעלת כלל מס' 2 באלגוריתם על כל גזירה הראשונית. (G הסימן הימני ביותר באגף ימין של כל גזירה ו- \$ נמצא ב- FOLLOW(S)).

הטרמינליים  $g, b_1, b_2$  יוכנסו ל- FOLLOW (A) בעקבות הפעלת כלל מספר 1 באלגוריתם על כלל הגזירה השני. שימו לב ש- FIRST (BG) = {  $b_1, b_2, g, \epsilon$  }  
בנוסף לכך, גם  $a, z$  (שבשלב זה FOLLOW (D) מכיל רק אותן), יוכנסו גם הם ל- FOLLOW (A) וזאת בעקבות הפעלת כלל מספר 2 באלגוריתם על כלל הגזירה השני. (כאן A נמצא בתפקיד של B בתאור כלל 2 באלגוריתם, BG הם ה-  $\beta$  שמופיעים באלגוריתם שמננו ניתן לגוזר את אפסילון, D בתפקיד ה- A המופיע באלגוריתם ו-  $\alpha$  המוזכר באלגוריתם הוא ריק).

מה ניתן להוסיף בשלב זה ל- FOLLOW (B) לפי כלל הגזירה השני ?

הפעלה של כלל מספר 1 של האלגוריתם על כלל הגזירה השני מראה FIRST (G) = {  $b_1, b_2, g, \epsilon$  } FOLLOW (B) (כי {  $b_1, b_2, g$  } ל- FOLLOW (B) יש להוסיף את  $\epsilon$ ).

הפעלה של כלל מספר 2 של האלגוריתם על כלל הגזירה השני מראה שיש להוסיף את  $z, a$  (הנמצאים ב- FOLLOW (D) ל- FOLLOW (B) כי מ- G (בתפקיד ה-  $\beta$  בתאור הכלל) ניתן לגוזר את  $\epsilon$ ).

גם ל- FOLLOW (G) יש להוסיף את  $z, a$  (הנמצאים ב- FOLLOW (D) בעקבות הפעלה של כלל מספר 2 באלגוריתם על כלל הגזירה השני).

כאן הובא רק חלק מהלך האלגוריתם על הדקוק הנוכחי. התוצאה הסופית של חישובי ה- FOLLOW על דקוק זה מופיעה לעיל.

## bison – כלי לייצור מנהכים תחביריים

הערה: כאן יש הסברים בסיסיים. מדריך מكيف על bison ניתן למצוא בקישור [www.gnu.org/software/bison/manual/](http://www.gnu.org/software/bison/manual/)

כללי: bison הוא parser generator קלומר כלי המיציר באופן אוטומטי מנתח תחבירי (parser) עבור דקדוק המוגדר על ידי המשתמש. ה-parser כתוב ב- C והוא מסוג shift reduce parser המשמש בטבלת (1) LALR. יש אפשרות לייצר טבלאות מסווגים נוספים.

המשתמש יכול להגדיר פעולות סמנטיות (semantic actions) בשפת C אותן יבצע ה-parser במהלך הניתוח התחבירי. פעולות אלו יכולות לעשות דברים שונים כמו למשל לתרגם את הקלט לאיזו שהיא שפה מטרית או לבנות syntax tree המציג את הקלט.

נוח מאד להשתמש ב-bison וב-flex ביחד. המנתח התחבירי המיציר על ידי bison קורא לפונקציה () yylex כאשר הוא נזקק לאיסימון הבא בקלט. ניתן להשתמש ב-flex כדי ליצור אוטומטית את () yylex. לחילופין, ניתן לכתוב באופן ידני את yylex.

bison פותח כחלק מפרויקט GNU שמטרתו פיתוח מערכת חופשית דומה ל- Unix. (המערכת המוכרת היום כ-Linux פותחה בחלוקת הגודל במסגרת GNU כך שיש לקרוא לה GNU/Linux). ראו <http://www.gnu.org/why-gnu-linux.html> bison הוא התחליף yacc של GNU ל- yacc שהוא ידועה שרצה על Unix.

.Java .C++ גם בשפות bison תומך גם ב- .

כדי לדעת שיש הרבה מאוד parser generators נוספים התומכים בשפות שונות.

קובץ הקלט של bison  
קובץ הקלט ל-bison כולל את החלקים הבאים :

```
%{  
    C code  
}%  
bison declarations  
  
%%
```

grammar rules

```
%%  
C code
```

הקובץ מחולק לשלווה של שלושה חלקים המופרדים ע"י %.  
הנה תאור קצר של חלקים אלו :

הכרזות עבור bison

בחלק זה מופיעות ההכרזות מסווג `%union`, `%token`, `%type` ועוד. חלק מההכרזות תתוארנה בהמשך. הכרזות ל- `bison` מתחילה ב- %.

בחלק הראשון ניתן לרשום גם קטעי קוד בשפת C מוקפים ע"י { % }. `bison` מעתיק את אלו ללא שינוי לתחילת הקובץ אותו הוא מייצר (קובץ בשפת C שבו מוגדר ה- `parser`). בדרך כלל יופיעו כאן הכרזות על משתנים גלובליים, על פונקציות (בינהן `yyerror()` ו- `yylex()`) ועל טיפוסים בהם משתמשים בפעולות (actions) שמופיעות בכללי הדקדוק. כמו כן מקובל לשים כאן פקודות `#include` ו- `#define`.

במוקום להקייף את ההכרזות ב- %...% ניתן להקייף אותן ב-

```
%code {
    C code goes here
}
```

במוקום `%code` ניתן להשתמש גם ב-

```
%code requires
%code provides
%code top
```

הבדל בין האלטרנטיבות הללו הוא במיקום של הקוד בקובץ (או בקבצים – `bison` מייצר גם `include file` אם מפעילים אותו עם האופציה `-d`) שמייצר `.bison`.

אם אין סיבה לנוהג אחרית יש להשתמש ב- `.%code` במשך מקרה שבו צריך להשתמש ב- `.%code requires`.

החלק השני בקובץ הקלט ל- `bison` מכיל את כללי הגזירה של הדקדוק בהם משולבים פעולות סמנטיות. (ביחד הם מהווים "סכימת תרגום").

החלק השלישי כולל קוד נוסף בשפת C. קוד זה מועתק כפי שהוא לסוף של הקובץ `bison`. לעיתים נוח לשים כאן פונקציות הנקראות על ידי הפעולות הסמנטיות. ניתן לשים כאן גם את הקוד עבור `yylex()`.

### כתיבת כללי הדקדוק (grammar rules)

#### שמות של סימני דקדוק

למשתנים (non terminals) ולאיסימונים (tokens או terminals (טרמינלים)) יש שמות שיכולים לכלול אותיות, ספרות (לא בהתחלה) וקו תחתון. שמות של משתנים יכולים לכלול גם נקודה. המוסכמה היא ששמות של משתנים נכתבים באותיות קטנות (לדוגמא `stmt`) ושמות של איסימונים – באותיות גדולות (לדוגמא `ID`).

יש להזכיר על כל שם של איסימון. לדוגמה: `%token ID`

אין צורך להזכיר על משתנים כדי לומר ל-`bison` מהם משתנים (אם כי לעיתים יש צורך להזכיר על משתנה כדי לציין את הטיפוס של ערכו הסמנטי).

יש צורה נוספת שבה ניתן לכתוב איסימון: באותה צורה שבה כתבים בשפת C קבועים מטיפוס `char` למשל '+' או 'ח' (אלו נקראים character literals). לפי

המוסכמה, אסימונו הנקתב בצורה כזו מייצג את האסימון המורכב מהתו. אין צורך להזכיר על אסימונו כזה כדי לומר ל-*bison* שהוא אסימונו (אם כי לעיתים יש צורך להזכיר עליו לצורך אחר למשל כדי לציין מה הטיפוס של ערכו הסמנטי).

יש צורה שלישית לכתיבה אסימונים: אלו הם `.literal string tokens`: אלו אסימונים אלו נכתבים כמו מחרוזות בשפת C (למשל `"<=`"). ב-*manual* מופיעים פרטיים נוספים. לא נשמש בצורה כתיבה זו בהמשך.

#### איך נראים כללי הגזירה

התחברר של כללי הגזירה דומה לזה שלמדו אבל במקום חץ מופיעים נקודותיים ובסוף כל כלל גזירה מופיעה נקודה פסיק. לדוגמה:

```
expr : expr '+' term
      ;
expr: term
      ;
```

משמעותי רק ספיריד בין שמות – ניתן להוסיף whitespace כרצונו. ניתן לצרף ביחד כללי גזירה עם אותו המשתנה מצד שמאל. למשל את שני הכללים הניל ניתן כתוב בקיצור כך:

```
expr : expr '+' term
      | term
      ;
```

צד ימין של כלל גזירה עשוי להיות ריק. (המשמעות היא כאלו היה כתוב אפסילון מצד ימין). במקרה זה מקובל לכתוב העלה `/* empty */` מצד ימין. לחילופין ניתן לרשום `%empty` מצד ימין אבל אפשרות זו לא קיימת בגרסאות אחרות של *bison*.

לדוגמה, הכללים הבאים מတאים סידרה של אפס או יותר מספרים המופרדים על ידי פסיקים:

```
list : /* empty */
      | list1
      ;
list1 : NUM
      | list1 ',' NUM
      ;
```

#### המשנה ההתחלתית

המשנה ההתחלתית של הדקדוק הוא זה שופיע מצד שמאל של כלל הגזירה הראשון. זו היא ברירת המחדל אבל ניתן לציין משתנה ההתחלתית אחר בעזרת הכרזה כמו לדוגמה `%start program`. כאן אמרנו שהמשנה ההתחלתית הוא `program`. הכרזה כזו יכולה להופיע בקובץ הקלט של *bison* בחלק המיעוד `bison`. להכרזות עבר *bison*.

#### פעולות (actions)

פעולה ב-*bison* היא קטע קוד בשפת C המוקף בסוגרים מסולסלות והמופיע מצד ימין של כלל גזירה. כאשר המנתה התחברי עשויה `reduce` לפיה כל גזירה מסוים, הוא מבצע את הפעולה המופיעה בקצת הימני של הכלל (אם ישנה כזו).

לדוגמה:

```
date: day month year { printf("Israeli style date\n"); }
```

```
| month day year { printf("American style date\n"); }
```

כאן תודפס הodata בהתאם לסגנון התאריך. כאשר המנתח יעשה reduce לפי הכלל הראשון – תודפס ההodata הראשונה. כאשר הוא ימצטם לפי הכלל השני – תודפס ההodata השנייה.

bison אינו מבין את הקוד בפעולות – הוא פשוט מעתיק אותו לקובץ שהוא מייצר (עם שינויים קלים: החלפה של \$2, \$1 וכיו' לקוד בשפת C – ראו בהמשך).

פעולה יכולה להופיע גם "באמצע" צד ימין של כלל גזירה, ככלומר לא ממש בקצתה הימני. פעולות כאלה מכונות mid rule actions. מועד ביצועה של פעולה כזו נקבע בהתאם למיקומה בכלל הגזירה.  
לדוגמא:

```
two_colors: {printf ("one "); }
             color {printf ("two "); }
             color { printf (" end"); }
             ;
color:  WHITE { printf ("white"); }
       |  BLACK { printf ("black"); }
       |  RED { printf ("red"); }
       |  BLUE {"printf ("blue"); }
       ;
;
```

בהנחה שהקלט כולל את האסימון RED ולאחריו האסימון WHITE אז הפלט יהיה:  
one red two white end

bison ממש mid rule actions ע"י כך שהוא הופך אותם לפעולות שמופייעות בקצתה הימני של כללי גזירה. לצורך כך הוא מוסיף לדקדוק משתנים חדשים (אחד עבור כל rule action) שנitinן לגזיר מהם רק אפסילון.

לדוגמא, bison יփוך את הכללים בדוגמה שראינו למשהו כזה (הכללים עברו(color הם ללא שינוי)).

```
two_colors: m1 color m2 color { printf (" end"); }

m1: /* empty */ { printf ("one "); }
m2: /* empty */ { printf ("two "); }
```

עתה כל הפעולות הן בקצתה הימני של כללי גזירה. פעולה מתבצעת כאשר עושים reduce לכלל גזירה שלה.

ראינו שהשימוש ב- mid rule action משנה בפועל את הדקדוק. לכן זה עלול לגרום לكونפליקט. ראו דוגמא בסעיף Conflicts due to Mid-Rule במדריך של bison באינטראנט.

ערכים סמנטיים (semantic values)

לSIMNI הדקוק (משתנים ואסימונים) יכולים להיות ערכים סמנטיים (זה המינוח של bison עבור ערכים של תכונות). ערכים אלו נשמרים בזמן הניתוח התחברי במחסנית שנקרה לה - semantic value stack או, בקיצור, ה-state stack. לשתי המתחניות ה-h-state stack תמיד באותו הגודל והן גדולות וקטנות ביחד. כאשר SIMN של הדקוק מוחזק על ה-state stack, הערך הסמנטי שלו מוחזק בכניסה המקבילה ב-value stack.

(הערה : נוח להתייחס ל-state stack כailo היא מכילה גם SIMNI דקוק בנוסף לVALUES של האוטומט הסופי. זאת למראות שבפועל היא מכילה רק VALUES של האוטומט הסופי).

בכל פעם שSIMN דקוקי נדחף על ה-state stack, נדחף הערך הסמנטי שלו ל-value stack.

ישנם שני מקרים כאלו :

-- כאשר עושים shift לאסימון, המשתנה הגלובלי yyval מועתק בראש ה-state stack. המנתח הלקסיקלי (yylex()) אמרור לדואג לכך ש-yyval אכן יכול את הערך הסמנטי של האסימון (במידה ויש לו ערך סמנטי).

-- כאשר עושים reduce למשתנה A (וזוחפים אותו ל-state stack) לפי כלל גזירה מסוים, הפעולה הסמנטית המופיעה בצד ימין של אותו הכלל מתבצעת. פעולה זו (אותה כותב המשתמש) אמורה לחשב את הערך הסמנטי של A (במידה ויש לו כזה) ולדחוף אותו ל-value stack.

בדרך כלל, לצורך חישוב הערך הסמנטי של משתנה מצד שמאל של כלל גזירה, משתמשים בערכים סמנטיים של SIMNI הדקוק המופיעים מצד ימין של אותו הכלל. (במילים אחרות, התכונות ה-h נוצרות (ניבנות)). את הערכים הסמנטיים של SIMNI הדקוק האלו מוצאים ב-value stack. המשתמש יכול לגשת ל-value stack באמצעות הSIMN הבאים :

\$1 הוא הערך הסמנטי של הסימן הדקוקי הראשון מצד ימין של כלל הгазירה,  
\$2 הוא הערך הסמנטי של הסימן השני, \$3 הוא הערך של הסימן השלישי וכן הלאה.

\$\$ הוא הערך הסמנטי של המשתנה שנמצא מצד שמאל של כלל הгазירה.

דוגמא (כל הערכים הסמנטיים כאן הם מティפוס int) :

```
expression : expression '+' term { $$ = $1 + $3; } ;
```

כאן \$1 הוא הערך הסמנטי של expression (זה שצד ימין) ו-\$3 הוא הערך הסמנטי של term. אילו ל-'+' יהיה ערך סמנטי, היינו מתיחסים אליו כ-\$2. \$\$ הוא הערך הסמנטי של ה-expression שצד שמאל.

כאשר המנתח התחבירי יבצע reduce לפי כלל הגזירה הניל, הוא יבצע את הפעולה הרשומה בקצתה הימני, כלומר הוא יחשב את  $\$$  (שזה הערך הסמנטי של ה-expression).

לאחר מכן הוא ימחק (פיעולות pop) מה-state stack את שלושת הסימנים שבסzd ימין של כלל הגזירה (ליתר דיוק הוא ימחק 3 מצבים) ומה-state stack הוא ימחק את הערכים הסמנטיים שלהם. עתה הוא ידוחף על ה-value stack ואת הערך הסמנטי שלו (ה-  $\$$  שזה עתה חושב) הוא ידוחף על ה-.stack.

bison מחליף את כל הסימונים עם  $\$$  להתייחסויות ל-.value stack. לצורך המחשה: בקוד ש-bison מיצר עבור המנתח התחבירי מופיע משתנה yyvsp שמצויב בראש ה-value stack. בדוגמה הניל, בקוד ש-bison מיציר יופיע [0] yyvsp במקום 3 כי הערך של term בראש המחסנית (כאשר עושים reduce לפי הכלל הנוכחי). [2] yyvsp יופיע במקום 1 כי הערך של value stack (זה שבצד ימין של כלל הגזירה) מופיע ב-.value stackשתי כניסה מתחת לראש המחסנית. אילו היינו משתמשים גם ב-2 זה היה מופיע כ- [1] yyvsp.

ראוי להזכיר שהמשתמש אינו-Amור להשתמש ישירות במשתנה yyvsp (שכלל אינו מוזכר במדדיק של Bison) אלא בסימונים  $\$1$ ,  $\$2$  וכן הלאה. כאשר המשתמש אינו כותב במפורש כיצד לחשב את הערך הסמנטי של המשתנה בצד שמאל של כלל גזירה, ברירת המחדל היא  $1 = \$\$$  (כמובן שזה נכון רק במקרה שבו שבסzd ימין של הכלל יש לפחות סימן אחד). לדוגמה, אם כותבים

```
expression : term ;  
זה יכול כתבו  
expression : term { $1 = $2 }  
בדרך כלל האופציה השנייה עדיפה כי היא מבהירה את כוונתו.
```

### התיפוס של הערכים הסמנטיים

התיפוס של הערכים הסמנטיים הוא YYSTYPE. (מכאן שככל כניסה ב-.value stack היא מטיבוס YYSTYPE).

בתור ברירת מחדל, בקוד המוצע ע"י Bison מוגדר:

```
#define YYSTYPE int  
זה טוב כאשר כל הערכים הסמנטיים הם מטיבוס int.  
ניתן לשנות את זה למשל ע"י הוספת השורה  
#define YYSTYPE double  
את השורה יש לרשום בחלק הראשון בקובץ הקלט של Bison (בו כאמור ניתן  
לרשום קטעי קוד בשפת C)
```

בגרסאות חדשות יותר של Bison ניתן להגדיר את התיפוס של הערכים הסמנטיים גם כמו בדוגמה זו:

```
%define api.value.type { double }
```

לעתים קרובות נרצה שלסימני דקדוק שונים יהיו ערכיים סמנטיים מטיפוסים שונים. במקרה כזה YYSTYPE צריך להיות union. את ה-union ניתן להגדיר בעזרת ההכרזה union%. ההכרזה זו נראית כמו הגדרה של union union בשפת C (אבל בלי נקודה פסיק בסוף).

דוגמא : נניח שהערכים הסמנטיים של האסימון NUM ושל המשתנים term, factor הם מטיפוס int בעוד שהערכים הסמנטיים של האסימונים ADDOP, MULOP הם מטיפוס enum op זה האחרון יכול להיות מוגדר למשל כ-

```
%code requires {
    enum op { PLUS, MINUS, MUL, DIV };
}
```

עוד נניח לאסימון ID יש ערך סמנטי מטיפוס \* char (מצבע ל-).

או נגדיר :

```
%union {
    int ival;
    enum op operator;
    char *name;
}
```

עכשו יגדר את YYTYPE בעזרת union C typedef bison

(הערה : את ההגדרה של enum op נדרש להקייב כדי ש- bison ימוך אותה לפניהם ההגדרה של ה- union (שימוש בטיפוס enum op בקבצים שהוא מייצר (כולל ב- include file שהוא מייצר אם מפעלים אותו עם האופציה -d)).

בנוסף לכך נדרש לומר ל-bison, עבור כל סימן דקדוק שיש לו ערך סמנטי, באיזה מהשדות של ה- union יש להשתמש. עבור משתנים עושים זאת באמצעות הכרזות %token. כל ההכרזות אלו (וגם union%) צריכים להופיע בקובץ הקלט של bison בחלק המועד ל- "הכרזות עבור bison".

בדוגמא שלנו :

```
%token <ival> NUM
%token <name> ID
%token <operator> ADDOP MULOP

%type <ival> expression term factor
```

עכשו אם נכתוב לדוגמא :

```
expression : expression ADDOP term
            { if ($2 == PLUS)
                $$ = $1 + $3;
            else
                $$ = $1 - $3; } ;
```

או בקוד שיצר bison יופיע \$2 כי מתייחס לערך הסמנטי של ADDOP union .operator [−1].operator שבעור אסימון זה יש להשתמש בשדה operator של ה-union . בדומה לכך, \$3 יופיע כי \$3 מתייחס לערך הסמנטי של term .operator משנתה זה אמרנו לע- bison שיש להשתמש בשדה .ival [−2].ival .

לעתים אנו מעוניינים שלסימן דקודי יהיה יותר מערך סמנטי אחד. במקרה כזה נוח להגדיר את הערך הסמנטי שלו כ- struct :

אנו מעוניינים של משתנה date יהיה שלושה ערכים סמנטיים : היום בחודש, החודש והשנה.

אפשר להגדיר בחלק הראשון של קובץ הקלט של bison :

```
%code requires {
    struct mydate {
        int day, month, year;
    };
}
```

(ניתן כמובן לשים את ההגדרה של ה- struct ב- include file ולעשות לו .include).

נגיד גם (בחלק המועד להכרזות עבור bison) :

```
%union {
    struct mydate d;
    int ival;
}

%token <ival> NUM
%type <d> date
```

ואז אפשר לכתוב בחלק של כללי הדקדוק משחו כזה :

```
when: date { printf ("day is %d, month is %d,
                     year is %d\n",
                     $1.day, $1.month, $1.year ); } ;

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

### ערכים סמנטיים של mid rule actions

ל- mid rule action יכול להיות ערך סמנטי. הוא מקבל ערך על ידי השמה ל- \$\$. (אפשר לחשב על זה כעל הערך הסמנטי של המשתנה החדש ש- bison מייצר לצורך הטיפול ב- .mid rule action).

דוגמה :

```
list:NUM NUM NUM { $$ = $1 + $3; } NUM NUM { $$ = $4 * $6; } ;
```

כאו הערך הסמנטי של ה- mid rule action הוא הסכום של המספר הראשון והמספר השלישי. שימושו לבשה- mid rule action אינו יכול להשתמש בערכיים סמנטיים של הסימנים שלימינו כי אלו טרם חשובו.

פעולות שMOVEDיעות לימין mid rule action נתון יכולות להתייחס לערך הסמנטי שלו בעזרת הסימון הרגיל  $\$$ . נלקחים בחשבון כאשר סופרים רכיבים לצורך סימון זה.

בדוגמא שלנו,  $\$4$  הוא הערך הסמנטי של ה- mid rule action  $\$5$ , והוא הערך הסמנטי של ה- NUM הראשון לימינו וכן הלאה.

הערך הסמנטי של list יהיה המכפלה של המספר האחרון בסכום של המספר הראשון עם המספר השלישי.

שימוש לבשה-union  $\$\$$  בפעולה שנמצאת בקצת הימני מתייחס לערך הסמנטי של המשתנה בצד שמאל של הכלל בעודו סימון כאשר הוא מופיע ב- mid rule action לערך הסמנטי של ה- .mid rule action.

כאשר משתמשים ב- union %, יש לכתוב במפורש באיזה שדה של ה- union יש להשתמש כאשר ניגשים לערך סמנטי של mid rule action .  
זה נעשה בעזרת סימון <union-member-name> כמו בדוגמא הבאה.

נניח שהכרזנו על union שאחד משדותיו נקרא `ival`.  
או אפשר לכתוב את הדוגמא הקודמת כך :

```
list: NUM NUM NUM { $<ival>$ = $1 + $3 } NUM NUM
                    { $$ = $<ival>4 ; }
כמה שיש גם צורך בהכרזות כמו
%token <ival> NUM
%type <ival> list
```

#### ( ) yyparse (), yyerror ()

כדי לבצע את הניתוח התחבירי צריך לקרוא לפונקציה () yyparse אותה מייצרת bison. היא מחזירה 0 כאשר הניתוח התחבירי מסתנאים בהצלחה. אחרת היא מחזירת 1.

() yyparse קוראת לפונקציה () yylex כל פעם שהיא נזקקת לאסימון הבא בקלט. לפני הקריאה ל- () yyparse יש לדאוג לכך ש- () yylex תקרא את הקלט שלא מהמקום המתאים. אם מייצרים את () yylex בעזרת flex אז צריך לכוון את המשתנה הגלובלי yyin (שבאמצעותו () yylex תקרא את הקלט שלא) לקובץ המתאים.

כאשר () מגליה שגיאה תחבירית היא קוראת לפונקציה () yyerror עם המחרוזת "parse error" "parse error" כארגומנט. על המשתמש להגדיר בעצמו את yyerror(). הנה הגדלה לדוגמא :

```
void yyerror (char *s)
{
```

```

        fprintf (stderr, "%s\n", s);
}

```

כדי לקבל הודעה שגיאה מפורטת יותר כאשר קוראים ל- `yyerror`, כדי להוסיף את ההכרזה `%error-verbose` (בחילק הראשון של קובט הקלט ל- `#define YYERROR_VERBOSE bison`). בגרסאות ישנות יותר יש לרשום `bison`

### פתרונות קונפליקטים

`bison` מוציא הודעות אזהרה על כל הקונפליקטים שהוא מגלה.

אם יש קונפליקטים שאתם יודעים ש- `bison` פתר בצורה נכוןה (ראו בהמשך) ניתן לגרום לו להימנע מההודעות אלו בעזרת ההכרזה `n %expect` כאשר `n` מסמן שלם. במקרה ש- `bison` מוצא בדיקת `n` קונפליקטים, הוא לא יוציא הודעות עליהם.

ניתן לשנות באופן שבו `bison` פותר קונפליקטים מסווג `shift/reduce` באופן הבא.

אפשר להגדיר עדיפות אסוציאטיבית לאופרטורים בעזרת ההכרזות `%right` ו- `%left`.

לכל טרמינל שמכרז עם `%left` יש אסוציאטיביות שמאלית. לטרמינל שמכרז עם `%right` יש אסוציאטיביות ימנית. הסדר שבו מופיעות ההכרזות קובע את העדיפות של הטרמינלים: אלא שמכרזים בהכרזה הראשונה מסווג `%left` או `%right` הם בעלי העדיפות הנמוכה ביותר. הטרמינלים שמוספיים בהכרזה הבאה מסווג זה הם בעלי עדיפות מעט יותר גבוהה וכן הלאה.

לכל גזירה מסוימת גם יש עדיפות: זוהי העדיפות של הטרמינל האחרון שמוספי עבוגם שלהם (אם הטרמינל האחרון אין עדיפות או שאין בו טרמינל אז גם לכל הגזירה אין עדיפות).

כך פותר `bison` קונפליקטים מסווג `shift/reduce`: אם `-l lookahead` יש עדיפות גבוהה מאשר לכל הגזירה הרלוונטי אז עושים `shift`. אם ההיפך נכון אז עושים `reduce`. אם `-l lookahead` ולכל הגזירה יש אותה עדיפות אז האסוציאטיביות קבועות: אם זו אסוציאטיביות שמאלית אז עושים `shift`. אם זו אסוציאטיביות ימנית אז עושים `reduce`.

אם ל- `lookahead` או לכל הגזירה אין עדיפות אז עושים `shift` בתור ברירת מחדל.

לדוגמה:

```

%left '+' '-'
%left '*' '/'

```

```

%%
exp: exp '+' exp
| exp '-' exp
| exp '*' exp
| exp '/' exp
| NUM
;

```

נניח לדוגמה שהקלט הוא  $5 * 4 + 3$  (המספרים הם אסימוניים מסוג NUM). אחרי שהמנת החבברי יראה בקלט את  $3 + 4$  וה lookahead יהיה '\*' תהיה לו ברירה לעשות shift או reduce לפי הכלל  $\text{exp} : \text{exp} '+' \text{exp}$ . בגלל שבמקרה זה ל lookahead יש עדיפות גבוהה יותר מאשר לכל הגזירה (שעדיפותיו היא כמו זו של '+') המנתה יבצע shift. פרוש הדבר שעż הגזירה שיבנה עבור הקלט מתאים לו  $(5 * 4) + 3$ . אילו היה מtbody reduce באותו זמן אז עż הגזירה היה מתאים לו  $5 * (3 + 4)$ .

לעומת זאת אם בקלט היה  $5 * 4 + 3$ , אז אחרי שהמנת החבברי יראה את  $4 * 3$  וה lookahead יהיה '+', הוא יבחר לעשות reduce לפי  $\text{exp} : \text{exp} * \text{exp}$  (ולא shift) כי לכל הגזירה יש עדיפות כמו לו  $'*$  וזו עדיפות גבוהה יותר מאשר לש lookahead '+'. כך הקלט יפרק כ- $5 + 4 * (3 + 5)$  ולא כ-  $(4 + 5) * 3$ .

**דוגמא נוספת:** נניח שהקלט יש  $5 - 4 - 3$ . אחרי שהמנת החבברי יראה בקלט את  $3 - 4$  וה lookahead יהיה '-'. תהיה לו ברירה לעשות shift או reduce לפי הכלל  $\text{exp} : \text{exp} '-' \text{exp}$ . בgalל שבמקרה זה ל lookahead יש עדיפות זהה לו של לכל הגזירה האסוציאטיביות היא שתקבע כיצד ניתן לפתר הקונפליקט. לאחר שהטרמינל '-' הוכרז כבעל אסוציאטיביות שמלואית יבוצע reduce. פרוש הדבר שעż הגזירה שיבנה עבור הקלט מתאים לו  $5 - (4 - 3)$ . אילו היה לטרמינל '-' אסוציאטיביות ימנית אז היה מtbody shift ועż הגזירה היה מתאים לו  $(5 - 4) - 3$ .

**הערות נוספות:** טרמינל שמוכרז עם %nonassoc איןנו אסוציאטיבי זאת אומרת שם הוא מופיע פעמיים "ברצף" אז זו שגיאה תחבירית.

ניתן לציין באופן מפורש מה העדיפויות של כלל גזירה ע"י הוספה %prec terminal-symbol בסוף הגוף של כלל הגזירה. במקרה זה העדיפויות של כלל הגזירה תהיה כמו זו של הטרמינל.

bison פותר קונפליקטים מסוג reduce/reduce ע"י כך שהוא בוחר לעשות reduce לפי כלל הגזירה המקוריים קודם. בכך כל רצוי לדאוג לכך שהדקוק לא יכלול קונפליקטים מסוג זה.

## תאום עם () yylex

לכל סוג של אסימון יש קוד (מספר שלם) המיציג אותו. () yylex צריכה להחזיר את הקוד עבור סוג האסימון שזה עתה מצאה. כאשר היא מגיעה לסוף הקלט היא צריכה להחזיר אפס. (אם () yylex מיוצרת בעורת flex היא תחזיר כנדרש אפס כאשר תתקל בסוף קובץ הקלט).

כאשר לאסימון יש שם אז בקוד המיציר על ידי bison, השם של האסימון הוא מקרו (שפה C) שערכו הוא הקוד עבור אותו סוג של אסימון.

%token NUM ADDOP MULOP

לדוגמה הכרזה :

תגרום ל- bison ליצר משהו כזה :

```
#define NUM 258
#define ADDOP 259
#define MULOP 260
```

בגרסאות חדשות יותר, bison מיציג את האסימונים עם :enumeration typeenum yytokentype {  
NUM = 258,  
ADDOP = 259,  
MULOP = 260  
};

() yylex יכולה להשתמש בהגדרות אלו כדי להחזיר את סוג האסימונים שהיא מוצאת. כמו בקוד של () yylex יכול להופיע משהו כמו ; return NUM;

כאשר מתיחסים לאסימון בכללי הדקוק באמצעות character literal (לדוגמה +'') אז הקוד עבור התו (קוד ה- ascii שלו) משמש גם עבור סוג האסימון. לכן return '+'; () yylex יכולה לעשות משהו כמו ; כדי להחזיר אסימון כמו +''

() yylex צריכה לשים את הערך הסמנטי של האסימון שהוא מחזירה בתוך המשתנה הגלובלי yyval. הטיפוס של משתנה זה הוא YYSTYPE (שהיה גם הטיפוס של כל כניסה ב- value stack שהזוכר לעיל).  
כאשר YYSTYPE הוא טיפוס פשוט כמו למשל int (שו ברירת המחדל), () yylex יכולה לעשות משהו כזה :

```
yyval = <the number>;
return NUM;
```

כאשר משתמשים בערכים סמנטיים מטיפוסים שונים או yyval הוא union (כפי שהוגדר בעורת הכרזות union%). במקרה כזה () yylex צריכה לשמור את הערך הסמנטי בשדה המתאים של ה- union.  
לדוגמה, אם הכרנו :

```
%union {
    int val;
    char *name;
}
%token <val> NUM
%token <name> ID
```

از הקוד ב-) yylex יכול להראות כך :

```
...
yylval.val = <the number>;
return NUM;
...
yylval.name = <the name>;
return ID;
...
```

מן האמור כאן, ברור ש-) yylex צריכה להכיר את ההגדרות של האסימונים ואת המשתנה הגלובלי .yylval. יש שני דרכי לדואג לכך:

הראשונה היא להגדיר את () ex yylex בחלק האחרון של קובץ הקלט של bison (החלק המיועד ל- "קוד נוסף בשפת C") או לחלופין להוסיף שם שורה שעשה include לקובץ המכיל את ההגדרה של () yylex – למשל .yylex">#include lex.yy.c במקורה משתמשים ב- flex כדי ליצור את () ex.

הדרך השנייה היא להפעיל את Bison עם אופציה -d. זה יגרום ל- Bison ליצרת (בנוסף לקובץ עם ההגדרה של ה- parser) גם קובץ include שיכלול את הגדרות האסימונים, את ההכרזה של yylval ואת ההגדרה של YYSTYPE.

אם משתמשים ב- flex אז יש להוסיף לקובץ הקלט של flex (בחלק הראשון של הקובץ בחלק שבו רושמים קוד בשפת C של המשתמש) שורה שעה שעשוה include לקובץ ה- Bison ייצור.

איך לדבג את המנתה התחביברי  
יש אפשרות שהמנתה התחביברי ידפיס תאור של פעולותיו. זה כולל דיווח על כל פעולות ה- shift וה- reduce שהוא עושה, מה תוכן מחסנית ה- state stack בכל שלב ואיזה אסימון הוחזר לו בכל פעם שקרה לך () yylex.

לצורך כך יש להגדיר את המקרו YYDEBUG. יש מספר דרכים לעשות את זה.

```
#define YYDEBUG 1
```

בחלק המיועד ל"קוד בשפת C" בחלק הראשון של קובץ הקלט של Bison. יש גם להוסיף את השורה #include <stdio.h> לאותו המקום.

בנוסף להגדרת YYDEBUG כנ"ל, כדי לגרום למנתה התחביברי לדוח על פעולותיו יש להכניס ערך שונה מאפס למשתנה הגלובלי yydebug. אפשר לעשות את זה בתוך () main.

אם מפעילים את Bison עם האופציה -d אז הוא מייצר קובץ שכולל בין היתר תאור של כל מוצבי האוטומט (שבו משתמש המנתה התחביברי) כולל איזה פעולה (shift, lookahead) יעשה ה- parser בכל מצב עבור כל סוג של פוליה. בעזרת התאור הזה ניתן להבין מדוע Bison מודיע שמצא קונפליקטים (ולתקן את הדקדוק אם צריך).

עדכן לאחרונה ב- 15 באפריל 2015

ניטוח תחבירי – גישת top down ודקודוי (1)

ניתוח תחבירי

מנתח תחבירי (parser או syntax analyser) מקבל קלט סיירה של איסימונים (טרמינליים) אותם מעביר המנתח הלקסיקלי. המנתח התחבירי מוצא גזירה של הקלט שלו לפי כללי הדקדוק. אם לא ניתן לגזר את הקלט לפי כלליים אלו, הקלט נחשב לשגוי ועל המנתח לתהבירו להודיע על כך. על המנתח לנסות להתאושש משגיאות תחביריות כדי לגלות שגיאות נוספות אם ישן.

הפלט של המנתח התכוברי עשוי להיות עצ' גזירה של הקלט שלו או syntax tree. בדוגמה שנראה בהמשך, הפלט של המנתח יהיה תאור של גזירה שמאלית ביותר של הקלט.

ישנן מספר שיטות לביצוע ניתוח תחבירי. נראה כאן שיטה שעוברת על עץ הגזירה מלמעלה למטה (top down) : מתחילה למלعلاה (בשורש) ומתקדמים למטה לכיוון העלים. השיטה נקראת predictive parsing.

הסתייעות בסיכון הקלט הבא (ה-lookahead).  
 גזירה. דקוק שהוא (1) LL תמיד יאפשר לבחור את הכלל המתאים תוק  
 כליל יש מספר אפשרויות כי למשתנה בו מסומנת הצומת יכולים להיות מספר כללי  
 נתנו הבעיה היא לבחור כלל גזירה כאשר יוצרים את הילדים של צומת נתונה. בדרך  
 בשיטה זו נקרא דקוק (1) LL. כאשר בונים (מלמעלה למטה) עץ גזירה עבור קלט  
 שיטה זו מתאימה לכל הדקדוקים. דקוק המאפשר ניתוח תחבירי

**נשתמש בדקדוק הבא לצורך הדגמה:**

- (1)  $\text{stmt} \rightarrow \underline{\text{if}} \text{ boolexp } \underline{\text{then}} \text{ stmt } \underline{\text{else}} \text{ stmt } \underline{\text{fi}}$
  - (2)  $\text{stmt} \rightarrow \underline{\text{while}} \text{ boolexp } \underline{\text{do}} \text{ stmt } \underline{\text{od}}$
  - (3)  $\text{stmt} \rightarrow \underline{\text{begin}} \text{ stmtlist } \underline{\text{end}}$
  - (4)  $\text{stmt} \rightarrow \text{assignstmt}$
  - (5)  $\text{stmtlist} \rightarrow \text{stmt } \text{stmtlist}$
  - (6)  $\text{stmtlist} \rightarrow \varepsilon$
  - (7)  $\text{boolexp} \rightarrow \underline{\text{id}} \text{ R}$
  - (8)  $\text{R} \rightarrow \underline{\text{relop}} \text{ } \underline{\text{num}}$
  - (9)  $\text{R} \rightarrow \varepsilon$
  - (10)  $\text{assignstmt} \rightarrow \text{id} = \text{num} ;$

## הערות:

מהמשתנה stmtlist ניתן לגוזר כל סידרה סופית של אפס או יותר stmt – ים. ולכן המשתנה stmtlist מייצג סדרות של משפטים). ניתן היה להחילף את כל הגזירה החמישית בכלל : stmtlist → stmtlist stmt אבל אז הדקוק לא יהיה מתאים לשיטת הניתוח התכבירי שבה אנו עוסקים. (במילים אחרות: הדקוק לא היה (1) LL).

הערה נוספת : טبعי יותר היה לתאר ביטויים בוליאניים בעזרת שני הכללים הבאים  
(במקום כלליים 9, 7, 8) :

boolexp → id  
boolexp → id relop num

השימוש בשני הכללים האלה היה מונע שימוש ב- predictive parsing. (הדקוק לא היה (LL(1)).

בנית טבלה עבור predictive parsing לא רקורסיבי  
קלט : דקוק 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 להכיל כללי גזירה (רצוי לא יותר מאחד – ראו בהמשך) או ציון של שגיאה (error).

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

האלגוריתם :

1. עבור כל כללי גזירה (של הדקוק G)  $\alpha \rightarrow A$  בצע את צעדים 2 ו- 3

2. עבור כל טרמינל a שנמצא ב- FIRST( $\alpha$ ), הוסף את  $\alpha \rightarrow A$  לכניסה M [A, a]

3. אם a נמצא ב- FIRST( $\alpha$ ) אז עבור כל טרמינל b שנמצא ב- FOLLOW(A). אם a נמצא ב- FIRST( $\alpha$ ) הוסף את  $\alpha \rightarrow A$  לכניסה [A, b]. אם a נמצא ב- FOLLOW( $\alpha$ ), הוסף את  $\alpha \rightarrow A$  לכניסה [M, \$].

4. כל כניסה של M שנותרה ריקה נחשבת לכניסה error.

5. אם יש כניסה אחת (או יותר) ב- M שמקילה יותר מכל גזירה אחד אז האלגוריתם נכשל אחרת הוא מסתיים בהצלחה.

הערה : זה לא טוב כאשר כניסה בטבלה מכילה יותר מכל גזירה אחד (זה נקרא كونפליקט) כי כאשר המנתח נזקק לכניסה זו, אין הוא יודע באיזה מהכללים הרשומים בה להשתמש.

דוגמא

הרצת האלגוריתם לבנית טבלה על הדקוק שהובא לעיל תנו את הטבלה הבאה.

המספרים בטבלה מציננים כללי גזירה. הכניסות הריקות מציננות error.

	if	the n	else	fi	whi le	do	od	begi n	end	id	=	num	;	relo p	\$
stmt	1				2			3		4					
stmtlist	5				5			5	6	5					
boolexp										7					
R		9			9								8		
assign stmt											10				

לצורך בניית הטבלה יש לחשב ראשית את FIRST ו-FOLLOW של מושתני הדקדוק. בדקדוק שהובא לעילו מתקיים :

FIRST (stmt) = { if, while, begin, id }  
 FIRST (stmtlist) = {  $\epsilon$ , if, while, begin, id }  
 FIRST (boolexp) = { id }  
 FIRST (assignstmt) = { id }  
 FIRST (R) = {  $\epsilon$ , relop }

בדקדוק שהובא לעילו מתקיים :

FOLLOW (stmt) = { \$, else, fi, od, end, if, while, begin, id }  
 FOLLOW (stmtlist) = { end }  
 FOLLOW (boolexp) = { then, do }  
 FOLLOW (assignstmt) = { \$, else, fi, od, end, if, while, begin, id }  
 FOLLOW (R) = { then, do }

עכשו ניתן לבנות את הטבלה.

איך (לדוגמא) מחליטים היכן בטבלה לרשום את כלל 5 ?

(5) stmtlist  $\rightarrow$  stmt stmtlist

ברור שיש לרשום את הכלל בשורה של stmtlist (כי הוא המשתנה באגף שמאל של הכלל) אבל באילו עמודות ?

מחשבים את FIRST של אגף ימין של הכלל ומתקבלים :

FIRST (stmt stmtlist) = { if, while, begin, id }  
 . if, while, begin, id בעמודות עבר id מכיוון שהוא הראשון שמאפשר רישום כלל 5.

איך (לדוגמא) מחליטים היכן בטבלה לרשום את כלל מס' 9 ?

(9) R  $\rightarrow$   $\epsilon$

מחשבים את FIRST של אגף ימין של הכלל :

FIRST ( $\epsilon$ ) = {  $\epsilon$  }  
 הכלל שאפסילון נמצא ב-FIRST של אגף ימין, יש לרשום את כלל 9 בשורה של R בעמודות עבר then ו- do כי FOLLOW (R) = { then, do }

הגדרה : דקדוק שאין בטבלה עבורי כניסה הכלולית יותר מכל גזירה אחד נקרא דקדוק (1) LL. L אחד מייצג את העובדה שכאשר עושים ניתוח תחבירי, קריאת הקלט היא משמאלי לירני (left to right). L שני מייצג את העובדה שהניתוח התחבירי מתאים ל-leftmost derivation. ה- "1" אומר שה-lookahead הוא בגודל 1 כלומר המנתח התחבירי עושה שימוש בטרמינל הבא בקלט כדי להחליט מה לעשות. (באופן כללי, דקדוק (k) LL הוא כזה שנitinן לעשות ניתוח תחבירי לפיו תוך שימוש ב-k הטרמינלים הבאים בקלט).

ניתן להראות שדקדוק הוא (1) LL אם ורק אם עבור כל שני כללי גזירה שונים  $\beta | \alpha \rightarrow A$  מתקיימים התנאים הבאים :

1. לא ניתן לגוזר מ-  $\alpha$  ומ-  $\beta$  מחרוזות שמתחללות באותו טרמינל.
2. לא ניתן לגוזר את  $\epsilon$  גם מ-  $\alpha$  וגם מ-  $\beta$ . (יתכן שנitinן מאחד מהם).

3. אם ניתן לגזר את  $\epsilon$  מ- $\beta$  אז לא ניתן לגזר מ- $\alpha$  מחרוזת שמתחלילה בטרמינל  
שנמצא ב- FOLLOW(A).

#### דוגמאות

הקדוק הבא אינו LL(1) כי הוא מפר את התנאי הראשון הנ"ל:

- (1)  $S \rightarrow ab$
- (2)  $S \rightarrow ac$

בטבלה עבור דקדוק זה, שני כללי הגזירה יופיעו בשורה של S ובעמודה של a

הקדוק הבא אינו LL(1) כי הוא מפר את התנאי השני הנ"ל:

- (1)  $S \rightarrow Ac$
- (2)  $A \rightarrow B$
- (3)  $A \rightarrow \epsilon$
- (4)  $A \rightarrow a$
- (5)  $B \rightarrow \epsilon$
- (6)  $B \rightarrow b$

בטבלה עבור דקדוק זה, כללי הגזירה 2 ו- 3 שניהם יופיעו בשורה של A ובעמודה של c – שימושם לב שמות קיימים  $\{c\} = \text{FOLLOW}(A)$

הקדוק הבא אינו LL(1) כי הוא מפר את התנאי השלישי הנ"ל:

- (1)  $S \rightarrow Aa$
- (2)  $A \rightarrow \epsilon$
- (3)  $A \rightarrow a$

בטבלה עבור דקדוק זה, כללי הגזירה 2 ו- 3 שניהם יופיעו בשורה של A ובעמודה של a – שימושם לב שמות קיימים  $\{a\} = \text{FOLLOW}(A)$

#### הערות:

1. כל דקדוק רקורסיבי שמאליל אינו LL(1).
2. כל דקדוק רב משמעי אינו LL(1).

#### מה עושים כאשר הדקדוק אינו LL?

1. לעיתים ניתן למצוא דקדוק שקול שהוא.cn (1). לצורך כך ניתן להפעיל על הדקדוק המקורי טרנספורמציות (ביטול רקורסיה שמאלית, left factoring).
2. לעיתים ניתן לשנות את טבלת הנитוח תחבירי באופן ידני.
3. לעיתים ניתן לבצע ניתוח תחבירי כאשר מגדים את ה-lookahead.
4. משתמשים בשיטה אחרת לצורך ניתוח תחבירי.

#### אלגוריתם ל-predictive parsing לא רקורסיבי

קלט: מחרוזת של טרמינלים w וטבלת ניתוח תחבירי M עבור דקדוק G.

פלט: אם  $w \in L$ , גזירה שמאלית ביותר של  $w$ , אחרת הודעת שגיאה. הגזירה השמאלית ביותר של  $w$  תתואר בפלט ע"י כך שהאלגוריתם יכתוב לפט את כללי הגזירה בהם משתמשים בגזירה זו.

איתחול: במחסנית יש שני סימנים:  $\$$  כאשר  $S$  (המשתנה ההתחלתי) בראש המחסנית. בקלט נמצא  $w$  (הסימן  $\$$  מסמן את סוף הקלט). מתחלים מצביע לקלט כך שיצביע לסימן הראשון ב- $w$

חוורים על הצעדים הבאים:  
יהי  $X$  הסימן שבראש המחסנית ו- $a$  סימן הקלט הנוכחי.

אם  $X$  הוא טרמינל אז

אם  $X = a$  אז

1. הוצאה את  $X$  מהמחסנית (pop)

2. קדם את המצביע לקלט לטרמינל הבא.

אחרת: כתוב הודעת שגיאה ועוצר.

אם  $X$  הוא  $\$$  אז

אם סימן הקלט הנוכחי הוא  $\$$  (כלומר הגענו לסוף הקלט) אז עוצר ודווח על הצלחה.

אחרת כתוב הודעת שגיאה ועוצר.

אם  $X$  הוא משתנה אז

אם  $[X, a] = X \rightarrow Y_1 Y_2 \dots Y_k$  אז

1. הוצאה את  $X$  מהמחסנית (pop)

2. דחוף למחסנית בסדר הפיך את כל הסימנים שמופיעים בצד ימין של

כל הגזירה כלומר דחוף את  $Y_k$ , ולאחריו את  $Y_{k-1}$  וכן להאה עד שלבסוף דחוף את  $Y_1$  (כך שבסיומו של דבר  $Y_1$  יהיה בראש המחסנית).

3. כתוב לפט שנעשה שימוש בכל הגזירה  $Y_1 Y_2 \dots Y_k$

אחרת (כלומר  $[X, a] = \text{error}$ ) כתוב הודעת שגיאה ועוצר.

#### דוגמא להרצת האלגוריתם

זה הקלט. (למונן הנוחות הוספו כאן אינדקסים למופעים השונים של id ושל num).

while id<sub>1</sub> do if id<sub>2</sub> relop num<sub>1</sub> then id<sub>3</sub> = num<sub>2</sub>; else id<sub>4</sub> = num<sub>3</sub>; fi od \$

האלגוריתם משתמש בטבלה שמופיעה לעלה.

תוכן המחסנית lookahead

\$ stmt	while	הדף כלל גזירה 2
\$ od stmt do boolexp while	while	
\$ od stmt do boolexp	id <sub>1</sub>	הדף כלל גזירה 7
\$ od stmt do R id	id <sub>1</sub>	
\$ od stmt do R	do	
\$ od stmt do	do	הדף כלל גזירה 9
\$ od stmt	if	

\$ od fi stmt else stmt then boolexp if	if	הדפס כלל גזירה 1
\$ od fi stmt else stmt then boolexp	id <sub>2</sub>	
\$ od fi stmt else stmt then R id	id <sub>2</sub>	הדפס כלל גזירה 7
\$ od fi stmt else stmt then R	relop	
\$ od fi stmt else stmt then num relop	relop	הדפס כלל גזירה 8
\$ od fi stmt else stmt then num	num <sub>1</sub>	
\$ od fi stmt else stmt then	then	
\$ od fi stmt else stmt	id <sub>3</sub>	הדפס כלל גזירה 4
\$ od fi stmt else assignstmt	id <sub>3</sub>	
\$ od fi stmt else ; num = id	id <sub>3</sub>	הדפס כלל גזירה 10
\$ od fi stmt else ; num =	=	
\$ od fi stmt else ; num	num <sub>2</sub>	
\$ od fi stmt else ;	;	
\$ od fi stmt else	else	
\$ od fi stmt	id <sub>4</sub>	הדפס כלל גזירה 4
\$ od fi assignstmt	id <sub>4</sub>	
\$ od fi ; num = id	id <sub>4</sub>	הדפס כלל גזירה 10
\$ od fi ; num =	=	
\$ od fi ; num	num <sub>3</sub>	
\$ od fi ;	;	
\$ od fi	fi	
\$ od	od	
\$	\$	הכרז על הצלחה

#### הערות:

1. ניתן להסתכל על הסימנים במחסנית כמייצגים את מה שהוא מצפים למצוא

בשימוש הקלט. לדוגמה אם ברגע מסוים המחסנית מכילה :

\$ od stmt do boolexp

(כאשר boolexp בראש המחסנית), פרוש הדבר שמצפים למצוא בקלט ביטויי בוליאני (כלומר מחרוזת של טרמינלים הניתנת לגזירה מ- boolexp), לאחריו את הטרמינל od, ולאחריו משפט (כלומר מחרוזת של טרמינלים הניתנת לגזירה מ- stmt), ולאחריו את הטרמינל do, ולבסוף מצפים למצוא את סוף הקלט.

2. קונפיגורציה של המנתח התחבירי ברגע נתון כוללת את :

- הקלט ואת המיקום הנוכחי בקלט
- תוכן המחסנית

כאשר הקלט אינו שגוי (כלומר שיך לשפה (G) L), כל קונפיגורציה כזו מתאימה לתבנית פסוקית אחת בגזירה השמאלית ביותר של הקלט : ההשিירוש של סימני הקלט שכבר נקראו עם סימני הדקדוק שבמחסנית (מראש המחסנית ועד לתחרתייה) נותן את התבנית הפסוקית המתאימה. מנתח עבר מkonfiguraciah אחת לבאה אחרת בשתי דרכים :

- הוא עושה סוק לטרמינל בראש המחסנית ומקדם את המצביע למיקום הנוכחי בקלט. במקרה זה הkonfiguraciyah החדש מתאימה לאותה תבנית פסוקית כמו קודמתה.
- הוא מחליף את המשטנה בראש המחסנית מצד ימין של כלל גזירה עבורו. במקרה זה הkonfiguraciyah החדש תתאים לתבנית הפסוקית הבאה בגזירה השמאלית ביותר של הקלט. המשטנה שהיה בראש המחסנית היה המשטנה השמאלי ביותר בתבנית הפסוקית כי משמאלו היו רק טרמינלים (אלו שכבר נקראו).

### predictive parsing רקורסיבי

#### מאפייני שיטה זו :

1. זה סוג של recursive descent.
2. את הרוטינות עבור המשטנים של הדקדוק ניתן לכתוב באופן ידני.
3. זו גירסה רקורסיבית של שיטת ה- predictive parsing הלא רקורסיבי. גם הגירסה הזו תעבור עבור דקדוקים שהם (1) LL. דקדוקים אלו מאפשרים לבחור באיזה כלל גזירה להשתמש תוך שימוש ב- lookahead בגודל אחד.

#### העקרון

לכל משטנה A של הדקדוק כתבים רוטינה נפרדת. הרוטינה מחליטה עבור כל lookahead אפשרי, באיזה כלל גזירה של A להשתמש. החלטה נעשית כמו בשיטת הקודמת שראינו (predictive parsing לא רקורסיבי).

(הערה : אם יש למשטנה A כלל גזירה שמצד ימין שלו ניתן לגזור אפסילון או ניתן להשתמש בכלל הזה בכל מקרה שאינו כלל אחר ישים. השיטה הבסיסית אומרת שצורך לוודא שה- lookahead (A) FOLLOW ורק אז להשתמש בכלל הזה (ואחרת להריצו על שגיאה) אבל אין בכך צורך כי השגיאה ממילא תתגלתה בהקדם עוד לפני שהקלט יקודם מעבר ל- lookahead הנוכחי).

לאחר בחירת כלל גזירה תטפל הרוטינה בכל סימני הדקדוק שמופיעים מצד ימין שלו לפי הסדר ובאופן הבא :

- כאשר הסימן הוא טרמינל הוא תבדוק שכאן זה הטרמינל הבא בקלט. אם כן – היא תקדם את המצביע לקלט לטרמינל הבא. אם לא – היא תוציא הودעת שגיאה. הרוטינה () match מבצעת מטלה זו בדוגמה שנראה.
- כאשר הסימן הוא משתנה תבצע קריאה לרוטינה המתאימה למשטנה זה.
- אםצד ימין של כלל גזירה שנבחר הוא א' אז הרוטינה עבור A תחזיר מיד.

אם אף כלל גזירה אינם מתאימים אז הרוטינה תוציא הודעת שגיאה.

#### הערות

- האפקט הכלול של קריאה לרוטינה עבור המשטנה A הוא ש"אוכלים" מהקלט מחרוזת של טרמינלים שניתנת לגזירה מ- A. זאת בתנאי שכאן קיימת מחרוזת כזוتا במקום שבו ממוקם הקלט. אחרת – מכרים על שגיאה.

- בהתחלה קוראים לרוטינה עבור המשטנה ההתחלתי של הדקדוק. אם הרוטינה הזו מצליחה והגענו לסוף הקלט (אחרי שchorה) אז הניתוח התחבירי הצלich אחרת – הוא נכשל.
- אם כל רוטינה כתוב לפלט באיזה כל גזירה היא בחרה קיבל בפלט את כללי הגזירה שיש להפעיל בגזירה שמאלית ביותר של הקלט.

### דוגמא

```

TOKEN lookahead;

void match (TOKEN token)
{
    if (token == lookahead)
        lookahead = nextToken ();
    else
        error ();
}

int main () {
    lookahead = firstToken ();
    stmt ();
    if (lookahead == $) /* end of input */
        return(SUCCESS);
    else return (FAILURE);
}

void stmt () {
    switch (lookahead) {
    case IF :
        match (IF); boolexp (); match (THEN);
        stmt (); match (ELSE); stmt (); match (FI);
        break;

    case WHILE :
        match (WHILE); boolexp (); match (DO);
        stmt (); match (OD);
        break;
    case BEGIN :
        match (BEGIN); stmtlist (); match (END);
        break;

    case ID :
        assignstmt ();
        break;
    }
}

```

```

        default:
            error ();
        }
    }

void stmtlist () {
    switch (lookahead) {
    case IF : case WHILE : case BEGIN : case ID :
        stmt (); stmtlist ();
        break;
    default :
        return; /* stmtlist → ε */
    }
}

void boolexp () {
    if (lookahead == ID) {
        match (ID); R ();
    }
    else error ();
}

void R () {
    if (lookahead == RELOP) {
        match (RELOP); match (NUM);
    }
    /* else R → ε */
}

void assignstmt () {
    if (lookahead == ID) {
        match (ID); match ( = ); match (NUM);
        match ( ; );
    }
    else error ();
}

```

## אלגוריתם ל- top down parsing (עם מחסנית ושימוש בטבלת (1)LL)

### סיכום

נתאר קודם פועלות match: משווים בין האסימון (טרמינל) בראש המחסנית לבין lookahead. אם הם שווים אז מוחקם את האסימון מהמחסנית ומתקדמים בקלט (כלומר קוראים למנתח הלקסיקלי (ה- lexer) והאסימון שהוא מחזיר הופך להיות ה- lookahead). אם האסימון בראש המחסנית אינו זהה ל- lookahead אז התגלתה syntax error. כי בכל שלב ה- parser עשה את הצעד האפשרי היחיד ובכל זאת עץ הגזירה שבנה אינו מתאים לקלט. המסקנה היא שלא ניתן לבנות עץ גזירה לקלט ולכון הקלט שגוי --- הוא אינו מתאים לדקדוק).

### האלגוריתם להרצאת ה- parser על קלט נתון:

דוחפים \$ למחסנית (S, המשטנה ההתחלתי, יהיה בראש המחסנית בהתחלה. \$ יהיה בקרקעית של המחסנית לכל אורך הריצה של ה- parser).

### חווזרים בלולאה :

אם בראש המחסנית יש אסימון או עושים match כפי שתואר לעילו.  
אם בראש המחסנית יש משתנה או גזירים אותו. כדי לדעת באיזה כל גזירה להשתמש ניגשים לטבלת ה- (1)LL (ニゲシム לשורה של המשתנה אותו יש לגזור ולעמודה המתאימה ל- lookahead). את המשתנה שבראש המחסנית מחליפים בגוף (צד ימין) של הכלל שנבחר. (אם הכניסה אליה ניגשים בטבלה היא ריקה אז זה syntax error -- אין כלל גזירה שמאפשר להמשיך ולבנות את עץ הגזירה ולכון המסקנה היא שלא קיים עץ גזירה עבור הקלט).

אם בראש המחסנית יש \$ (אם כך קורה אז זה כל מה שנשאר במחסנית) אז אם ה- lookahead הוא \$ (זה אומר שהגענו לסוף הקלט -- אין באמת \$ בקלט) אז ה- parsing הסתיים בהצלחה (זה נקרא שה- parser עושה "accept" -- הוא "מקבל" את הקלט). אם לא הגיעו לסוף הקלט אז זה syntax error.

### הערות

בכל שלב המחרוזת על המחסנית משקפת את מה שצפוי בהמשך הקלט. במילים אחרות, מהחרוזת שעל המחסנית ה- parser אמר לגור את מה שנותר מהקלט (החל מה- lookahead ועד לסוף הקלט). כל האסימונים שלפני

ה- lookahead כבר נגזרו כולם ה

העלים השמאליים בעץ הגזירה שהולך ונבנה
 כבר מתאימים לאסימונים שהופיעו בקלט לפני ה- lookahead. המטרה היא כموון לסיים את בניית עץ הגזירה כך שכל ה

העלים שלו יתאימו
 לקלט.

#### איך בונים את טבלת ה- (1)LL :

בטבלה תהיה שורה עבור כל משתנה של הדקדוק ועמודה עבור כל lookahead אפשרי כולם עמודה עבור כל אסימון (טרמינל) של הדקדוק ועמודה עבור \$.

כדי להחליט היכן בטבלה יש לרשום כל אחד מכללי הגזירה:  
מחשבים את ה- FIRST של צד ימין (הגוף) של כלל הגזירה. את הכלל רושמים בשורה של המשתנה בראש הכלל (צד שמאל של הכלל) ובכל העמודות שמתאימות לטרמינלים ב- FIRST חשוב.

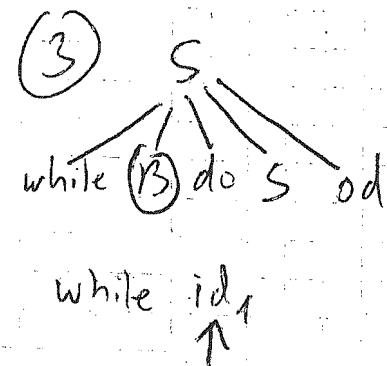
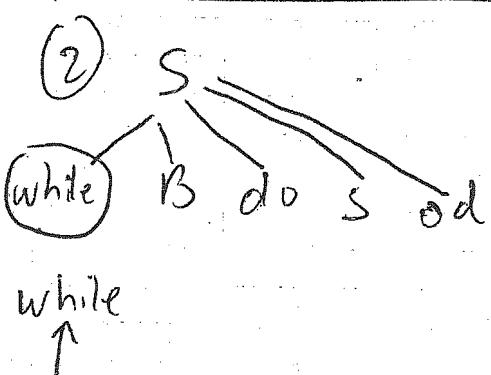
בנוסף לכך, אם ה- FIRST הנ"ל שחשוב כולל גם את המילה הריקה (כלומר הגוף של הכלל אפס -- ניתן לגוזר ממנו את המילה הריקה -- זה כולל גם את המקרה שהגוף של הכלל הוא המילה הריקה) אז יש לחשב את FOLLOW של המשתנה בראש הכלל ולרשום את הכלל בכל העמודות המתאימות לאיברי ה- FOLLOW. (כਮובן בשורה של המשתנה בראש הכלל).

אם בכל הנסיבות בטבלת ה- (1)LL יש לכל היותר כלל גזירה אחד (כלומר או שיש כלל גזירה אחד או שהכניסה ריקה) אז הדקדוק הוא מסווג (1)LL וזה אומר שניתן לעשות parsing לפי הדקדוק עם האלגוריתם שרשום לעיל. אבל אם יש (פחות) כניסה אחת שבה יותר משלם גזירה אחד (נקרא לתופעה זו "קונפליקט") אז הדקדוק אינו מסווג (1)LL וזה אומר שלא ניתן לעשות parsing עם האלגוריתם הנ"ל. הבעיה היא שכאשר ה- parser יגיש לכניסה בטבלה שיש בה מספר כללי גזירה הוא לא ידע באיזה מהם לבחור. (יש מקרים בהם בחירה באחד מהכללים תוביל לסיום מוצלח של בניית העץ ובחירה באחרים תכשל ויש גם מקרים בהם בחירה בכלים שונים תוביל לבניית עצי גזירה שונים (לאותו קלט)).

# top down parsing

(1) S

white  
↑



(4)

S  
white  
id  
R

white id

(5) S

white B do S od  
id R

white id do

(6) S

white B do S od  
id R E

white id do

(7)

S  
white B do S od  
id R E

white id do it

(8)

S  
white B do S od  
id R E B then S else S fi

white id do it

(9)

S  
white B do  
id R E  
if (B) then S else S fi

white id do it id<sub>2</sub>

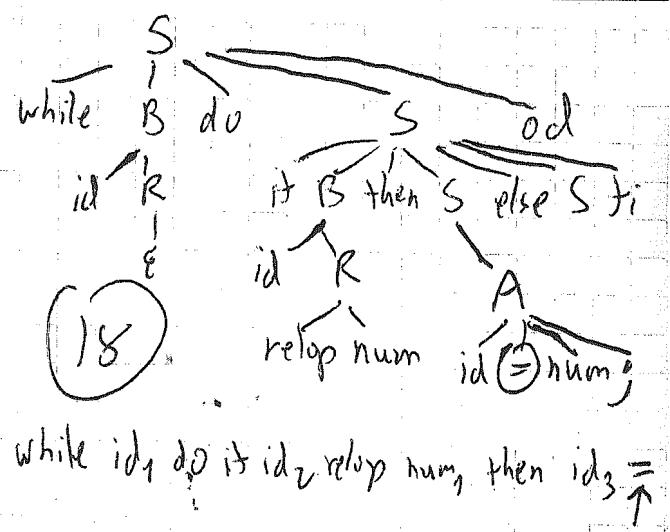
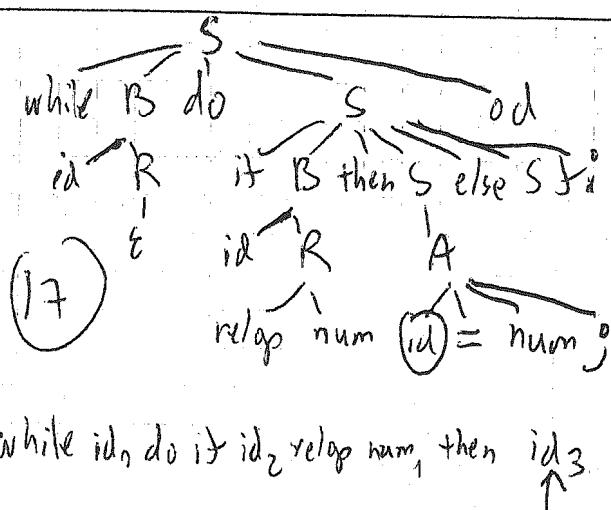
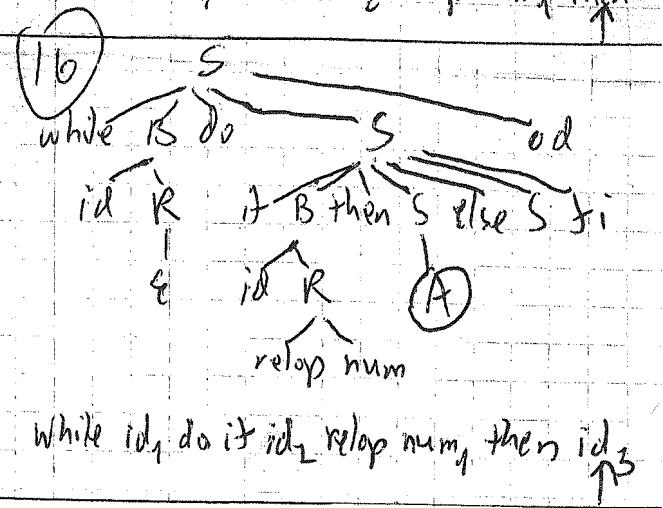
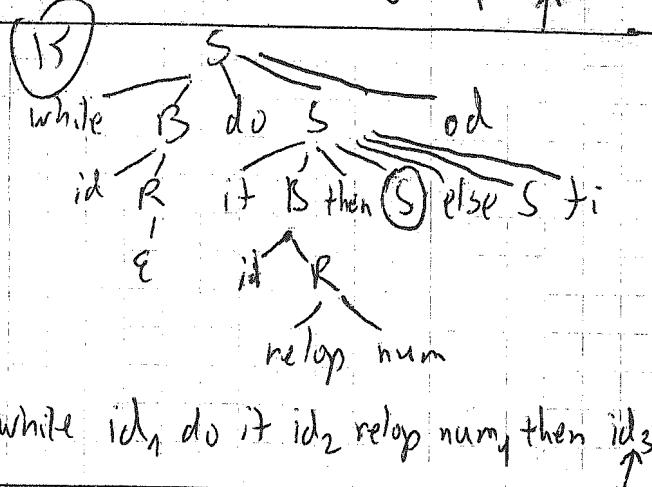
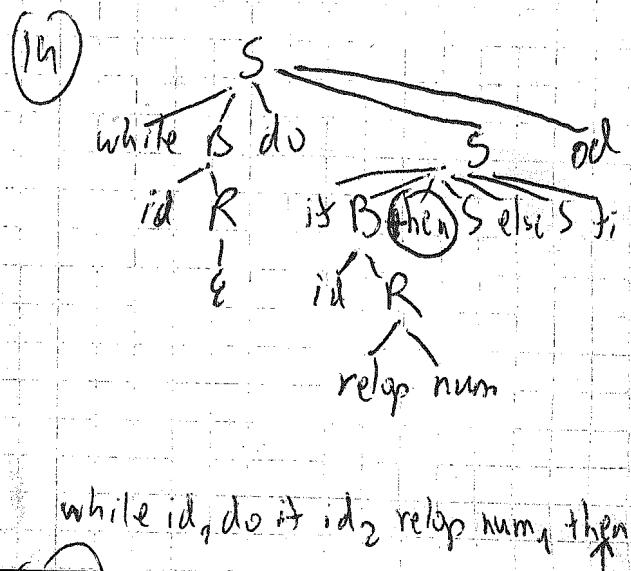
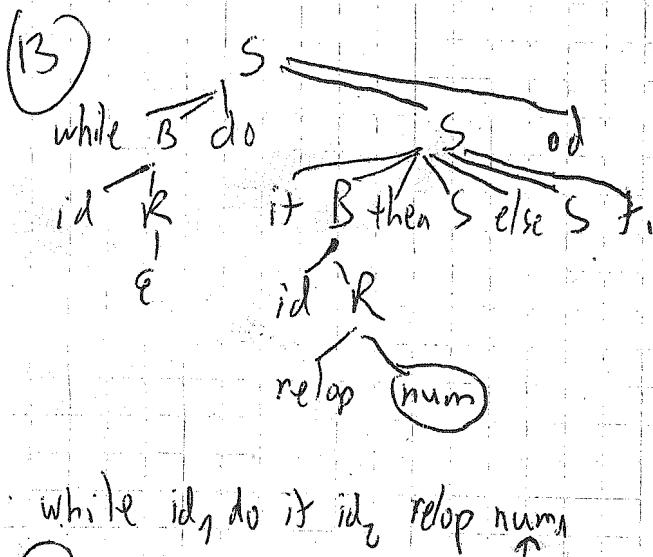
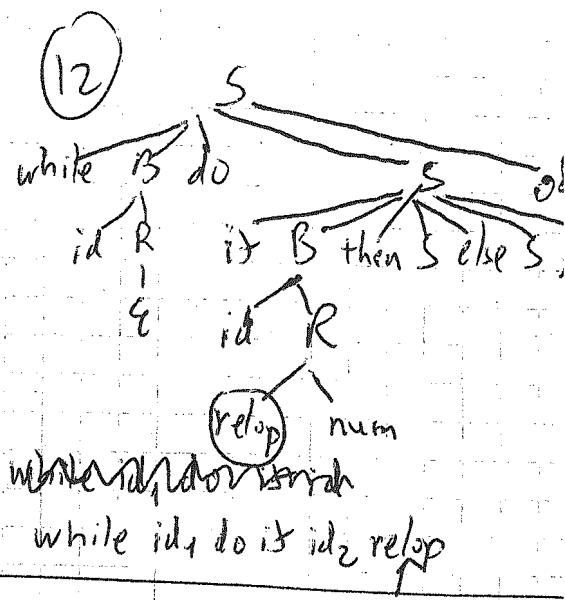
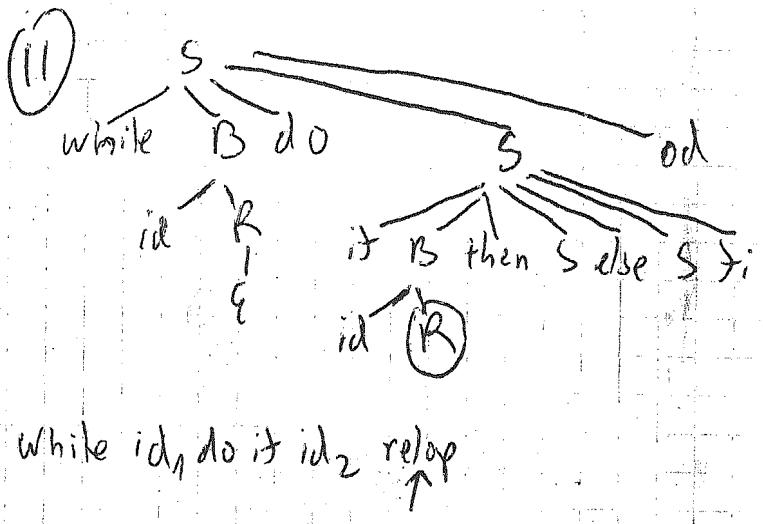
(10)

S  
white B do  
id R E  
if B then S else S fi  
(id) R

white id do if id<sub>2</sub>

→ joint else (B) if > A if > B or, > S or G >

· joint joint? B/B - DS JHO give same prob



## Recursive Descent Parser

נשתמש בדקדוק הבא כדוגמה  
 (מילים עם קו תחתון כמו למשל fi הם טרמינליים (איסימוניים). גם הסימנים '=' ו'-' הם טרמינליים).

- (1)  $\text{stmt} \rightarrow \underline{\text{if}} \text{ boolexp } \underline{\text{then}} \text{ stmt } \underline{\text{else}} \text{ stmt } \underline{\text{fi}}$
- (2)  $\text{stmt} \rightarrow \underline{\text{while}} \text{ boolexp } \underline{\text{do}} \text{ stmt } \underline{\text{od}}$
- (3)  $\text{stmt} \rightarrow \underline{\text{begin}} \text{ stmtlist } \underline{\text{end}}$
- (4)  $\text{stmt} \rightarrow \text{assignstmt}$
- (5)  $\text{stmtlist} \rightarrow \text{stmt} \text{ stmtlist}$
- (6)  $\text{stmtlist} \rightarrow \epsilon$
- (7)  $\text{boolexp} \rightarrow \underline{\text{id}} \text{ R}$
- (8)  $\text{R} \rightarrow \underline{\text{relop}} \text{ num}$
- (9)  $\text{R} \rightarrow \epsilon$
- (10)  $\text{assignstmt} \rightarrow \underline{\text{id}} = \underline{\text{num}} ;$

הדקוד הוא LL(1). הנה טבלת LL(1) עבור הדקוד הנטוון.  
 המספרים בטבלה מציננים כללי גזירה. הכנסות הריקות מציננות .error.

	if	the n	else	fi	whi le	do	od	begi n	end	id	=	num	;	relo p	\$
stmt	1				2				3	4					
stmtlist		5				5			5	6	5				
boolexp											7				
R			9										8		
assign stmt						9						10			

לצורך בניית הטבלה יש לחשב ראשית את FIRST ו-FOLLOW של משתני הדקדוק. בדקוד שחוובא למעלה מתקיים:

```

FIRST (stmt) = { if, while, begin, id }
FIRST (stmtlist) = {  $\epsilon$ , if, while, begin, id }
FIRST (boolexp) = { id }
FIRST (assignstmt) = { id }
FIRST (R) = {  $\epsilon$ , relop }

```

```

FOLLOW (stmt) = { $, else, fi, od, end, if,
                  while, begin, id }
FOLLOW (stmtlist) = { end }

```

```

FOLLOW (boolexp) = { then, do }
FOLLOW (assignstmt) = { $, else, fi, od, end, if,
while, begin, id }
FOLLOW (R) = { then, do }

```

הנה ה .parser

```

recursive descent parser
TOKEN lookahead;

void match (TOKEN token)
{
    if (token == lookahead)
        lookahead = lexer();
    else
        error ();
}

int main () {
    lookahead = lexer();
    stmt ();
    if (lookahead == $) /* end of input */
        return(SUCCESS);
    else return (FAILURE);
}

void stmt () {
    switch (lookahead) {
    case IF :
        /* use production
           stmt -> if boolexp then stmt else stmt fi
        */
        match(IF); boolexp(); match(THEN);
        stmt(); match(ELSE); stmt(); match(FI);
        break;

    case WHILE :
        /* use production
           stmt -> while boolexp do stmt od
        */
        match(WHILE); boolexp(); match(DO);
        stmt(); match(OD);
        break;
    case BEGIN :

```

```

/* use production stmt -> begin stmtlist end
*/
    match(BEGIN); stmtlist(); match(END);
    break;

case ID :
/* use production stmt -> assignstmt */
    assignstmt();
    break;

default:
    error ();
}

}

void stmtlist () {
switch (lookahead) {
case IF : case WHILE : case BEGIN : case ID :
/*use production stmtlist -> stmt stmtlist */
    stmt(); stmtlist();
    break;
default :
    return; /* use production stmtlist -> ε */
}
}

void boolexp () {
if (lookahead == ID) {
/* use production boolexp -> id R */
    match(ID); R ();
}
else error ();
}

void R () {
if (lookahead == RELOP) {
/* use production R -> relop num */
    match(RELOP); match(NUM);
}
/* else use production R -> ε */
}

void assignstmt () {
if (lookahead == ID) {

```

```

/* use production assignstmt -> id = num ; */
match(ID); match( = ); match(NUM);
match( ; );
}
else error();
}

```

### הערות

הfonקציה lookahead מקבלת אסימון כARGINENT. אם האסימון זהה ל- `id` היא מתקדמת בקלט: היא קוראת למתוח הלקסיקלי (שנקרא `anon()`) שמחזיר את האסימון הבא בקלט ואסימון זה הוא ה- lookahead החדש.

אם האסימון שהועבר כARGINENT ל- `match` אינו זהה ל- `id` אז גילינו `syntax error`. בכך למטה פשוט קוראים לפונקציה `error()`. זו אמורה לפחות להוציא הודעה שגיאה.

עבור כל משתנה של הדקדוק מוגדרת פונקציה. פונקציה עבור משתנה A יודעת לקרוא מהקלט מחרוזת הניתנת לגזירה מ- A ולבנות מעבר על תח העץ המתאים לה בעץ הגזירה (لتת עץ זה שורש A). העץ לא "באמת" נבנה בכך הניל' אבל לא קשה לבנות מבנה נתונים המיציג את העץ אם רוצים.

הfonקציה `stmt` למשל קוראת משפטים מהקלט ועובד להם ניתוח תחבירי (כלומר עושה מעבר על תח העץ המתאים). הפונקציה `boolexp()` קוראת ביטויים בוליאניים מהקלט ועובד להם ניתוח תחבירי וכן הלאה.

כל קריאה לפונקציה של משתנה של הדקדוק מתאימה לצומת בעץ הגזירה. כל קריאה לפונקציה match מתאימה לעלה של עץ הגזירה. המעבר על עץ הגזירה הוא ב- DFS (כאשר המעבר על ילדים של צומת נעשה באופן רקורסיבי משמאלי לימיין).

לכל הפונקציות המוגדרות עבור משתנים של הדקדוק יש מבנה דומה. הפונקציה עבור משתנה A מחייבת באיזה כל גזירה להשתמש לפי ה- lookahead. עבור דקדוקים שהם LL(1) (כמו הדקדוק הנוכחי) זה אפשרי עם lookahead בגודל 1. ההחלטה באיזה כל גזירה להשתמש מתאימה לטבלת ה- LL(1) של הדקדוק (אם כי אין כאן טבלת LL(1) נפרצת). לנוכח הטעמיות להכין טבלת (1) LL של הדקדוק לפני שכותבים את ה- `parser`.

מה עושים עם דקדוקים שאינם LL(k) ? אם הדקדוק הוא LL(1) אז ה- `parser` יכול להסתכל על k האסימונים הבאים בקלט כדי לדעת באיזה כל גזירה להשתמש. אפשרות אחרות היא להסתכל קדימה בקלט ככל הנדרש כדי לבחור בכל גזירה. יש גם אפשרות לעשות backtracking שזה אומר שככל פעם נדרש לבחור כל גזירה -- מנסים את כל גלילי הגזירה עד ש모וצאים כל שעובד כלומר אפשר לסיים את ה- `parsing` בהצלחה.

אבל כאן עוסוק רק ב- parser שפועל לפי דקדוק שהוא (1)LL ולכן האסימון הבא בקếtל מאפשר לו לדעת באיזה כל גזירה לבוחר.

אחרי שהחליט באיזה כל גזירה להשתמש ה- parser מטפל בגורם של כל הגזירה (צד ימין של כל הגזירה) ע"י מעבר על סימני הדקדוק שמאל לימין. עברו כל טרמינל הוא עושה קריאה ל- .match. עברו כל משתנה הוא קורא לפונקציה שהוגדרה עבור המשתנה.

אם כל הגזירה שנבחר הוא כלל אפסילון (בצד ימין יש אפסילון) הפונקציה מיד חזרת שכן היא סימנה לעבר על תת העץ המתאים שכולך רק על המסומן באפסילון.

אם אין כל גזירה שמתאים לו lookahead או הפונקציה גילתה syntax error ולכן היא קוראת לו ()error.

אם למשנה A יש כלל epsilon  $\epsilon \rightarrow A$  אז ניתן להשתמש בו במקום לקרוא לו ()error. מובטח שהשגיאה תtgtלה בהמשך. זה אנלוגי לרשום הכלל  $\epsilon \rightarrow A$  בכל הנסיבות הריקות בשורה של A בטבלת LL(1).

parser recursive descent parser עבור דקדוק שהוא LL(1) הוא אנלוגי לו parser שמשתמש בטבלת LL(1). אבל הקוד שלו יכול להיות קל יותר להבנה. בעוד ש- parser שמשתמש בטבלת LL(1) עושה שימוש מפורש במחסנית, recursive descent parser עושה שימוש במחסנית "מאחורי הקלעים" -- זו מחסנית זמן הריצה שבה נשמרות רשומות הפעלה של הפונקציות.

## ניתוח תחבירי בשיטת LR (LR parsing)

ניתוח תחבירי בשיטת LR היא שיטה הפועלת bottom up כלומר הינה עוברת על עצה הגזירה של הקלט מלמטה למעלה.

המנתח מנסה לצמצם את מילת הקלט למשתנה ההתחלתי של הדקדוק על ידי ביצוע סידרה של פעולות צימצום.

נגידר: צימצום (פעולות reduce) היא החלפה של המחרוזת שנמצאת באגף ימין של כל גזירה במשתנה שנמצא באגף שמאל של הכלל. לדוגמה צימצום לפי הכלל

$\text{id} \rightarrow B$  →  $B \rightarrow \text{id}$  relop  $\text{id} \rightarrow \text{id}$  → B.

פרש הדבר שהמנתח מזהה שהסימנים  $\text{id}$  relop  $\text{id}$  מהווים ביחד B (בדוגמא זו B משתנה המייצג ביטויים בוליאניים).

שיטת LR חזקה יותר מ-predictive parsing בכך שהיא ישימה עבור מספר גדול יותר של דקדוקים. מרבית המבנים התחביריים שימושיים בשפות תיכנות ניתנים לניתוח בשיטה זו.

### ביצד פועל המנתח

מנתח תחבירי בשיטת LR משתמש ב-

- מחסנית שבה נשמרים סימני דקדוק ומצבים של אוטומט סופי דטרמיניסטי.
- טבלת ניתוח תחבירי שמכילה שורה עבור כל מצב של האוטומט. הטבלה מחולקת לשני חלקים. החלק האחד מכיל עמודה עבור כל טרמינל (ועבור \$ המסמל את סוף הקלט) והוא מתאר את הפעולות שעל המנתח לבצע. חלק זה מכונה טבלת ה- action. החלק השני מכיל עמודה עבור כל משתנה. חלק זה מכונה טבלת ה- goto.

### טבלת ה action מכילה ארבעה סוגים פעולות :

- j shift (מופיע בטבלה כ- j כאשר j הוא מספר המיצג מצב של האוטומט). במסגרת פעולה זו המנתח מבצע את הפעולות הבאים : דוחף את סימן הקלט הנוכחי (שהוא טרמינל) למחסנית. דוחף את המצב j למחסנית. מקדם את המצביע לקלט לסימן הקלט הבא.

- reduce i (מופיע בטבלה כ- i כאשר i הוא מספר של כלל גזירה). נניח שככל i הוא  $\alpha \rightarrow A$  במסגרת פעולה זו המנתח מבצע את הפעולות הבאים :

מוציא מהמחסנית  $| \alpha |$  סימנים. ( $| \alpha |$  מסמל את מספר הסימנים ב-  $\alpha$ ). קלומר עושים קוק לכל הסימנים שבדצ ימין של כלל הגזירה  $\alpha$  ולכל המצביעים שכיסו אותם במחסנית. בעיקבות כך נחשף בראש המחסנית מצב שנקרה לו s . המנתח דוחף למחסנית את המצב [ s, A ]. (קלומר את המצב שמוופיע בשורה עבור s ובעמודה A בטבלת ה goto ). בשלב זה המנתח יכול לעשות פעולה סמנטיביות שוניות הקשורות לכל הגזירה  $\alpha$  . נניח לעת עתה שהוא משתמש בהדפסת כלל הגזירה לפיו

עשה את ה- reduce.

- accept (מופיע בטבלה לדוגמא בהמשך כ- acc). המנתח התחריבי עוצר ומכריז על הצלחה (המנתח "מקבל" את הקלט).
- error כל משבצת בטבלה שהיא ריקה משמעותה "שגיאה". המנתח עוצר ומכריז על שגיאה (כלומר הקלט אינו שיך לשפה המוגדרת ע"י הדקוק).

### תאור האלגוריתם לניטוח תחריבי LR

קלט: מחרוזת קלט w וטבלת ניטוח תחריבי LR עבור דקוק G.

פלט: אם w בשפה (G) L אז ניטוח תחריבי של w. אחרת המנתח יכריז שמצא שגיאה.

בתאור שיוואה כאן המנתח ידפיס לפט תאו ר של גזירה של הקלט w. וביתר פרוט: בפלט יופיעו כללי הגזירה המופעלים בגזירה ימנית ביותר של w. הכללים יופיעו בפלט בסדר הפוך ככלל האחורי המופעל בגזירה ימנית ביותר של w יופיע ראשון בפלט

איתחול: המצביע לקלט מצביע לסימן הראשון בקלט. המחסנית מכילה רק את s שהוא המצב ההתחלתי של האוטומט.

האלגוריתם יבצע בלולאה את הצעדים הבאים :

נניח שסימן הקלט הנוכחי הוא a ושה מצב בראש המחסנית הוא s.

1. בצע את הפעולה הרשומה בקונסעה [s, a] action (זו הקונסעה בטבלה ה- action שנמצאת בשורה עבור s ובעמודה a).
2. אם הפעולה שזה עתה בוצעה היא accept או error אז עצור אחרת תמשיך בלולאה.

דוגמת ריצה של האלגוריתם  
נשתמש בדקוק הבא לצורך הדוגמה.

- (1) S → while B do S od
- (2) S → begin SQ end
- (3) S → assign
- (4) SQ → SQ S
- (5) SQ → ε
- (6) B → id
- (7) B → id relop id

כאן S מייצג משפטי (statements), SQ מייצג סדרות של משפטי (sequences) ו- B מייצג ביטוייםبولיאניים. assign הוא טרמינל המייצג משפטי השמה.

הטבלה הבאה מתאימה לדקוק הנתון.

action	goto
--------	------

	<u>whi</u>	<u>do</u>	<u>od</u>	<u>beg</u>	<u>end</u>	<u>assi</u>	<u>id</u>	<u>relo</u>	\$	S	SQ	B
	<u>le</u>			<u>in</u>		<u>gn</u>		<u>p</u>				
0	s2			s3		s4				1		
1									acc			
2							s6				5	
3	r5			r5	r5	r5					7	
4	r3			r3	r3	r3			r3			
5		s8										
6		r6						s9				
7	s2			s3	s10	s4				11		
8	s2			s3		s4				12		
9							s13					
10	r2			r2	r2	r2	r2		r2			
11	r4				r4	r4	r4					
12		s14										
13		r7										
14	r1		r1	r1	r1	r1			r1			

הנה דוגמא להרצת האלגוריתם :

מה בוצע	יתרת הקלט	תוכן המחסנית
	while id relop id do begin assign assign end od \$	0
	id relop id do begin assign shift assign end od \$	0 while 2
	relop id do begin assign shift assign end od \$	0 while 2 id 6
	id do begin assign assign shift end od \$	0 while 2 id 6 relop 9
	do begin assign assign end shift od \$	0 while 2 id 6 relop 9 id 13
	do begin assign assign end reduce od \$ 7	0 while 2 B 5
	begin assign assign end od shift \$	0 while 2 B 5 do 8
	assign assign end od shift assign assign end od reduce 5	0 while 2 B 5 do 8 begin 3
	assign end od shift assign end od reduce 3	0 while 2 B 5 do 8 begin 3 SQ 7
	assign end od \$	assign 4
	assign end od \$	0 while 2 B 5 do 8 begin 3 SQ 7 S
	reduce 3	11

0 while 2 B 5 do 8 begin 3 SQ 7	assign end od \$	reduce 4
0 while 2 B 5 do 8 begin 3 SQ 7 assign 4	end od \$	shift
0 while 2 B 5 do 8 begin 3 SQ 7 S 11	end od \$	reduce 3
0 while 2 B 5 do 8 begin 3 SQ 7	end od \$	reduce 4
0 while 2 B 5 do 8 begin 3 SQ 7 end 10	od \$	shift
0 while 2 B 5 do 8 S 12	od \$	reduce 2
0 while 2 B 5 do 8 S 12 od 14	\$	shift
0 S 1	\$	reduce 1
0 S 1	\$	accept

#### הערות:

- אין צורך לשמור סימני דקדוק על המחסנית – די לשמור שם את מצב הautomat. סימני הדקדוק בכל זאת נשמרים כאן על המחסנית כדי שייהי קל יותר להבין מה קורה.
- המצב בראש המחסנית הוא תמיד המצב אליו היה מגע automat אליו קרא את כל סימני הדקדוק על המחסנית (החל מהתיקון בקרקעיתה ועד לסימון הראשה).

גזרה ימנית ביותר היא גזרה שבה בכל צעד נגור המשטנה הימני ביותר. מהלך הניתוח התחבירי עוקב אחר גזרה ימנית ביותר של הקלט מסוף הגזרה להתחלה. לכן השיטה נקראת LR parsing : L – LR parsing R – rightmost derivation (גזרה ימנית ביותר). R בغالל שהמנתח מוצא R – rightmost derivation (גזרה ימנית ביותר). הסבר מפורט יותר על עניין זה מופיע בגרסה הארוכה יותר של קובץ זה הנמצא באתר הקורס.

המחזרות אותה יש לצמצם כדי לעבור למבנה הפסוקית הקודמת (בגזרה ימנית ביותר של הקלט) נקראת handle.

האלגוריתם של המנתח התחבירי ניתנו לתימצאות כך:  
בעוד את הצעדים הבאים שוב ושוב עד שmotgala שגיאה או שמצמצמים – S

- כל עוד אין handle בראש המחסנית בצע shift .
- (כעת הסימנים שניצבו בראש המחסנית מהווים handle reduce בצע .)

S' הוא המשתנה התחלתי בדקדוק המורחב – זה מוסבר בהמשך).

הערה : ה-handle שמצמצמים תמיד בראש המחסנית כלומר לא קורה שמצמצמים handle ש"קבור" במחסנית מתחת לסימנים אחרים.

### הבעיה של המנתח התחבירי היא ליזות handles

לא ניתן ליזות handles על ידי כך שפשות סורקים את הקלט משמאלו לימין עד שמוצאים מחרוזת המהווה את אגף ימין של איזה שהוא כלל גזירה. ברור שזה נכון עבור כללי גזירה שאגף הימני הוא (שערי ע' "נמצא" בכל מקום בקלט) אבל הבעיה קיימת גם עבור כללי גזירה אחרים.

נתבונן לדוגמה במילת הקלט הבאה (זו הדוגמא שראינו קודם – כאן סומן ה-handle בקו תחתון).

while id relop id do begin assign assign end od

ה-id הראשון במילת הקלט אינו מהו handle למרות שהוא מתאים לאגף ימין של כלל גזירה id → B. אם נצטetz לפि כלל זה נקבל :  
while B relop id do begin assign assign end od  
כלומר לא נצליח לצמצם אותה למשתנה התחלתי והኒתו התחבירי יכשל. ניתן להשתכנע בכך גם באופן הבא : אילו היינו מצליחים לצמצם מחרוזת זו למשתנה התחלתי אז (לפי הגדרת תבנית פסוקית) היא הייתה תבנית פסוקית ואז relop היה ב- (B) FOLLOW (כי לפניו תבנית פסוקית) שבה relop מופיע מיד אחרי B) אבל relop אינו ב- (B) FOLLOW (ב- FOLLOW אפשר להפעיל את האלגוריתם לחישוב FOLLOW ולהיווכח בכך).

גם כאשר המנתח מצליח למצא handle נותרה השאלה לפי איזה כלל גזירה מצמצמה. למשל במקרה הזה, לאחר קריאת 4 הסימנים הראשונים מתעוררת השאלה : האם ה-handle הוא ה-id השני או שמא id relop .

את הבעיה של זיהוי handles פותרים בעזרת אוטומט סופי דטרמיניסטי כפי שנראה.

כיצד בונים את טבלת הניתוח התחבירי  
יש (לפחות) שלוש שיטות לבנות את הטבלה והן כולן מבוססות על רעיון דומה.  
1. LR קוני – זו השיטה הכללית ביותר כלומר היא ישימה עבור המספר הגדול ביותר של דקדוקים. רוב המבנים התחביריים שימושיים בשפות תיכנות ניתנות לניתוח בשיטה זו. דקדוק שהשיטה הזו עובדת עבورو (כאשר ה- lookahead בגודל 1) נקרא דקדוק (1) LR .

הבעיה בשיטת ה- LR היא שהטבלאות שהיא מניבה עלולות להכיל הרבה מאוד שורות.

2. אנו נלמד רק את השיטה שנקראית SLR (simple LR). זו השיטה פשוטה מבין השלוש והיא מצליחה עבור פחות דקדוקים יחסית לשיטות האחרות. הטבלה

שבדונים נקראית טבלת SLR ודקודק שהשיטה ישימה עבورو נקרא דקודק (1) או פשוט דקודק SLR (לא עוסק ב- lookahead גדול אחד).

3. שיטת LR (look ahead LR). זו שיטה שישימה עבר מספר רב יותר של דקודקים מאשר SLR ומוניבה טבלאות קטנות יותר מאשר שיטת LR. yacc ו – bison (מחוללי מנתכים תחביריים פופולריים) משתמשים בה.

יש הבדל משמעותי בין דקודקי LR לבין דקודקי LL: כדי שדקודק יהיה (1) LR علينا להיות מסוגלים להזות שימוש בכל גזירה מסוים אחרי שרairoן בקלט את כל מה שנוצר מאגף ימין של הכלל וגם ראיינו סימן קלט נוסף (ה – lookahead). לעומת זאת כדי שדקודק יהיה (1) LL علينا להיות מסוגלים להזות שימוש בכל גזירה מסוים אחרי שרairoן רק את הסימן הראשון שנוצר מאגפו הימני. לכן קבוצת דקודקי (1) LR מכילה ממש את קבוצת דקודקי (1) LL.

### בנייה טבלת SLR

לצורך בניית הטבלה נבנה אוטומט סופי דטרמיניסטי (DFA). המ מצבים של האוטומט זהה הם המ מצבים שמופיעים במחסנית בזמן הניתוח התחבירי וקובעים (בעזרת סימן הקלט הבא (ה – lookahead)), מה יהיה הצעד הבא בכל שלב. לצורך בניית האוטומט נזדקק למספר הגדרות.

נדיר : פריט (0) item (0) LR (0) ובקיצור : פריט, הוא כלל גזירה שנקדזה מופיעה באגף ימין שלו.  
דוגמאות :

$$\begin{aligned} S &\rightarrow \bullet \text{while } B \text{ do } S \text{ od} \\ S &\rightarrow \text{while } \bullet B \text{ do } S \text{ od} \\ S &\rightarrow \text{while } B \bullet \text{ do } S \text{ od} \\ (\text{כאן כלל הגזירה הוא } & \epsilon \rightarrow SQ \rightarrow \bullet) \end{aligned}$$

הערה : לעיתים נקייף פריטים בסוגרים מרובעות כדי להקל על הקריאה לדוגמא [ while B • do S od ]  $\rightarrow$  S  $\rightarrow$ .

הכוונה היא שימוש הנקדזה יראה לנו את השלב אליו הגיעו בניתוח התחבירי. לדוגמה, הפריט החלישי למעלה אומר שכבר ראיינו את הטרמינל while (ועשינו לו shift), לאחריו ראיינו בקלט סידרה של טרמינלים שהצלחנו למצטם ל – B (כלומר הם מהווים ביחד ביטוי בוליאני) וכרגע שני הסימנים בראש המחסנית הם while ו – B (הסימן B נמצא בראש ו – while מתחתיו). עתה אנו מצפים למצוא בקלט את הדברים הבאים (בסדר זה) :

הטרמינל do  
משפט (מחירות של טרמינלים שניתנו לצמצם ל – S)  
הטרמינל od

אם וכאשר אכן נראה את כל אלו בקלט ונចBOR בראש המחסנית את הסימנים while B do S od (אשר מתוכם צברנו בשלב זה כאמור רק את while B do) אז .S  $\rightarrow$  while B do S od reduce נעשה לפי הכלל reduce

כפי שנראה, אנו נזהה כל מצב באוטומט עם קבוצה של פריטים.

נגידר שתי פעולות על קבוצות של פריטים. הפעולה הראשונה היא הסגור של I (באנגלית : (I) closure) כאשר I היא קבוצה של פריטים.

בהתאם קבוצת פריטים I, נבנה את הסגור שלו בעזרת שני כלליים :

1. כל פריט ב- I נמצא בסגור של I.

2. נפעיל כלל זהה כל עוד ניתן להוסיף פריטים נוספים לסגור :

אם הפריט  $\beta \bullet A \rightarrow \alpha$  בsburg ו-  $\gamma \rightarrow B$  הוא כלל גזירה אז נוסיף

את הפריט  $\gamma \bullet B \rightarrow$  לsburg.

הסיבה לכלל השני היא כזו : מן הפריט  $\beta \bullet A \rightarrow \alpha$  משתמע שאנו מצלפים למצוא בקלט B (כלומר שהוא שנייה לגזירה מ- B). יתכן ש- B זה יהיה "מורכב" מ-  $\gamma$  ( $\gamma$  היא לפי הסימונים הרגילים שלנו סידרה של סימני דיקדוק). לכן יתכן שנמצא בקלט  $\gamma$  (כלומר שהוא שנייה לגזירה מ-  $\gamma$ ).

דוגמה : נשתמש בבדיקה הבא :

- (0)  $E' \rightarrow E$
- (1)  $E \rightarrow E + T$
- (2)  $E \rightarrow T$
- (3)  $T \rightarrow T * F$
- (4)  $T \rightarrow F$
- (5)  $F \rightarrow ( E )$
- (6)  $F \rightarrow \underline{\text{num}}$

נחשב את הסגור של קבוצת הפריטים  $\{ [E' \rightarrow \bullet E] \}$  לפי הכלל הראשון, הסגור יכלול את הפריט  $[E \rightarrow \bullet E']$ . נשתמש עתה בכל השני :

מהחר והמשתנה E מופיע מיד אחרי הנקודה בפריט  $[E' \rightarrow \bullet E]$ . נוסיף לsburg את הפריטים הבאים :  $T [E \rightarrow \bullet E + T], [E \rightarrow \bullet T]$ ,  $F [E \rightarrow \bullet E * F]$ ,  $( E ) [E \rightarrow \bullet ( E )]$ . בפריט הראשון שזה עתה הוסףנו לsburg, המשתנה E מופיע מיד אחרי הנקודה וזה אומר שצורך להוסיף לsburg את אותם שני פריטים שכבר הוסףנו הרגע.

בפריט  $[E \rightarrow \bullet T]$ , T מופיע מיד אחרי הנקודה ולכן נוסיף לsburg את הפריטים  $[T \rightarrow \bullet F], [T \rightarrow \bullet T * F]$ . בגל שבספריט השני, F מופיע מיד אחרי הנקודה, נוסיף את הפריטים  $[F \rightarrow \bullet \underline{\text{num}}], [F \rightarrow \bullet ( E )]$  ובזאת סיימנו.

הפעולה השנייה שנגידר היא (I, X) goto כאשר I היא קבוצה של פריטים ו- X הוא סימן דקדוקי (משתנה או טרמינל). נגידר : (I, X) goto (I, X) היא הסגור של קבוצת כל הפריטים מהצורה  $\beta \bullet X \rightarrow \alpha$  כך שהפריט  $\beta [A \rightarrow \alpha]$  נמצא ב- I.

דוגמה :

נניח ש- I היא קבוצת הפריטים  $\{ [B \rightarrow \bullet \underline{\text{id}}], [B \rightarrow \bullet \underline{\text{id}} \underline{\text{relop}} \underline{\text{id}}] \}$

או (id, I) goto יהיה הסגור של קבוצת הפריטים { ] id • relop id → [ B ]. ב מקרה זה הסגור יכול את אותם שני פריטים ושותם דבר מעבר לכך.

(X, I) goto עברו כל סימן דקדוקי X השונה מ- id יהיה במקרה זה ריק (כלומר קבוצה של פריטים המכילה 0 פריטים) כי הסימן היחיד שמופיע מיידי לנוקודה בפריט כלשהו ב- I הוא id. למשל (I, while) goto הוא ריק.

פונקציה ה- goto תשמש כפונקציה המעבירים של האוטומט. לדוגמה אם J = (I, X) goto אז אם האוטומט במצב I (או מתיחסים כאן למצבidal קבוצה של פריטים I) והוא רואה את הסימן X אז הוא יעבור למצב J. במילים אחרות, בדיאגרמה עבור האוטומט תופיע קשת המסומנת ב- X והמובילה מהמצב I למצב J.

נגידר : דקדוק מורחב (augmented grammar) עבור דקדוק נתון G עם משתנה התחלתי S הוא הדקדוק G עם משתנה התחלתי חדש 'S' וכל גזירה S → 'S'.

הכוונה היא שכאשר המנתח התחבירי עומד בפני צימצום (reduce) לפי הכלל  $S \rightarrow 'S'$  (וסימן הקלט הבא יהיה \$) הוא יעצור "ויקבל" את הקלט (כלומר יכיריז שהኒותה התחבירי הצליח ומילת הקלט שיצכת לשפה המוגדרת על ידי הדקדוק). הערה : אם המשתנה ההתחלתי המקורי S אינו מופיע באגד ימין של כלל גזירה כלשהו או ניתן לו יותר על הרחבות הדקדוק והמנתח התחבירי יקבל את הקלט כאשר הוא יעשה reduce לפי כלל שבצדיו השמאלי S (וסימן הקלט הבא יהיה \$).

נראה עתה את האלגוריתם לבניית מבני האוטומט. כאמור, כל מצב של האוטומט מזוהה עם קבוצת פריטים. לכן האלגוריתם בונה אוסף של קבוצות של פריטים. אוסף זה נקרא אוסף הקבוצות המקורי (0) של הדקדוק הנתון.

### אלגוריתם לבניית מבני האוטומט

קלט : דקדוק מורחב

פלט : קבוצת מבני האוטומט שנסמנה C.

מהלך האלגוריתם :

1. המצב ההתחלתי של האוטומט הוא ( { [ S ' → • S ]. closure ( { [ S ' → • S ].natichil את C כך שיכיל רק מצב זה.
  2. כל עוד ניתן להוסיף מצבים נוספים ל- C, נבצע :
- עבור כל מצב I שנמצא ב- C וסימן דקדוקי X כך ש- (X, I) goto אינו ריק ואני נמצא כבר ב- C  
מוסיף את (X, I) goto ל- C.

הערות :

-- פונקציה ה- goto משמשת כפונקציה המעבירים של האוטומט כאמור לעיל.

-- המצב ההתחלתי של האוטומט הוא הסגור של  $[S \rightarrow 'S]$  כי בתחילת אנו מצפים למצוא בקלט מחרוזת של טרמינלים שנייתת לגזירה מ-  $S$  והתוכנית היא שלאחר שנמצאים מחרוזת זו ל-  $-S$ , נמצאים את ה-  $-S$ .

### אלגוריתם לבניית טבלת SLR

קלט: דקדוק מורחב 'G'.  
פלט: טבלת ניתוח תחבירי SLR.

האלגוריתם:

1. בנה את האוטומט כפי שראינו.
2. את הפעולות עבור מצב  $i$  קבועים כך:
  - א. אם הפריט  $[A \rightarrow \alpha \bullet a \beta] \rightarrow [A \rightarrow \alpha \bullet a \beta]$  נמצא ב-  $i$  ו-  $j = (i, a) = j$  אז כתוב  $j$  shift בכניסה  $[i, a]$ . action  $a$  הוא טרמינל.
  - ב. אם הפריט  $[A \rightarrow \alpha \bullet a \beta] \rightarrow [A \rightarrow \alpha \bullet a \beta]$  נמצא ב-  $i$ , אז עברו כל  $a$  (טרמינל או  $\$$ ) שמצא ב-  $(A, FOLLOW)$ , כתוב בכניסה  $[i, a]$  action  $a$ .
  - ג. אם הפריט  $[S \rightarrow 'S \bullet a \beta] \rightarrow [S \rightarrow 'S \bullet a \beta]$  נמצא ב-  $i$  אז כתוב accept בכניסה  $[i, \$]$ .

אם נוצר קונפליקט כלומר יש כניסה בטבלה שבה רשומה יותר פעולה אחת אז האלגוריתם נכשל והדקדוק אינו SLR. (קונפליקט זה דבר רע כי אם המנתה התחבירי נזקק לכניסה בה מופיע קונפליקט הוא אינו יודע באיזו מהפעולות המופיעות בכניסה לבוחר).

3. את טבלת ה-  $goto$  ממלאים לפי פונקציית ה-  $goto$ : לכל משתנה  $A$ , אם  $j = (i, A)$  אז כתוב  $j$  בכניסה  $[i, A]$ .
4. כל כניסה בטבלה שנותרה ריקה אחרי צעדים 3, 2 – מציין שגיאה.

הסבר:

הפריט  $[A \rightarrow \alpha \bullet a \beta]$  משמעותו שצברנו על המחסנית רק חלק מה-*handle*, קרי את המחרוזות  $\alpha$  (ייתכן שיש מתחתייה עוד סימנים במחסנית). ה"תוכנית" היא לצבור על המחסנית גם את  $a$  ולאחריו את המחרוזות  $\beta$  ואז לצמצם את ה-*handle*  $\beta a \rightarrow -A$ . לכן, אם סימן הקלט הבא הוא  $a$ , נעשה shift כלומר נדחוף אותו למחסנית.

הפריט  $[A \rightarrow \alpha \bullet a \beta]$  משמעותו שצברנו את ה-*handle*  $\alpha$  בראש המחסנית וניתן  $handle$   $\beta$  ליצומו ל-  $-A$ . אם סימן הקלט הבא אינו ב-  $(A, FOLLOW)$  או  $\alpha$  אינה צריכה לצלצלו (יש בה משך דוגמא שמדגימה עניין זה).

דוגמה

הטבלה שהובאה לעיל נבנתה לפי האלגוריתם הזה.  
לצורך הוכנה צריך לחשב את  $FOLLOW$  עבור כל משתנה בדקדוק.  
הנה תוצאות חישוב זה:

$FOLLOW(S) = \{ \underline{od}, \underline{while}, \underline{begin}, \underline{assign}, \underline{end}, \$ \}$

$FOLLOW(SQ) = \{ \underline{while}, \underline{begin}, \underline{assign}, \underline{end} \}$

$\text{FOLLOW}(B) = \{ \underline{\text{do}} \}$

נראה לדוגמה כיצד מולאה השורה עבור מצב 6 בטבלה.

מצב 6 כולל את הפריט  $\bullet \text{id} \rightarrow \underline{\text{relop}}$ . פריט זה אומר שכאשר סימן הקלט הבא הוא  $\underline{\text{relop}}$  צריך לעשות shift. מאחר ש-  $\underline{\text{relop}} = 9$  goto (6) נכתוב 9 בסעיף action [6,  $\underline{\text{relop}}$ ] כל זה לפי סעיף 2 באלגוריתם.

הסבר: הפריט האמור אומר שכבר ראיינו  $\underline{\text{id}}$ , עשינו לו shift והוא כרגע בראש המחסנית (מתחתיו יש סימנים נוספים). לפי הפריט, ה-  $\underline{\text{id}}$  הזה לבדו הוא רק התחלת של handle. כדי לקבל את כל ה- handle בראש המחסנית (ואז לצמצמו ל- B לפि כלל הגזירה  $\text{id} \rightarrow \underline{\text{id}}$ ), צריך תחילת לצבור על המחסנית גם את  $\underline{\text{relop}}$  ו-  $\underline{\text{id}}$  נוסף. لكن אם סימן הקלט הבא הוא  $\underline{\text{relop}}$ , נעשה לו shift.

הפריט  $\bullet \underline{\text{id}} \rightarrow B$  שגם הוא כולל במצב 6 אומר שאנו במצב 6 וסימן הקלט הבא הוא ב- (B) FOLLOW אז צריך לעשות reduce לפि הכלל  $\underline{\text{id}} \rightarrow B$  (כלל מספר 6). לכן נכתוב  $\underline{\text{id}} \rightarrow 6$  כאן מצין את המספר של כלל הגזירה) בכניסה [6,  $\underline{\text{do}}$ ] action . כל זה בהתאם לסעיף 2 באלגוריתם.

הסבר: פריט זה אומר שהוא  $\underline{\text{id}}$  שנמצא בראש המחסנית מהו זה handle וניתן לצמצמו לפि הכלל  $\underline{\text{id}} \rightarrow B$ . העובדה שעושים את הצימצום רק במקרה שסימן הקלט הבא נמצא ב- (B) FOLLOW מנעה כאן קונפליקט: אילו היינו עושים את הצימצום ללא תנאי אז במקרה שהמנתח היה במצב 6 וסימן הקלט הבא היה  $\underline{\text{relop}}$  אז הוא לא היה יודע מה לעשות: shift או reduce.

הערה: אין טעם לצמצם ל- B כאשר סימן הקלט הבא הוא  $\underline{\text{relop}}$  (או כל סימן אחר שאינו ב- (B) FOLLOW) כי אז אין סיכוי שהнтוחות התחבירי ייצלח.  
מדובר ? נניח שכן נמצאים ל- B כאשר סימן הקלט הבא הוא  $\underline{\text{relop}}$ . אז במחסנית נקבל מהצורה  $B^\alpha$  כאשר  $\alpha$  מחרוזת של סימני דקדוק (נתעלם ממצביע האוטומט שנמצאים גם הם על המחסנית). שארית הקלט היא מהצורה  $w \underline{\text{relop}}$  (כאשר  $w$  היא מחרוזת הטרמינלים שמופיעים בקלט אחרי  $\underline{\text{relop}}$ ). כדי שהнтוחות התחבירי יסתוים בהצלחה צריך לצמצם את המחרוזת  $w$  ב-  $\underline{\text{relop}}$   $\alpha$  לשנה אחרת ( $S$  אבל זה בלתי אפשרי כי אילו זה היה אפשרי אז  $\underline{\text{relop}}$  היה של תבנית פסוקית), המחרוזת  $w \underline{\text{relop}}$   $\alpha$  הייתה תבנית פסוקית ואז  $\underline{\text{relop}}$  היה ב- (B) FOLLOW (כי יש תבנית פסוקית שבה הוא מופיע מיד אחרי (B) וזה לא נכון).

הסבר קצר יותר: צימצום ל- B משמעותו של "ראיינו B" בקלט (כלומר ראיינו ביטוי בוליאני). אין טעם לעשות זאת כאשר סימן הקלט הבא הוא  $\underline{\text{relop}}$  כי  $\underline{\text{relop}}$  לא יכול לבוא מיד אחרי ביטוי בוליאני (כי  $\underline{\text{relop}}$  לא ב- (B) FOLLOW).

### קונפליקטים

יש שני סוגים אפשריים של קונפליקטים:

קונפליקט shift reduce זה קורה כאשר במצב  $\alpha$  של האוטומט מופיעים

שני פריטים כאלה :

$$\begin{aligned} A &\rightarrow \alpha \bullet a \beta \\ B &\rightarrow \gamma \bullet \end{aligned}$$

ומלבד זאת מתקיים ש-  $a$  שייך ל- FOLLOW(B).

במקרה כזה הפריט הראשון אומר שבמצב  $\bullet$ , כאשר סימן הקלט הבא הוא  $a$ , צריך לעשות לו shift. הפריט השני אומר שבמצב  $\bullet$ , כאשר סימן הקלט הבא הוא  $a$ , צריך לעשות reduce לפי  $\gamma \rightarrow B$ .

קונפליקט reduce reduce זה קורה כאשר במצב  $\bullet$  של האוטומט מופיעים שני פריטים כאלה :

$$\begin{aligned} A &\rightarrow \alpha \bullet \\ B &\rightarrow \beta \bullet \end{aligned}$$

ומלבד זאת החיתוך של FOLLOW(A) עם FOLLOW(B) אינו ריק.  
(נניח ש-  $a$  נמצא בשנייהם).

במקרה כזה, הפריט הראשון אומר שבמצב  $\bullet$ , כאשר סימן הקלט הבא הוא  $a$ , צריך לעשות reduce לפי  $A \rightarrow \alpha$ . הפריט השני אומר שבמצב  $\bullet$ , כאשר סימן הקלט הבא הוא  $a$ , צריך לעשות reduce לפי  $\beta \rightarrow B$ .

הערה : אין הכרח ש-  $\alpha$  תהיה זהה ל-  $\beta$  אבל בכל מקרה אחת מהן תהיה סיפא של השניה. זה נובע מכך שלפי הפריט הראשון אנו יודעים שבראש המחסנית מופיעה המחרוזת  $\alpha$  ולפי הפריט השני אנו יודעים שבראש המחסנית מופיעה המחרוזת  $\beta$ .  
לדוגמא יתכן ש-  $\alpha$  היא bcD ו-  $\beta$  היא cD (או D או bcD).

אין קונפליקט מסווג shift shift שני פריטים כאלה :

$$\begin{aligned} A &\rightarrow \alpha \bullet a \beta \\ B &\rightarrow \alpha_2 \bullet a \beta_2 \end{aligned}$$

במקרה כזה, שני הפריטים אמורים שבמקרה שסימן הקלט הבא הוא  $a$ , יש לעשות shift לאותו המציב, כלומר ל-  $i$  (goto  $i$ , a) (זה המציב שיכלול את הפריטים  $\alpha$  ו-  $\beta$  [  $A \rightarrow \alpha \bullet a \rightarrow \alpha_2 a \rightarrow \beta_2$  ] ). לכן אין כאן קונפליקט.  
מצב 2 בדוגמה שהובאה לעיל (דוגמה לטבלת SLR) היא דוגמא למצב המתואר כאן (במצב 2 הטרמינל a הוא i).

כל דקדוק רב משמעי אינו SLR. בשפה המוגדרת על ידי דקדוק רב משמעי יש לפחות מילה אחת שניתן לגוזר אותה בשתי צורות שונות. כאשר המנתח בשיטת SLR ינתח מילה זו והוא יגיע בשלב מסוים למצב בו תהיה לו בחירה בין שתי פעולות שונות (יהיה קונפליקט). בחירה באחת הפעולות תתאים לגזירה מסוימת של המילה ובבחירה ב פעולה אחרת תתאים לגזירה אחרת.  
(יש דוגמא לכך בהמשך).  
יש גם דקדוקים חד משמעיים שאינם SLR (1).

לעתים (אבל לא תמיד) ניתן "לפתח קונפליקט" באופן ידני כולם במקרה שהאלגוריתם לבנית הטעלה מניב כניסה עם יותר פעולה אחת ניתן לבחור באחת מהן ולמוחק את השאר.

דוגמה לפיתרון של קונפליקט  
נתבונן בבדיקה (הרב משמעי) הבא:

$$\begin{aligned} E' &\rightarrow E \\ E &\rightarrow E + E \mid E * E \mid (E) \mid \underline{id} \end{aligned}$$

כאשר בונים את האוטומט עבור הבדיקה הזאת מקבלים בין היתר מצב הכלול את הפריטים הבאים:

$$\begin{aligned} E &\rightarrow E + E \bullet \\ E &\rightarrow E \bullet + E \\ E &\rightarrow E \bullet * E \end{aligned}$$

כאשר נמצאים במצב האמור וסימן הקלט הבא הוא \* יש קונפליקט:

לפי הפריט הראשון צריך לצמצם לפי הכלל  $E \rightarrow E + E$  (FOLLOW(\*)) נמצא ב- (E). לפי הפריט השלישי צריך לעשות shift. (הפריט השני אינו רלבנטי כאשר סימן הקלט הבא הוא \*).

נראה דוגמא שבה הקונפליקט בא לידי ביטוי. עבור הקלט  $id * id + id$  מתקבל תחבירי יעשה את הצעדים הבאים (נראה רק את סימני הדיקזוק על המחסנית ולא את מצבים האוטומטיים).

תוכן המחסנית	שארית הקלט	פעולה שבוצעה
$id$	$id + id * id \$$	
$E$	$+ id * id \$$	shift
$E +$	$+ id * id \$$	reduce $E \rightarrow id$
$E + id$	$id * id \$$	shift
$E + E$	$* id \$$	shift
	$* id \$$	reduce $E \rightarrow id$

עכשו המנתח נמצא במצב הנ"ל והקונפליקט בא לידי ביטוי. אם המנתח יעשה shift ההמשך יהיה:

תוכן המחסנית	שארית הקלט	פעולה שבוצעה
$E + E *$	$id \$$	shift
$E + E * id$	$\$$	shift
$E + E * E$	$\$$	reduce $E \rightarrow id$
$E + E$	$\$$	reduce $E \rightarrow E * E$
$E$	$\$$	reduce $E \rightarrow E + E$
$E$	$\$$	accept

כאנו הקלט נתחaal כתוב  $(id * id) + id$  כלומר פעולה הכפל מתבצעת לפני פעולה החיבור וזה בסדר כי מקובל שכפל יש עדיפות גבוהה מלחיבור.

מה קורה אם המנתה בוחר לצמצם כאשר מתעורר הקונפליקט ? ההמשך יהיה :

תוכן המקסנית	שארית הקלט	פעולה שבוצעה
E	* id \$	$E \rightarrow E + E$
E *	id \$	shift
E * id	\$	shift
E * E	\$	reduce $E \rightarrow id$
E	\$	reduce $E \rightarrow E * E$
E	\$	accept

כאן הקלט נתחזק אליו היה כתוב  $id * id$  (  $id + id$  ) כולם החיבור מתבצע לפני הכפל ולכן זה לא טוב . הפיתרון של הקונפליקט הוא אם כן לעשות shift.

באוטו המצב יש גם קונפליקט כאשר סימן הקלט הבא הוא + . הפריט הראשון אומר לעשות reduce (+ נמצא ב- (E) FOLLOW (E)). הפריט השני אומר לעשות shift . איך יתבטא הקונפליקט כאשר מילת הקלט היא  $id + id + ?$  כמו בדוגמה הקודמת גם כאן הקונפליקט יבוא לידי ביטוי כאשר במחסנית יש  $E + E$  (והפעם סימן הקלט הבא הוא +). המשמעות של לעשות reduce היא שונת את המילה אליו הייתה כתובה  $id + id + id$  . המשמעות של shift היא  $(id + id + id) + id$  . מקובל להעדיף את האפשרות הראשונה (זה אומר שלופרטור + יש אסוציאטיביות שמאלית) ולכן פיתרון הקונפליקט הזה הוא reduce .

## שיטות שונות לבניית טבלת LR parsing

ישנן שלוש שיטות שונות לבניית טבלת LR parsing. הטבלה המתבקשת נקראת טבלת (1) SLR, טבלת (1) LR (קונונית) או טבלת (1) LALR. אם בטבלת (1) SLR אין קונפליקטים, הדקדוק הוא (1) LR. אם בטבלת (1) LALR הקונונית אין קונפליקטים, הדקדוק הוא (1) SLR. אם בטבלת (1) LR הקונונית אין קונפליקטים, הדקדוק הוא (1) LALR.

### טבלת (1) SLR

השיטה הפשוטה ביותר משתמש באוטומט פריטי (0) LR. פריט מהצורה  $\alpha \rightarrow A$  המופיע בקובץ הפריטים, משמעתו: כאשר האוטומט במצב  $i$  והסימן הבא בקלט (ה-lookahead) שיך ל- FOLLOW ( $A$ ) אז יש לבצע reduce לפי  $\alpha \rightarrow A$ . כאשר בונים את הטבלה בשיטה זו אין בה קונפליקטים, הדקדוק הוא (1) SLR. הערכה: האוטומט נקרא אוטומט פריטי (0) LR כי לפריטים אין lookahead (ראו בהמשך) אבל הטבלה היא טבלת (1) SLR כי יש לדעת מה הה-lookahead כדי להשתמש בטבלה.

### טבלת (1) LR קונונית

לצורך בניית טבלה זו יש לבנות אוטומט פריטי (1) LR. המצביעים של האוטומט הם קבוצות של פריטי (1) LR. פריט (1) LR הוא פריט (0) LR בתוספת lookahead.

פריט (1) LR מצוין כך:  $[a, \gamma \rightarrow \beta]$  (לעתים נשמיט את הסוגרים המרובעות). כאן  $a$  הוא ה-lookahead והוא עשוי להיות טרמינל או  $\$$ . המשמעות של פריט זהה: ראיינו בקלט  $\beta$ , אנו מצלפים לראות את  $\gamma$  בהמשך הקלט ולאחריו צפוי להימצא  $a$  בקלט. אם אכן כך יקרה אז נגיע אליו מספר צעדים למצב הכלל את הפריט  $a \rightarrow \beta$ . באותו שלב, אם אכן ה-lookahead  $\gamma$  יהיה  $a$  אז נבצע reduce לפי הכלל  $\gamma \rightarrow \beta$ .

החידוש כאן הוא שההחלטה מתי לבצע reduce היא מדויקת יותר:

פריט מהצורה  $a \rightarrow \alpha$  משמעתו שיש לבצע reduce לפי  $\alpha \rightarrow A$  כאשר ה-lookahead הוא  $a$ . באוטומט פריטי (0) LR לעומת זאת, יופיע פריט  $\alpha \rightarrow A$  משמעתו שיש לבצע reduce לפי  $\alpha \rightarrow A$  כאשר ה-lookahead שיך ל- FOLLOW ( $A$ ). תמיד יכול גם את  $a$  אבל הוא עשוי להכיל גם איברים נוספים למשל  $b$ .

העובדת שבטבלת ה- (1) SLR (במצב הרלוונטי) ירשם שיש לבצע reduce לפי  $\alpha \rightarrow A$  כאשר ה- lookahead הוא  $b$  ובטבלת ה- (1) LR הכנוגנית זה לא ירשם עשויה להיות קריטית: יתכן שהמנתח אמור לעשות פעולה אחרת באותו מצב כאשר ה- lookahead הוא  $b$ . במקרה כזה, בטבלת ה- (1) SLR נקלט קונפליקט כשה- lookahead הוא  $b$  ובטבלת ה- (1) LR הkonfliket ימנע.

### דוגמא

נתבונן בדקדוק הבא :

$$(1) S \rightarrow a c$$

$$(2) S \rightarrow A b A c$$

$$(3) A \rightarrow a$$

דקדוק זה הוא LR (1) אך אינו SLR (1).

ראו דף מצורף (קובץ LR1\_but\_not\_SLR1.pdf).

הבעיה בטבלת ה- (1) SLR נוצרת במצב הכלל את הפריטים  $S \rightarrow a \bullet$  ו-  $S \rightarrow a \bullet c$  lookahead על המנתח התחבירי לעשות reduce לפי  $a \rightarrow A$  כאשר ה- lookahead שוייך ל- FOLLOW(A) = {b, c}. לכן כשה- lookahead הוא  $c$  השיביך ל- FOLLOW(A). בדקדוק זה מנטח  $c$  על מנת לבצע reduce  $a \rightarrow a$ . אבל לפי הפריט  $A \rightarrow a \bullet c$  הוא צריך לעשות reduce לפי  $a \rightarrow a$ . ואכן יש shift/reduce conflict כאשר יש  $c$  בקלט וכך קיבלנו shift.

לעומת זאת, המנתח התחבירי שמשתמש בטבלת (1) LR מבין שבקשר הזה, אחרי ה-  $A \rightarrow a \bullet$  בקלט (לייטר דיק בקלט מופיעות מהירות מ-  $A$  אותה מצבם המנתח ל-  $A$ ) אמור להופיע  $b$  ולא  $c$ . זה מותbeta באפריט  $b, a \rightarrow A$  המופיע במצב 2. לכן אין כאן קונפליקט: במצב 2, כשماופיע  $b$  בקלט יש לעשות reduce לפי  $a \rightarrow A$  וכשהוא מופיע  $c$  בקלט עושים shift.

שים לב שהפריט  $b, a \rightarrow A$  במצב 2 מקורו ב-  $A$  הראשון בפריט  $S \rightarrow \bullet A b A c, \$$  המופיע במצב 0. אחרי ה-  $A$  הזה אמור להופיע  $b$ . נכון ש- FOLLOW(A) כולל גם את  $c$  אבל בהקשר שבו מדובר, לא אמור להופיע  $c$  אחרי ה-  $A$ . העובדת הזאת מתbetaת

בטבלת ה- (1) LR הקונונית אבל לא בטבלת ה- (1) SLR : לפי זו האחרונה, גם ס

משמעותו בקלט מהווע סיבה מספקת לעשות reduce ל- A.

### בנייה אוטומט פריטי (1)

מראים את הדקדוק עם הכלל  $S \rightarrow S'$ . המצב ההתחלתי של האוטומט הוא

$\{ S' \rightarrow \bullet S, \$ \}$ .

מעברים בין מצבים : נקרא לפונקציית המעברים של האוטומט GOTO כלום :

$J = (I, X)$  פירושו שיש מעבר המסומן ב- X (שיכול להיות טרמינל או משתנה) ממצב I למצב J. שימושו לב שאוטומט הוא דטרמיניסטי.

או :

$$GOTO(I, X) = CLOSURE(\{ [A \rightarrow \alpha \cdot X \bullet \beta, a] \mid [A \rightarrow \alpha \bullet X \beta, a] \in I \})$$

חישוב הסגור של קבוצת פריטים : כל מצב מתקיים כסוגר של קבוצת פריטים אבל חישוב הסגור צריך לנקות בחשבו גם את ה- lookahead. אם בקבוצת הפריטים קיים פריט מהצורה

$a \beta \bullet A \rightarrow B$  (כרגיל מצין משתנה) אז עברו כל כלל גזירה מהצורה  $\gamma \rightarrow B$  (כלומר עברו כל כלל גזירה של B) ועברו כל איבר b השיך לו. -- נסיף לסגור את הפריט

$$. B \rightarrow \bullet \gamma, b$$

דוגמא : ראו דוגמא לאוטומט פריטי (1) LR המופיע בקובץ LR(1)\_items\_automaton.pdf. האוטומט מתאים לדקדוק הבא :

(1)  $S \rightarrow a$

(2)  $S \rightarrow \text{begin SQ end}$

(3)  $SQ \rightarrow SQ S$

(4)  $SQ \rightarrow \epsilon$

הדקוק מותאר שפת תכנות מאוד פשוטה. S מיצג משפטי של השפה ו- SQ מיצג סדרות של משפטיים.

הערה : כאשר במצב מסוים מופיעים מספר פריטים השונים זה מזה רק ב- lookahead

אז ניתן לרשום אותם בדרך מקוצרת. למשל במצב 4 מופיע

$$S \rightarrow SQ \bullet S, a/b/e$$

זה כתיב מקוצר לשלוות הפריטים הבאים:

$$S \rightarrow SQ \bullet S, a$$

$$S \rightarrow SQ \bullet S, b$$

$$S \rightarrow SQ \bullet S, e$$

(b) מופיע בציור האוטומט כקיצור של הטרמינל begin ו- e זה קיצור של הטרמינל end

.  $[S \rightarrow \bullet \text{begin } SQ \text{ end}, \$]$

לכן יש מעבר ממצב 0 המסומן begin אל המצב  
מצב 3 בציור.

כיצד מחושב הסגור? יש להוסיף את כל הפריטים מהצורה

$$SQ \rightarrow \bullet \beta, d$$

$$SQ \rightarrow \bullet, end$$

$$SQ \rightarrow \bullet, end -> SQ \rightarrow \bullet SQ S, end$$

בגלל שהוספנו את הפריט  $SQ \rightarrow \bullet SQ S, end$  יש להוסיף גם את כל הפריטים מהצורה

$$SQ \rightarrow \bullet \beta, d$$

לכן נוסיף גם את ארבעת הפריטים הבאים: FIRST (S end) = { a, begin }

$$SQ \rightarrow \bullet SQ S, a/begin$$

$$SQ \rightarrow \bullet, a/begin$$

כך קיבלנו בסך הכל:

$$S \rightarrow \text{begin} \bullet SQ end, \$$$

$$SQ \rightarrow \bullet SQ S, a/begin/end$$

$$SQ \rightarrow \bullet, a/begin/end$$

### בנייה טבלת (1) LR הקנוני לפי אוטומט פריטי (1).

הבנייה מאוד דומה לבניה של טבלת (1) SLR מAUTOMAT FRITI (0).

פריט מהצורה  $A \rightarrow \alpha \bullet a \beta, b$  משמעו שיש לבצע shift (ולעבור למצב המתאים) כשה-lookahead הוא a. (a הוא טרמינל).

פריט מהצורה  $a \rightarrow A \bullet, a \rightarrow A$  פרשו שיש לבצע reduce לפי הכלל  $\alpha \rightarrow A$  כאשר ה- lookahead הוא  $a$ .

הפריט  $\$ \rightarrow S \bullet, \$ \rightarrow S'$  פרשו שיש לעשות accept כאשר ה- lookahead הוא  $\$$  (במקום לעשות reduce לפי  $S' \rightarrow S$ ).

טבלת ה-goto נבנית לפי המעברים המסומנים במשתנים (בדומה לטבלת (SLR (1)).

בדוגמא שלנו מתקבלת טבלת LR (1) קונקטיבית הבאה:

	a	begin	end	\$	S	SQ
0	s2	s3			1	
1				accept		
2				r1		
3	r4	r4	r4			4
4	s7	s8	s5		6	
5				r2		
6	r3	r3	r3			
7	r1	r1	r1			
8	r4	r4	r4			9
9	s7	s8	s10		6	
10	r2	r2	r2			

אין כאן קונפליקטים ולכן הבדיקה הוא (1).

### טבלת LALR (1)

כשעובדים עם אוטומט פריטי (1) LR ניתן לעיתים קרובות להמנע מkonפליקטים שהיו נוצרו LR parsing part 2 5

באוטומט פריטי (0) LR . החסרון הוא שבעור דקדוק נתון, לעיתים קרובות אוטומט פריטי (1) LR יוכל יותר מצבים מאשר אוטומט פריטי (0) LR . נתבונן לדוגמה באוטומט פריטי (1) LR הניל.

המצבים 2 ו- 7 זהים מלבד הבדלים ברכיבי ה- lookahead :

$$\text{מצב 2 : } S \rightarrow a \bullet, \$$$

$$\text{מצב 7 : } S \rightarrow a \bullet, a/\text{begin/end}$$

לשני הממצבים יש core זהה (ה-core הוא קבוצת הפריטים ללא רכיבי ה- lookahead).

באוטומט פריטי (0) LR עברו אותו הדקדוק מופיע רק מצב אחד במקומות מצבים 2 ו- 7.

$$\text{מצב זה כולל את הפריט } \bullet \rightarrow a S$$

בגלל התופעה זו של "פייצול" מצב בודד באוטומט פריטי (0) LR למספר מצבים באוטומט

פריטי (1) LR שכולם בעלי core זהה -- מספר הממצבים באוטומט פריטי (1) LR תמיד

גדול או שווה למספר הממצבים באוטומט פריטי (0) LR עברו אותו הדקדוק.

באופן מעשי מספר הממצבים אכן יהיה גדול יותר כשמדבר בדקדוקים של שפות תכנות כי

התופעה של מבנה תחבירי שיכל להופיע בהקשרים שונים היא נפוצה.

למשל בדקדוק שראינו, משפטים (המיוצגים ע"י המשטנה S)

עשויים להופיע בשני הקשרים שונים : הם עשויים להופיע כמשפטים ב- top level

ואז אחרים אמרו להופיע \$ (סוף הקלט) והם עשויים להופיע כחלק מסדרת משפטיים מוקפים ב- begin ו- end . במקרה זה אחרי המשפט אמרו להופיע end (אם זה המשפט האחרון בסדרה) או

a או begin (במקרה שאחרי המשפט בא משפט נוסף). שני הקשרים השונים בהם מופיעים

משפטים מתבאים בהבדלים בין רכיבי ה- lookahead במצבים 2 ו- 7.

הרעיוון של טבלת LALR : נאחד ממצבים באוטומט פריטי (1) LR בעלי core זהה וכן נקטין את מספר

المמצבים שיושווה למספר הממצבים באוטומט פריטי (0) LR ולכן יושווה למספר הממצבים בטבלת (1) SLR.

יחד עם זאת, לאחר שימוש בפריטי (1) LR שכפי שראינו "מידיקים" יותר (לעומת פריטי (0) LR) יש שאלת מתי יש לבצע -- לעיתים נצלח להמנע מקונפליקטים שהיו מתגלים אילו השתמשנו

בפריטי (0) LR .

### אלגוריתם פשוט לבנית טבלת (1) LALR :

בנה אוטומט פריטי (1) LR.

אחד מצבים בעלי core זהה.

בנה את הטבלה לפי האוטומט שקיים. (בדיקה כמו שבונם את טבלת ה- (1) LR הקיימת).

איחוד מצבים : הכוונה היא שהמצב המאוחד כולל את כל הפריטים של המצבים מהם אחד. לדוגמה, באוטומט שריאנו, מצבים 2 ו- 7 יאוחדו למצב אחד בעל ארבעת הפריטים

$S \rightarrow a \bullet, a/\text{begin/end}/\$$

מצבים 3 ו- 8 יאוחדו למצב אחד הכלל את הפריטים :

$S \rightarrow \text{begin } \bullet \text{ SQ end, a/begin/end}/\$$

$\text{SQ} \rightarrow \text{SQ S, a, begin/end}$

$\text{SQ} \rightarrow \bullet, a/\text{begin/end}$

כך יאוחדו גם מצבים 4 ו- 9 ומצבים 5 ו- 10. שימוש לב שבאופן כללי עשויים להיות יותר מאשר שני מצבים

בעלי core זהה אם כי זה לא קורה בדוגמה זו. ניתן גם שלא יהיה בכלל מצבים בעלי core זהה.

במקרה כזה לא יאוחדו מצבים, האוטומט ישאר ללא שניי וטבלת (1) LALR תהיה זהה לטבלת (1) LR.

מה יהיו המעברים של האוטומט המתקבל מאיחוד המצבים? נניח שמאחדים את מצב I עם מצב J (אוily עם מצבים נוספים) ונקרא למצב המאוחד K.

עבור סימן דקוק X נגידר ש- (X, K) GOTO הוא המצב המתקבל מאיחוד (X, I) GOTO עם (X, J) GOTO (אוily גם עם מצבים נוספים בעלי אותו core). שימוש לב שם ל- I ול- J יש את אותו core אז גם ל- (X, I) GOTO ול- (X, J) GOTO יהיה את אותו core (אם כי ניתן שאינם שוויים). לכן שני האחרונים אכן יאוחדו.

הערה : אם אין ל- I ול- J מעבר המסומן ב- X אז גם באוטומט החדש לא יהיה מעבר המסומן ב- X מהמצב המאוחד.

דוגמה : באוטומט פריטי (0) LR עבור הדקוק הניל, יש מעבר ממצב 3 למצב 4 המסומן ב- SQ

ויש מעבר המסומן ב- SQ ממצב 8 למצב 9. באוטומט המתקבל לאחר איחוד המצבים יש

מעבר המסמן ב- SQ מ מצב 3,8 (המצב המתקבל מאיחוד מצבים 3 ו- 8) למצב 9,4.

האוטומט המתקבל לאחר איחוד המצבים מופיע בקובץ LALRautomaton.pdf  
הנה טבלת (1) של הדקוק.

	a	begin	end	\$	S	SQ
0	s 2,7	s 3,8			1	
1				accept		
2,7	r1	r1	r1	r1		
3,8	r4	r4	r4			4,9
4,9	s 2,7	s 3,8	s 5,10		6	
5,10	r2	r2	r2	r2		
6	r3	r3	r3			

אין קונפליקטים בטבלה ולכן הדקוק הוא (1) LALR. הטבלה כוללת 7 מצבים לעומת טבלת LR (1) עבור אותו הדקוק שכוללת 11 מצבים. בטבלה (1) SLR עבר הדקוק יופיע 7 מצבים -- כל מצב באוטומט שהתקבל לאחר איחוד המצבים מותאים למצב אחד באוטומט פריטי (0) LR : קבוצת הפריטים במצב של אוטומט פריטי (0) LR זהה ל- core המשותף של המצבים שאוחדו.

לדוגמה באוטומט פריטי (1) LR הופיעו שני המצבים :

מצב 2 :  $S \rightarrow a \bullet, \$$

מצב 7 :  $S \rightarrow a \bullet, a/\text{begin/end}$

באוטומט המתקבל מאיחוד המצבים מופיע מצב 2,7 שהוא איחוד של שני המצבים :

$S \rightarrow a \bullet, a/\text{begin/end}/\$$

באוטומט פריטי (0) LR המצביע המתאים יכול את הפריט  $S \rightarrow a \bullet$

זו תופעה כללית: עבור דקדוק כלשהו, מספר המצבים בטבלת (1) SLR יהיה שווה למספר המצבים בטבלת (1) LALR ומספר זה יהיה קטן או שווה למספר המצבים בטבלת (1) LR קנונית.

### התנהגות דומה של מנתח (1) LR ומנתח (1) LALR

שני מנתחים תחביריים, האחד שפועל עם טבלת (1) LR קנונית והשני שפועל עם טבלת (1) LALR (עבור אותו דקדוק) יתנהגו בצורה דומה כאשר יופעלו על אותו קלט חוקי (כלומר קלט שנitin לגזירה בדקדוק).

בכל שלב של הנитוח התחבירי שניהם יעשו shift או שנייהם יעשו reduce לפי אותו הכלל. זה נכון כי שניהם "בונים" את אותו עץ גזירה. בנוסף לכך בכל שלב של פעולות, הם ימצאו במצבים מתאימים: כאשר מנתח ה- (1) LR נמצא במצב  $s$  אז מנתח ה- (1) LALR ימצא במצב שהתקבל מאחד  $s$  עם המצבים בעלי אותו core (אם ישם כאלו).

כאשר המנתחים יופעלו על קלט לא חוקי, שניהם יגלו את השגיאה כשיגיעו לאותו מקום בקלט אבל יתכן שמנתח ה- (1) LALR יגלה את השגיאה רק לאחר שיושה מספר פעולות reduce נוספת. במקרה, מנתח ה- (1) LALR יגלה את השגיאה מבלי שיושה פעולה shift נוספת.

דוגמה: נראה את פעולותם של שני המנתחים התחביריים הפורלים לפי הדקדוק הניל על הקלט השגוי (השגיאה היא שחרר end בסוף). (המחסנית לאמתיו של דבר כוללת רק מצבים אבל נוח להראות עלייה גם את סימני הדקדוק המתאים)

LR (1) parser stack	LALR (1) parser stack	remaining input	action
0	0	begin a a \$	
0 begin 3	0 begin (3,8)	a a \$	shift
0 begin 3 SQ 4	0 begin (3,8) SQ (4,9)	a a \$	reduce by $SQ \rightarrow \epsilon$
0 begin 3 SQ 4 a 7	0 begin (3,8) SQ (4,9) a (2,7)	a \$	shift

0 begin 3 SQ 4 S 6	0 begin (3,8) SQ (4,9) S 6	a \$	reduce by $S \rightarrow a$
0 begin 3 SQ 4	0 begin (3,8) SQ (4,9)	a \$	reduce by $SQ \rightarrow SQ S$
0 begin 3 SQ 4 a 7	0 begin (3,8) SQ (4,9) a (2,7)	\$	shift
	0 begin (3,8) SQ (4,9) S 6	\$	LR (1) parser detects error. LALR (1) parser reduces by $S \rightarrow a$
			now LALR (1) parser detects the error

#### יחסים הכללים בין משפחות של דקדוקים

כל דקדוק שהוא (1) LALR הוא גם (1) LR : אם אין קונפליקטים לאחר איחוד המצבים אז ברור שגם אין קונפליקטים באוטומט פריטי (1) LR לפני איחוד המצבים.

אבל ניתן שדקדוק יהיה (1) LR אבל לא (1) LALR : כאשר מתחדים מצבים עלולים להיווצר קונפליקטים מסווג reduce/reduce.

לדוגמא אם באוטומט פריטי (0) LR יש שני מצבים, האחד כולל את הפריטים  $B \rightarrow a \bullet, a$  והשני כולל את הפריטים  $B \rightarrow a \bullet, b$  ו-  $A \rightarrow a \bullet, b$  ו-  $A \rightarrow a \bullet, a$  אז בעקבות איחוד שני המצבים במצב הכלול את הפריטים נקבל שני קונפליקטים חדשים מסווג reduce/reduce כאשר ה lookahead הוא a יש לבצע reduce לפיה  $A \rightarrow a$  וגם reduce לפיה  $B \rightarrow a$  וכך גם כאשר

שימושו לב שאיחוד מצבים אינו יכול לגרום להיווצרות קונפליקטים חדשים מסווג shift/reduce לדוגמא נניח שלאחר איחוד מצבים מקבלים מצב שככל (בין היתר) את שני הפריטים  $B \rightarrow a \bullet, b$  ו-  $A \rightarrow a \bullet, b$ . שני הפריטים האלה מניבים קונפליקט מסווג shift/reduce LR parsing part 2 10

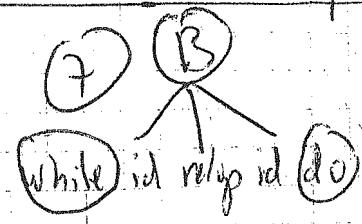
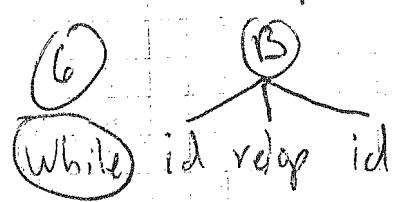
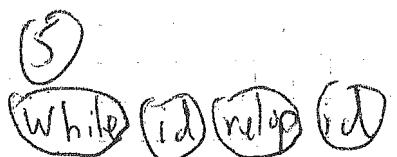
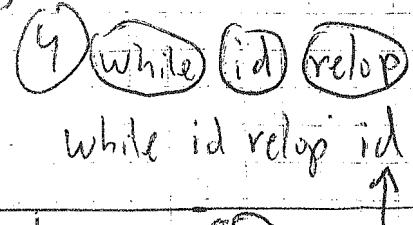
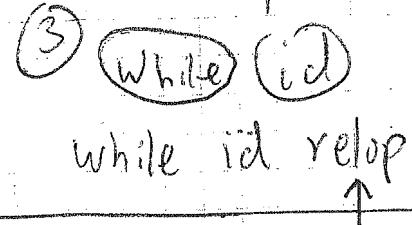
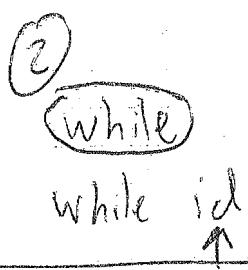
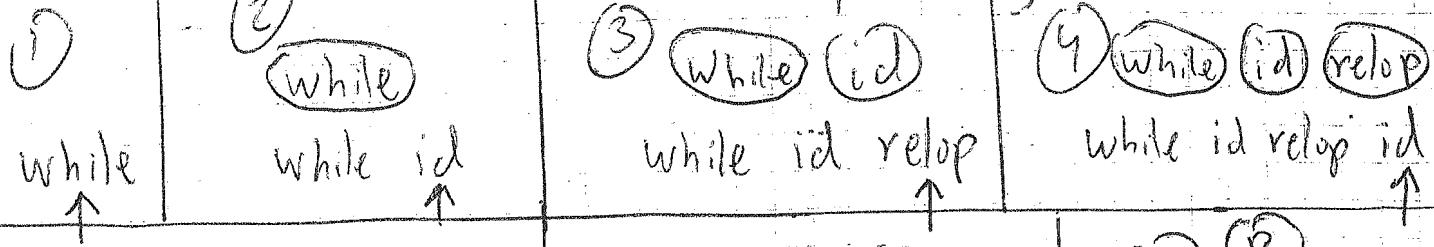
cash-lookahead הוא b. אבל קונפליקט זה היה קיים גם לפני שմצבים אוחדו. אחד המצבים אוחדו כלל את הפריט  $b \rightarrow a \bullet$ , ואותו מצב כלל גם פריט מהצורה  $A \rightarrow a \bullet b, d$  כאשר d הוא טרמינל או  $\$$  (או דזוקא c -- יתכן שהפריט  $c \rightarrow a \bullet b$  מקורו במצב אחר). מצב זה כלל אם כן את הקונפליקט האמור.

ראינו אם כן שקבוצת דקדוקי (1) LR מכילה ממש את קבוצת דקדוקי (1) LALR.

קבוצת דקדוקי (1) LALR מכילה ממש את קבוצת דקדוקי (1) SLR. כאמור, לעיתים קונפליקטים נמנעים בগল ספריט (1) LR "מדייק" יותר בשאלת מתי יש לעשות reduce. لكن דקדוק שאינו (1) SLR עשוי להיות (1) LR ולעיתים גם (1) LALR.

נתנו להוכיח שמשפחה דקדוקי (1) LR מכילה ממש את משפחת דקדוקי (1) LL. באופן אינטואיטיבי, לשוטטו של מנתה (1) LR עומד מידע רב יותר מאשר מנתה (1) LL כאשר הוא צריך להחליט אם להשתמש בכלל  $\alpha \rightarrow A$ : את ההחלטה מקבל מנתה (1) LL אחרי שכבר ראה בקלט את כל מה שנוצר מ-  $\alpha$  ובנוסף לכך ראה את הטרמינל הבא אחורי זה (ה-lookahead) ברגע קבלת ההחלטה לעשות reduce לפי  $\alpha \rightarrow A$ . מנתה (1) LL לעומת זאת צריכה לקלוט החלטה לשימוש בכלל  $\alpha \rightarrow A$  אחרי שראה בקלט רק את הטרמינל הראשון הנוצר מ-  $\alpha$ . לכן יש מקרים שבהם מנתה (1) LL לא יהיה מסוגל לדעת באיזה כלל גזירה להשתמש בעוד שמןתח (1) LR כן ידע.

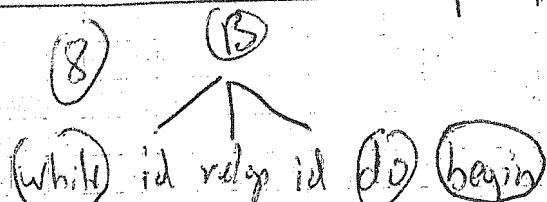
# LR parsing



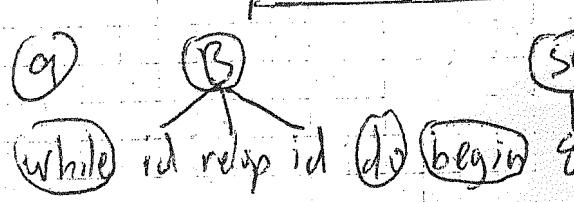
white id relop id do

white id relop id do

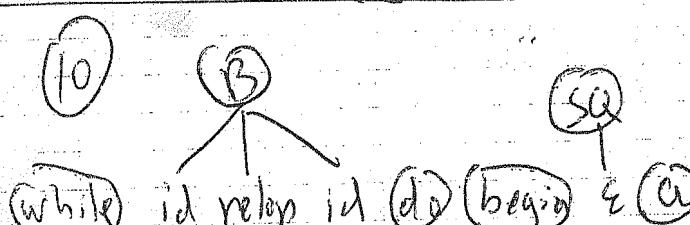
white id relop id do begin



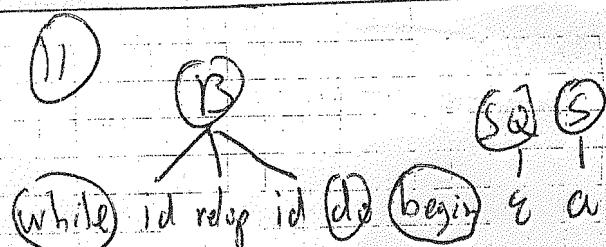
white id relop id do begin a



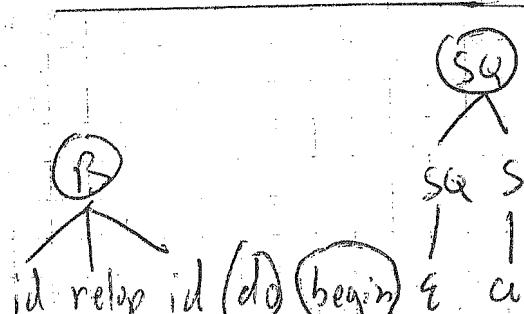
white id relop id do begin a



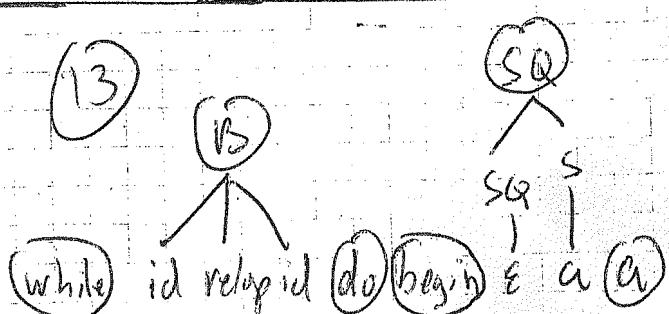
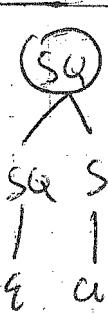
relop id do begin a a



white id relop id do begin a a

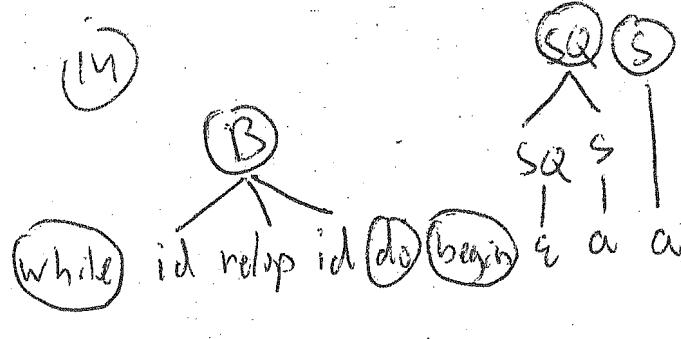


id relop id do begin a a

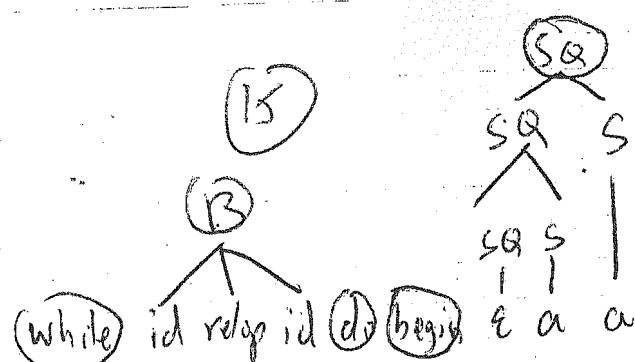


white id relop id do begin a a end

→ SQ S  
a a

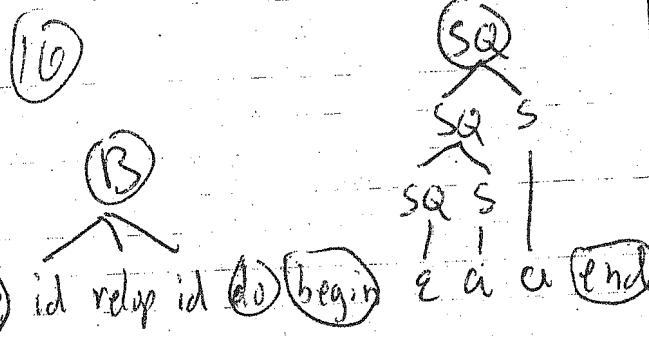


while id relop id do begin a a end

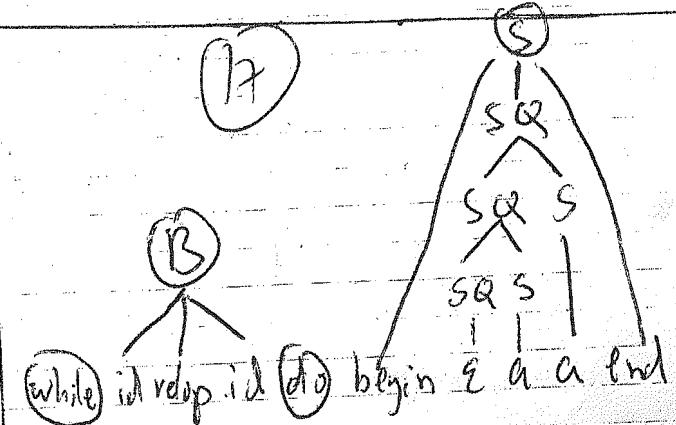


white id relop id do begin a a

white id relop id do begin a a end

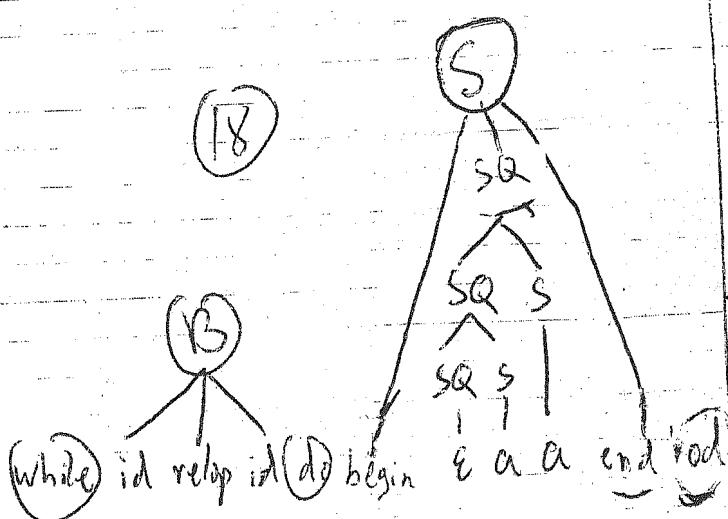


while id relop id do begin a a end od



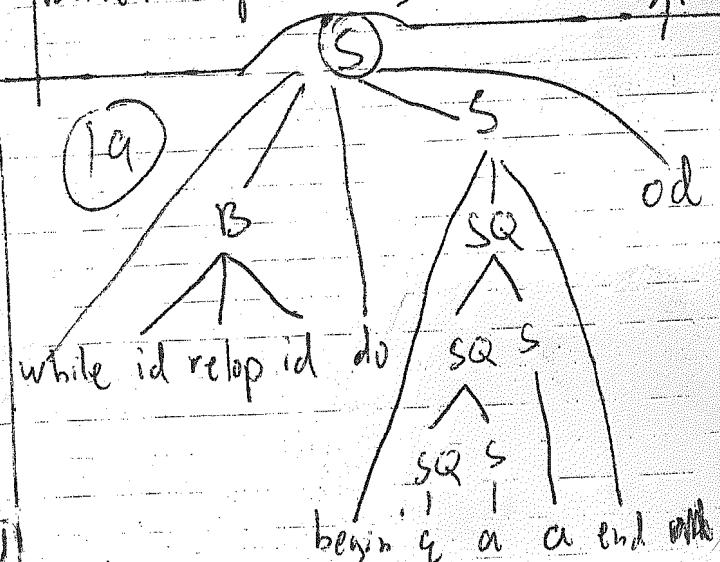
white id relop id do begin a a end od

white id relop id do begin a a end od



white id relop id do begin a a end od

white id relop id do begin a a end od



white id relop id do

begin a a end od

white id relop id do begin a a end od

1

S-a

S → begin SQ end

$SQ \rightarrow SQ : S$

$SQ \rightarrow e$

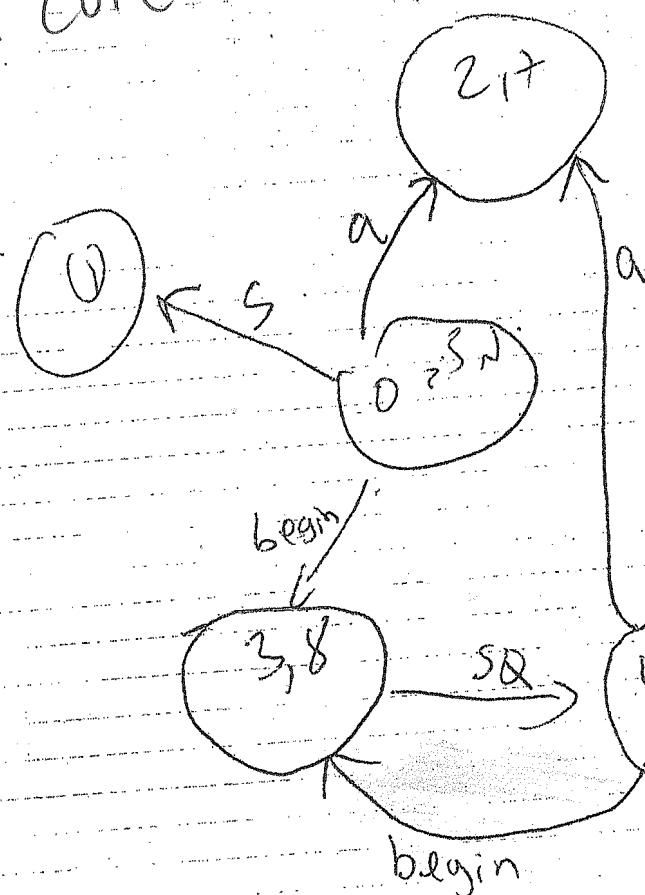
older core

Enigmo

三九

3)  $\mu^{\prime \prime} \mu$

162



LAR 100

LHR S1 SQ  
0 a begin end \$1 \$1

1

$\gamma_7 = \gamma_1 = \gamma_2 = \gamma_3$

38 ry m ry

49 52,7 53,8 55,10

5,10 r<sub>2</sub> r<sub>2</sub> r<sub>2</sub> r<sub>2</sub>

$$6 \quad r_3 \quad r_3 \quad m_3$$

49

LAR autor

①  $S \rightarrow a$

②  $S \rightarrow \text{begin } SQ \text{ end}$

③  $SQ \rightarrow SQ S$

④  $SQ \rightarrow \epsilon$

LR(1) (G, 2 done)

①  
 $S \rightarrow S_0 \epsilon$

$\nearrow S$

②  
 $S \rightarrow \epsilon S, \epsilon$

$S \rightarrow \epsilon a, \epsilon$

$S \rightarrow \epsilon \text{ begin } SQ \text{ end}, \epsilon$

$a \nearrow$

$S$

$\text{end} \nearrow$

③

$S \rightarrow \epsilon \text{ begin } SQ \text{ end}, \epsilon$

$\nearrow \text{begin}$

②  
 $S \rightarrow \epsilon \text{ begin } SQ \text{ end}, \epsilon$

$SQ \rightarrow \epsilon SQ S, a/b/e$

$SQ \rightarrow \epsilon, a/b/e$

$SQ \nearrow$

④

$S \rightarrow \epsilon \text{ begin } SQ \text{ end}, \epsilon$

$SQ \rightarrow \epsilon SQ S, a/b/e$

$S \rightarrow \epsilon a, a/b/e$

$S \rightarrow \epsilon \text{ begin } SQ \text{ end}, a/b/e$

$\nearrow \text{begin}$

$a \downarrow$

②  
 $S \rightarrow \epsilon a, a/b/e$

$\nearrow \text{end}$

①  
 $S \rightarrow \epsilon \text{ begin } SQ \text{ end}, a/b/e$

$\nearrow \text{begin}$

⑧  
 $S \rightarrow \epsilon \text{ begin } SQ \text{ end}, a/b/e$

$SQ \rightarrow \epsilon SQ S, a/b/e$

$SQ \rightarrow \epsilon, a/b/e$

$SQ \downarrow$

⑨

$S \rightarrow \epsilon \text{ begin } SQ \text{ end}, a/b/e$

$SQ \rightarrow \epsilon SQ S, a/b/e$

$S \rightarrow \epsilon a, a/b/e$

$S \rightarrow \epsilon \text{ begin } SQ \text{ end}, a/b/e$

$S \rightarrow (6)$

$a \downarrow$

⑦

P13-N P13-N	
2,7	
3,8	
4,9	
5,10	

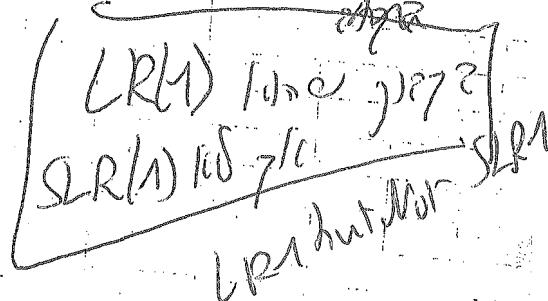
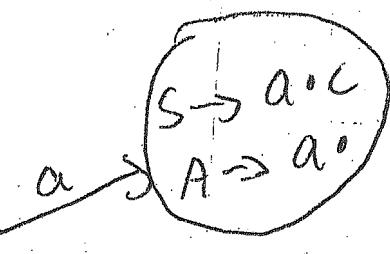
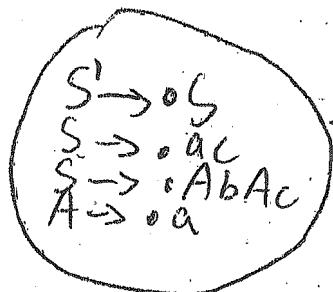
$$① S \rightarrow a.c$$

$$② S \rightarrow A.b.A.c$$

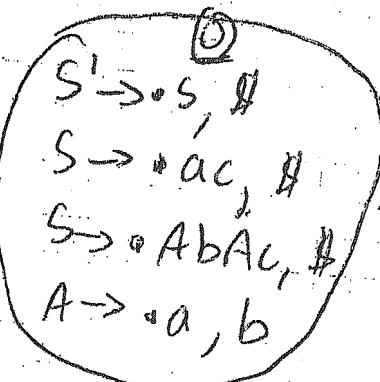
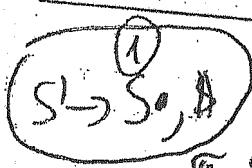
$$③ A \rightarrow a$$

Shift-reduce by LR(0)  
look-ahead c

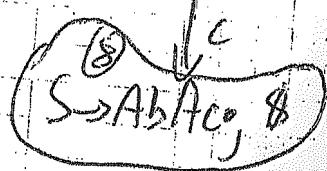
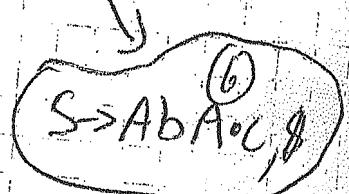
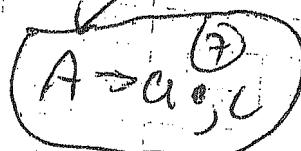
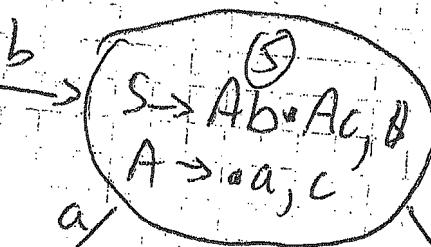
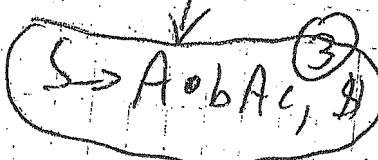
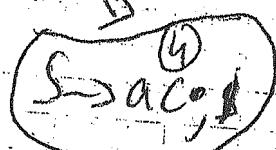
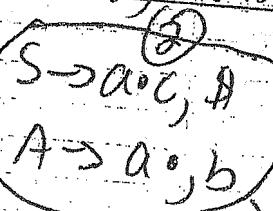
LR(0) (G.1.2 1.1.1c)



LR(1) but for



LR(1) (G.1.2 1.1.1c)



	a	b	c	g	s	A
0	S2					
1						
2						
3						
4						
5						
6						
7						
8						

340

הגדרה מונחת תחביר (syntax directed definition) כוללת דקדוק וכללים סמנטיים (semantic rules) המשויכים לכללי הגזירה. הכללים הסמנטיים מתארים כיצד יש לחשב תכונות (attributes) של משתני (non terminals) הדקדוק. לכל קלט המתאים לדקדוק ניתן לבנות עץ גזירה ואז לעבור על העץ ולחשב את ערכי התכונות בכל צמתיו.

מלבד חישוב של תכונות, הכללים הסמנטיים יכולים גם לבצע דברים נוספים לדוגמה הדפסות, עדכון טבלאות.

בהגדרה מונחת תחביר יכולים להופיע 2 סוגי של תכונות :

תכונה נבנית (נקראת גם תכונה נוצרת). synthesized של משתנה (non terminal) A בצומת N של עץ גזירה מחושבת לפי כלל סמנטי המשויך לכל הгазירה שבו משתמשים ב-N (בכל גזירה זה A מופיע בראש הכלל (משמאלו לחץ) וסימני הדקדוק בילדיהם של N מופיעים בגוף הכלל (מיימין לחץ). המשמעות היא שניתן לשימוש רק בתכונות בילדיהם של N ובתכונות N- עצמה לצורך חישוב תכונה נבנית בצומת N.

לפי הגדרה, תכונה של אסימון (טרמינל) נחשבת תכונה נבנית. היא אינה מחושבת לפי כללים סמנטיים של ה- SDD אלא מסופקת על ידי המנתח הלקסיקלי.

תכונה מורשת (inherited attribute) של משתנה B בצומת N של עץ גזירה מחושבת לפי כלל סמנטי המשויך לכל הгазירה שבו משתמשים באבא של N (בכל גזירה זה סימן הדקדוק באבא מופיע בראש הכלל והסימן B (עם סימנים נוספים במקרה של N יש אחיהם) מופיעים בגוף הכלל). המשמעות היא שניתן לשימוש רק בתכונות באבא של N- N עצמה ובאחיהם של N לצורך חישוב תכונה מורשת בצומת N.

: SDD כותבים

אם למשתנה A יש תכונה נוצרת אז יש לשיך כלל סמנטי לחישוב תכונה זו לכל אחד מכללי הגזירה בהם מופיע A בראש הכלל (כלומר משמאלו לחץ).

אם למשתנה A יש תכונה מורשת אז יש לשיך כלל סמנטי לחישוב תכונה זו לכל אחד מכללי הגזירה בהם מופיע A בגוף הכלל (כלומר מיימין לחץ).

#### סדר חישוב התכונות

הכללים הסמנטיים מגדרים אילוצים על הסדר שבו ניתן לחשב תכונות בצתמים של עץ גזירה.

הסדר צריך להיות כזה שתכונה מסוימת בצומת מסוים תחוسب רק לאחר שהושבו כל התוכנות הנחוצות לצורך חישובה. הנה דרך למציאת סדר לחישוב התוכנות. דרך זו ישימה עבור כל הגדרה מונחת תחבר תקינה (לא מעגלית).

בහינתן עץ גזירה (של קלט מסוים) אותו יש ל凱שט, בונים עבورو **גרף תלויות** (dependency graph). בגרף התלויות יש צומת עבור כל ערך של תוכנה המופיעה בצומת של עץ הגזירה. אם למשל בעץ הגזירה יש צומת מסוים (נקרא לו N) המסומן ב- A, ול- A יש שלוש תוכנות z, A.x, A.y, A.z אז בגרף התלויות יופיעו שלושה צמתים, אחד עבור התוכנה x בצומת N, השניה עבור התוכנה y.A בצומת N והשלישית עבור z.A. אם בעץ הגזירה יש לדוגמה עשרה צמתים המסומנים ב- A אז בגרף התלויות יופיעו שלושים צמתים, שלושה עבור כל אחד מצמתי ה- A בעץ הגזירה.

אם חישוב של תוכנה מסוימת (נאמר x.A) בצומת N בעץ הגזירה משתמש בערך של תוכנה אחרת (נאמר y.B) או אמורים ש- x.A תלוי ב- y.B. בגרף התלויות תהיה קשת מהצומת עבור y. שצומת עבור x.A. זאת אומרת שם בגרף התלויות יש קשת מצומת  $\delta$  לצומת c, פרוש הדבר שהיבאים לחשב את התוכנה ש-  $\delta$  מייצגת לפני שמחשבים את התוכנה ש- c מייצגת.

כדי למצוא סדר שבו ניתן לחשב את כל התוכנות, יש למצוא **מיון טופולוגי** (topological sort) של גרף התלויות. מיון טופולוגי של גרף מכובן חסר מעגלים (directed acyclic graph) הוא סידור של צמתים הגרף ...  $\delta_3, \delta_2, \delta_1$  כך שם יש קשת בגרף מצומת  $\delta_i$  לצומת  $\delta_j$  אז  $i < j$  בסידור. (לאו דווקא מיד לפני). הערה: לגרף מסוים יכולים להיות מספר מיונים טופולוגיים.

לאחר שמצאים מיון טופולוגי של גרף התלויות, ניתן לחשב את כל התוכנות בעץ הגזירה לפי הסדר שבו הם מופיעים במילוי.

**אלגוריתם למציאת מיון טופולוגי עבור גרף מכובן חסר מעגלים**  
אתחל את רשימת התוצאות לרשימה ריקה

כל עוד נותרו צמתים בגרף בצע:  
מצא צומת  $\delta$  שלא נכנסה לתוכה אף קשת.  
הוסף את  $\delta$  לרשימה התוצאות.  
מחק את  $\delta$  ואת כל הקשות היוצאות ממנו מהגרף.

אם בגרף הతלוויות מופיע מעגל אז ה- SDD אינו תקין ולא ניתן לחשב את התכונות בכל צמתי עץ הגזירה. הנה לדוגמה חלק מ-SDD לא תקין הכלול הגדרה מעגלית :

$$A \Rightarrow B C \quad A.s = B.b * 8 \quad B.b = A.s + 17$$

כאן התכונה  $s$ .  $A$  בצומת  $N$  המסומנת ב- $A$ - מחושבת בעזרת התכונה  $B.b$  בילד של  $N$  בעוד שזו האחרונה מחושבת בעזרת  $s$ .  $A.s$  בצומת  $N$ .

ברוב ההגדרות שנראה נוכל למצוא את הסדר שבו נחשב את התכונות בכל צמתי עץ גזירה מבלי שנייאץ לבנות גראף תלויות.

### הגדרה מסוג S

SDD הcollat רק תכונות נוצרות נקראת הגדרה מסוג S (S attributed definition).

בଘדרות מסוג S תמיד ניתן לחשב את ערכי התכונות בכל צמתיו של עץ גזירה נתון על ידי מעבר על העץ "מלמטה למעלה". מחשבים את ערכי התכונות בכל צומת לאחר שחושבו ערכי התכונות בכל ילדיו. post order traversal. שימושם לב שאין צורך לחשב את ערכי התכונות בעליים כי הם מסומנים בטרמינלים.

### דוגמא : חישוב ערך של ביטוי ארכיטמטי

כל התכונות כאן הן נוצרות :  
משמעות התכונות :

זה הערך של הביטוי  $E$ - מיצג (E.val

זה הערך של ה-term T-val

זה הערך של ה-factor F-val

זה המספר שהאסימון num.lexval מיצג

#### כל גזירה

#### כל סמנטי

(1) $L \rightarrow E$	print (E.val)
(2) $E \rightarrow E_1 + T$	$E.val = E_1.val + T.val$
(3) $E \rightarrow T$	$E.val = T.val$
(4) $T \rightarrow T_1 * F$	$T.val = T_1.val * F.val$
(5) $T \rightarrow F$	$T.val = F.val$
(6) $F \rightarrow ( E )$	$F.val = E.val$
(7) $F \rightarrow \underline{num}$	$F.val = \underline{num}.lexval$

### דוגמא : חישוב הערך המקסימלי של סידרת מספרים

נשתמש בתכונות הבאות :

$\underline{num}.val$  זה המספר שהאסימון num מיצג.

$S.max$  זה המקסימום בסידרת המספרים ש- S מיצג.

$L.max$  זה המקסימום בסידרת המספרים ש- L מיצג.

#### כל גזירה

#### כל סמנטי

(1) $S \rightarrow L$	$S.max = L.max$
(2) $L \rightarrow L_1 \underline{num}$	if ( $\underline{num}.val > L_1.max$ ) $L.max = \underline{num}.val$

$  \begin{aligned}  &\text{else} \\  &\quad \text{L.max} = \text{L}_1.\text{max} \\  (3) \text{ L} \rightarrow \text{num} &\quad \text{L.max} = \underline{\text{num}}.\text{val}  \end{aligned}  $	<p><u>דוגמה : חישוב הערך של מספר בינרי</u> גם כאן כל התוכנות הן נוצרות.</p> <p>S.val זה הערך של סידרת הביטים המיוצגת על ידי S. L.val זה הערך של סידרת הביטים המיוצגת על ידי L. B.bit זה הערך של הביט היחיד המיוצג ע"י B.</p>
-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

כל גזירה	כל סמנטי
(1) $S \rightarrow L$	$S.\text{val} = L.\text{val}$
(2) $L \rightarrow L_1 B$	$L.\text{val} = 2 * L_1.\text{val} + B.\text{bit}$
(3) $L \rightarrow B$	$L.\text{val} = B.\text{bit}$
(4) $B \rightarrow 0$	$B.\text{bit} = 0$
(5) $B \rightarrow 1$	$B.\text{bit} = 1$
<u>דוגמה : חישוב הערך של מספר בינרי : גירסה שנייה</u>	
שימושו לב שכלל הגזירה השניה השתנה נשתמש בתוכנות הבאות :	
S.val, L.val, B.bit -- כמו קודם – מספר הביטים בסידרת הביטים ש-L.length מייצג.	

כל גזירה	כל סמנטי
(1) $S \rightarrow L$	$S.\text{val} = L.\text{val}$
(2) $L \rightarrow B L_1$	$L.\text{val} = L_1.\text{val} + B.\text{bit} * 2^{L_1.\text{length}}$ $L.\text{length} = L_1.\text{length} + 1$
(3) $L \rightarrow B$	$L.\text{val} = B.\text{bit}$ $L.\text{length} = 1$
(4) $B \rightarrow 0$	$B.\text{bit} = 0$
(5) $B \rightarrow 1$	$B.\text{bit} = 1$

#### דוגמה : בניית syntax tree עבור ביטוי אריתמטי

- או נשתמש ברוטיונות הבאות לצורך יצירת עצמים בsyntax tree
  - mkleaf (type, info) יוצרת עלה ומחזיר מצביע לעלה זהה.
  - העליה יכולה לייצג מספר. במקרה זה הארגומנט הראשון הוא NUM והארגומנט השני הוא המספר עצמו. העלה גם יכולה לייצג מזהה. במקרה זה הארגומנט הראשון הוא ID והארגומנט השני הוא מצביע לכניסה בטבלת הסמלים עבור אותו המזהה.
  - mknnode (op, leftchild, rightchild) מיצרת צומת שבשדה האופרטור שלו מופיע דוחילדים שלה הם leftchild ו-rightchild (שני הארגומנטים האחרונים הם מצביעים לצמתים). מחזיר מצביע לצומת שיצר.

גם כאן כל התוכנות הן נוצרות. הערך של E.tree הוא מצביע ל-E.tree שהוא מצביע לערך syntax tree עבור הביטוי ש-

מיצג. דבר דומה נכון עבור התוכנות של F ו-T.

נניח שלאסימון `id` יש תכונה `entry` שמצוינה לכניסה בטבלת הסמלים עבור מזהה זהה.

כל גזירה	כל סמנטי
----------	----------

(1) $E \rightarrow E_1 + T$	$E.\text{tree} = \text{mknnode} ('+', E_1.\text{tree}, T.\text{tree})$
(2) $E \rightarrow T$	$E.\text{tree} = T.\text{tree}$
(3) $T \rightarrow T_1 * F$	$T.\text{tree} = \text{mknnode} ('\ast', T_1.\text{tree}, F.\text{tree})$
(4) $T \rightarrow F$	$T.\text{tree} = F.\text{tree}$
(5) $F \rightarrow ( E )$	$F.\text{tree} = E.\text{tree}$
(6) $F \rightarrow \underline{\text{num}}$	$F.\text{tree} = \text{mkleaf} (\text{NUM}, \underline{\text{num}}.\text{lexval})$
(7) $F \rightarrow \underline{\text{id}}$	$F.\text{tree} = \text{mkleaf} (\text{ID}, \underline{\text{id}}.\text{entry})$

#### דוגמה : סידרת פיבונצ'י

סידרת פיבונצ'י היא סידרה אינסופית של מספרים שלמים שבה כל מספר (החל מהמספר השלישי) שווה לסכום שני המספרים הקודמים לו. שני המספרים הראשונים בסידרה הם אפס ואחד. לדוגמה, הסידרה הבאה היא רישא (התחליה) של סידרת פיבונצ'י :

0 1 1 2 3 5 8 13 21

ההגדרה מונחת תחביר הבאה מחשבת את התוכונה  $S.\text{fib}$  שערכה true אם סידרת המספרים ש-  $S$  מיצגת היא רישא של סידרת פיבונצ'י ואחרת ערכה false.

נשתמש בתוכנות הבאות :

$\underline{\text{num}}$  -- המספר שהאSIMON  $\underline{\text{num}}$  מיצג.

$\text{L.fib}$  -- תוכונה שערכה true כאשר סידרת המספרים  $L$  היא רישא של סידרת פיבונצ'י. אחרת ערכה false.

$\text{L.last}$  -- הערך של המספר האחרון בסידרת המספרים  $L$ .  
 $\text{L.before\_last}$  -- הערך של המספר הלפני האחרון בסידרת המספרים  $L$ .

#### כלל גזירה

#### כללים סמנטיים

$S \rightarrow L$	$S.\text{fib} = L.\text{fib}$
$L \rightarrow L_1 \underline{\text{num}}$	$L.\text{last} = \underline{\text{num}}.\text{val}$ $L.\text{before\_last} = L_1.\text{last}$ $\text{if } (\underline{\text{num}}.\text{val} \neq (L_1.\text{before\_last} + L_1.\text{last}))$ $\quad L.\text{fib} = \text{false}$ $\text{else}$ $\quad L.\text{fib} = L_1.\text{fib}$
$L \rightarrow \underline{\text{num}}_1 \underline{\text{num}}_2 \underline{\text{num}}_3$	$L.\text{last} = \underline{\text{num}}_3.\text{val}$ $L.\text{before\_last} = \underline{\text{num}}_2.\text{val}$ $\text{if } (\underline{\text{num}}_1.\text{val} == 0 \text{ and } \underline{\text{num}}_2.\text{val} == 1$ $\quad \text{and } \underline{\text{num}}_3.\text{val} == 1)$

```

L.fib = true
else
    L.fib = false

```

אפשרות נוספת  
 כאן השתמש בתכונות הבאות:  
 -- S.fib, L.fib, num.val, L.last  
 -- S.sum2 -- סכום שני המספריים האחרונים בסידרת המספרים ש- L מייצג.

כלל גזירה	כללים סמנטיים
$S \rightarrow L$	$S.fib = L.fib$
$L \rightarrow L_1 \underline{num}$	$L.last = num.val$ $L.sum2 = L_1.last + \underline{num}.val$ $\text{if } (\underline{num}.val != L_1.sum2)$ $L.fib = \text{false}$ $\text{else}$ $L.fib = L_1.fib$
$L \rightarrow \underline{num}_1 \underline{num}_2 \underline{num}_3$	$L.last = \underline{num}_3.val$ $L.sum2 = \underline{num}_2.val + \underline{num}_3.val$ $\text{if } (\underline{num}_1.val == 0 \text{ and } \underline{num}_2.val == 1$ $\text{and } \underline{num}_3.val == 1)$ $L.fib = \text{true}$ $\text{else}$ $L.fib = \text{false}$

הנה מספר דוגמאות ל- SDD הכוילים גם תכונות מורשות.

#### דוגמה : הוספה מידע אודוות טיפוסים לטבלת סמלים

בדוגמה זו נניח שיש טבלת סמלים שבה יש כניסה עבור כל משתנה שב שומרים (בין השאר) את הטיפוס שלו. לאסימון id יש תכונה name שערכה הוא שם המשתנה (כמחרוזת).  
 הרוטינה (name, type) addSymbol יוצרת כניסה חדשה בטבלת הסמלים שבה יופיע שם המשתנה name והטיפוס שלו.type

כלל גזירה	כלל סמנטי
(1) $D \rightarrow T L$	$L.type = T.type$
(2) $T \rightarrow \underline{\text{int}}$	$T.type = INT$
(3) $T \rightarrow \underline{\text{double}}$	$T.type = DOUBLE$
(4) $L \rightarrow L_1, \underline{id}$	$L_1.type = L.type$

(5)  $L \rightarrow \underline{id}$

addSymbol ( $\underline{id.name}$ ,  $L.type$ )

addSymbol ( $\underline{id.name}$ ,  $L.type$ )

$T.type$  היא תכונה נוצרת.

$L.type$  היא תכונה מורשת –  $L$  מייצג כאן רשיימה של  $\underline{id}$ . לא ניתן מתוך התבוננות בראשימה כזו לדעת מה הטיפוס של המשתנים. לצורך כך יש להתבונן בהקשר – במקרה זה ההקשר הרלוונטי הוא הטיפוס שמוופיע לפני הרשיימה.

דוגמא : חישוב הערך של מספר בינירי (פעם שלישיית)

ניתן להשתמש כאן בקבוצות רק בתוכנות נוצרות אבל לשם הדוגמה נשתמש גם בתוכונה המורשת הבאה :

w.B. זה ה"משקל" של הבית שמייצג B. לביט הימני ביותר יש משקל 1, לביט השני מימין יש משקל 2, לשישי משקל 4, לרבייעי 8 וכן הלאה.

כמו כן נשתמש בתוכנות הנוצרות הבאות :

B.bit זו הסיבית ש- B מייצג (אפס או אחד)

$L.val$  זה הערך של סידרת הביטים  $L$ .

$L.length$  זה מספר הביטים בסידרת הביטים  $L$ .

$S.val$  זה הערך של המספר הבינירי כולם.

כלל גזירה	כלל סמנטי
(1) $S \rightarrow L$	$S.val = L.val$
(2) $L \rightarrow B \ L_1$	$L.val = L_1.val + B.bit * B.w$ $L.length = L_1.length + 1$ $B.w = 2^{L_1.length}$
(3) $L \rightarrow B$	$L.val = B.bit$ $L.length = 1$ $B.w = 1$
(4) $B \rightarrow 0$	$B.bit = 0$
(5) $B \rightarrow 1$	$B.bit = 1$

הערה  $w.B$  היא תכונה מורשת ולא נוצרת כי לא ניתן מתוך התבוננות ב B לדעת מה משקלו אלא צריך לזכור בחשבו את "הקשר" שבו מופיע ה – B.  
(במקרה זה ההקשר הרלוונטי הוא מספר הביטים שמוופיעים מימין ל – B).

הערה : זו אינה הגדרה מסוג L (L attributed definition). אפשר לחשב את התוכנות בכל צמתי עץ גזירה נתון לפי הסדר הבא :  
נחשב את ערכי  $L.length$  בכל הצמתים המסומנים ב- L (יש לעבור על הצמתים מלמטה למעלה).  
נחשב את  $w.B$  בכל הצמתים המסומנים ב- B (לא משנה באיזה סדר עוברים על צמתים אלו).  
נחשב את  $B.bit$  בכל צמתי B (לא משנה הסדר)  
נחשב את  $L.val$  בכל הצמתים המסומנים ב- L (נעבור עליהם מלמטה למעלה)  
לבסוף נחשב את  $S.val$  בשורש.

דוגמא : חישוב הערך של מספר בינירי (פעם רביעית)  
(שים לב שהדקוק אינו זהה לדדקוק בדוגמא הקודמת).

כאן משתמשים בתוכנות המורשת הבאות :

w.B. זה ה"משקל" של הבית שמייצג B. (כמו בדוגמא הקודמת).

$L.in$  זה מספר הביטים שמיימי לסדרת הביטים  $L$ .

כמו כן משתמשים בתוכנות הנוצרות הבאות :

--  $B.bit$  -- הסיבית ש- B מייצג

זו התרומה של סידרת הביטים L לתוצאה הסופית (זה סכום התרומות של הביטים המרכיבים את L). גם זה כמו בדוגמה הקודמת.

כלל גזירה	כלל סמנטי
(1) $S \rightarrow L$	$L.in = 0$ print (L.val)
(2) $L \rightarrow L_1 B$	$L_1.in = L.in + 1$ $B.w = 2^{L.in}$ $L.val = L_1.val + B.bit * B.w$
(3) $L \rightarrow B$	$B.w = 2^{L.in}$ $L.val = B.bit * B.w$
(4) $B \rightarrow 0$	$B.bit = 0$
(5) $B \rightarrow 1$	$B.bit = 1$

הערה : זו הגדרה מסוג L.

הערה נוספת : ניתן היה בקלוות לוותר על הוכנה w.B.w  
למשל לכלל הגזירה  $B \rightarrow L_1 B \rightarrow L$  ניתן היה לשيق את הכלל הסמנטי:  
 $L.val = L_1.val + B.bit * 2^{L.in}$

#### דוגמה נוספת -- תרמוסטט

הזדוק הבא מתאר סידרה של פקודות לתרמוסטט:

$S \rightarrow L$   
 $L \rightarrow L C$   
 $L \rightarrow \underline{\text{initial num}}$   
 $C \rightarrow \underline{\text{up num}}$   
 $C \rightarrow \underline{\text{up}}$   
 $C \rightarrow \underline{\text{down num}}$   
 $C \rightarrow \underline{\text{down}}$   
 $C \rightarrow \underline{\text{set num}}$

הפקודה initial num היא תמיד הפקודה הראשונה והיא קובעת את הטמפרטורה ההתחלתית.

הפקודה num up משמעותה הולכת את הטמפרטורה ב- val.num מעלות.  
הפקודה up בלבד (ללא מספר) משמעותו הולכת את הטמפרטורה במעלה אחת.

הפקודה num down אומرت להוריד את הטמפרטורה במספר המעלות הנוכחי (מעלה אחת אם לא נתון מספר).

הפקודה set num אומרת לשנות את הטמפרטורה ל- val.num מעלות.

דוגמה : עבור הקלט הבא (במקום כל num כתובナンך ערך המיספרי) :

initial 20 up 10 down 3 up

הטמפרטורה תהיה לפני הסדר (משמאלי לימין) : 20, 30, 27, 28  
והטמפרטורה הסופית היא 28.

עבור הקלט הבא : initial 20 up 10 set 40 down 3 up  
הטמפרטורה תהיה לפני הסדר (משמאלי לימין) : 20, 30, 40, 37, 38  
והטמפרטורה הסופית היא 38.

הנה הגדרה מונחת תחביר שמחשבת את הוכונה  $S.\text{temp}$  – הטמפרטורה בסיום ביצוע סידרת הפקודות  $S$ .

נשתמש רק בתכונות נוצרות :

$-- L.\text{temp}$  -- הטמפרטורה בסיום ביצוע סידרת הפקודות  $L$ .  
למשתנה  $C$  יש שתי תכונות שקל להבין את משמעותן.

כללי גזירה	כללים סמנטיים
$S \rightarrow L$	$S.\text{temp} = L.\text{temp}$
$L \rightarrow L_1 C$	$\text{if } (C.\text{type} == \text{set})$ $L.\text{temp} = C.n$ $\text{else if } (C.\text{type} == \text{up})$ $L.\text{temp} = L_1.\text{temp} + C.n$ $\text{else /* } C.\text{type} == \text{down } */$ $L.\text{temp} = L_1.\text{temp} - C.n$
$L \rightarrow \underline{\text{initial num}}$	$L.\text{temp} = \underline{\text{num}}.\text{val}$
$C \rightarrow \underline{\text{up num}}$	$C.\text{type} = \underline{\text{up}}$ $C.n = \underline{\text{num}}.\text{val}$
$C \rightarrow \underline{\text{up}}$	$C.\text{type} = \underline{\text{up}}$ $C.n = 1$
$C \rightarrow \underline{\text{down num}}$	$C.\text{type} = \underline{\text{down}}$ $C.n = \underline{\text{num}}.\text{val}$
$C \rightarrow \underline{\text{down}}$	$C.\text{type} = \underline{\text{down}}$ $C.n = 1$
$C \rightarrow \underline{\text{set num}}$	$C.\text{type} = \underline{\text{set}}$ $C.n = \underline{\text{num}}.\text{val}$

### דוגמא : תרמוסטט (גירסה שנייה)

נשנה קצת את הדקדוק של הדוגמא הקודמת. הדקדוק החדש מייצר את אותה השפה כמו מקודם. (למעט הדיווק, כמעט אותה השפה. כאשר סידרת הפקודות חייבת להכיל לפחות שתי פקודות (initial ופקודה נוספת). סידרת הפקודות בדוגמא הקודמת יכולה להיות להכיל רק פקודה אחת אבל ההבדל אינו ממשוני).

הנה SDD שמחשב את `S.temp` הטמפרטורה הסופית בעקבות ביצוע סידרת הפקודות. יש לשמש בתוכונה מורשת אחת – `L.initial` – שנותנת את הטמפרטורה ההתחלתית. למשתנה `I` יש תוכנה נוצרת `val` שימושו שהיא קלה להבנה.

יתר התכונות הן נוצרות ומשמשותן כמו מקודם. ההגדרה היא מסוג `L`.

כל גזירה	כללים סמנטיים
$S \rightarrow I\ L$	$L.initial = I.val$ $S.temp = L.temp$
$I \rightarrow initial\ num$	$I.val = \underline{num}.val$
$L \rightarrow L_1\ C$	$L_1.initial = L.initial$ if ( <code>C.type == set</code> ) $L.temp = C.n$ else if ( <code>C.type == up</code> ) $L.temp = L_1.temp + C.n$ else /* <code>C.type == down</code> */ $L.temp = L_1.temp - C.n$
$L \rightarrow C$	if ( <code>C.type == set</code> ) $L.temp = C.n$ else if ( <code>C.type == up</code> ) $L.temp = L.initial + C.n$ else /* <code>C.type == down</code> */ $L.temp = L.initial - C.n$

$C \rightarrow \underline{up} \ \underline{num}$	$C.type = \underline{up}$ $C.n = \underline{num}.val$
$C \rightarrow \underline{up}$	$C.type = \underline{up}$ $C.n = 1$
$C \rightarrow \underline{down} \ \underline{num}$	$C.type = \underline{down}$ $C.n = \underline{num}.val$
$C \rightarrow \underline{down}$	$C.type = \underline{down}$ $C.n = 1$
$C \rightarrow \underline{set} \ \underline{num}$	$C.type = \underline{set}$ $C.n = \underline{num}.val$

הערה : ניתן גם לפתרו את השאלה עם הגדירה מונחת תחביר שימושת רק בתוכנות נוצרות. הרעיון הוא של משתנה  $L$  תהיה תכונה נוצרת  $temp$  שנוננת את הטמפרטורה בעקבות סידרת הפקודות  $L$ . למשנה  $L$  תהיה תכונה נוספת שאומרת אם  $L.temp$  היא יחסית או מוחלטת. אם  $L$  כולל פקודת  $set$  אז  $temp$  היא מוחלטת אחרת היא יחסית. "יחסית" זה אומר לדוגמה שעקבות סידרת הפקודות  $L$ , הטמפרטורה תעלה בשלוש מעלות (אבל הטמפרטורה הסופית תלולה בטמפרטורה לפני ביצוע  $L$ ).

#### טרמוסטט (גירסה שלישית)

הזדוק שוב משתנה  $currTemp$  נותרת כמעט ללא שינוי. הפעם סידרת הפקודות חייבת להסתיים באסימון  $end$ . שימושו לב שהכל עבר  $L$  הוא כאן רקורסיבי ימני ולא שמאלי כמו בדוגמאות הקודמות. הנה שוב הגדירה מונחת תחביר לחישוב  $S.temp$ .

נשתמש בתכונה מורשת  $L.prevTemp$  (קייזר של previous temperature) שנוננת את הטמפרטורה לפני ביצוע סידרת הפקודות  $L$  (כלומר הטמפרטורה בעקבות ביצוע הפקודות הקודומות ל-  $L$ ). בנוסף לכך, התכונה הנוצרת  $L.temp$  שנוננת את הטמפרטורה בעקבות ביצוע סידרת הפקודות  $L$ . לא קשה להבין את משמעות יתר התוכנות. החגדה היא מסוג  $L$ .

כלל גזירה	כללים סמנטיים
$S \rightarrow I \ L$	$L.prevTemp = I.val$ $S.temp = L.temp$ $I.val = \underline{num}.val$
$I \rightarrow initial \ num$	
$L \rightarrow C \ L_1$	if ( $C.type == set$ )  $L_1.prevTemp = C.n$ else if ( $C.type == up$ )  $L_1.prevTemp = L.prevTemp + C.n$

	else /* C.type == down */
	L <sub>1</sub> .prevTemp = L.prevTemp - C.n
	L.temp = L <sub>1</sub> .temp
L → <u>end</u>	L.temp = L.prevTemp
C → <u>up</u> R	C.type = <u>up</u> C.n = <u>R</u> .val
C → <u>down</u> R	C.type = <u>down</u> C.n = <u>R</u> .val
C → <u>set num</u>	C.type = <u>set</u> C.n = <u>num</u> .val
R → <u>num</u>	R.val = <u>num</u> .val
R → ε	R.val = 1

גם במקרה זה ניתן לכתוב SDD שמשתמש רק בתכונות נוצרות.

#### הגדירה מסוג L (הגדירה מושג L – attributed definition)

הגדירה מונחת תחביר היא L-attributed אם כל החישובים של כל התכונות המורשות מקיימים את התנאי הבא :

- בכל מהצורה מורשת של  $X_j$  נעשה על פि :
- תכונות מורשות של A.
  - תכונות כלשון של הסימנים המופיעים משמאלי ל-  $X_j$  באופן ימין  
(כלומר  $(X_1, X_2, \dots, X_{j-1})$ )

זאת אומרת שчисוב תכונה מורשת בצומת N של עץ גזירה יכול להשתמש רק בתכונות (כלשהן) של אחים הנמצאים לשמאלו ובתכונות מורשות בלבד באבא של N.

כל הגדירה מונחת תחביר מסוג S היא L-attributed כי ההגבלה בהגדירה של L-attributed definition.

כאשר הגדירה היא מסוג L ניתן לחשב ערך של תכונה בצומת N בעץ גזירה על סמך הקלט עד וככל המחרוזת הנgorot מ- N. מידע (ערכי תכונות) "זורם" משמאלי לימין ולכן הגדירה מכונה מסוג L (left to right).

כאשר ההגדרה היא מסוג L מנתח תחבירי מסוגל לחשב את ערכי כל התכונות בזמן שהוא עושה ניתוח תחבירי. (כמוון אם הדקדוק מתאים לשיטת הניתוח התחבירי). אפשרות אחרת היא שהמנתח התחבירי יבנה עצ' גזירה (או syntax tree) ואז מחשבים את ערכי התכונות ב策מי העץ תוך כדי מעבר אחד או יותר על העץ. אפשרות זו קיימת עבור SDD תיקון כלשהו לאו דווקא מסוג L.

ביתר פרוט: כל SDD L המבוסס על דקדוק שהוא (1) LL ניתן למימוש על ידי recursive descent parser יכול לחשב את ערכי התכונות בכל策מי עצ' הגזירה שהוא "בונה" (כמוון שבדרך כלל לא ממש בונים את העץ).

bottom up parser יכול לחשב תכונות נבנות תוך כדי ניתוח תחבירי ע"י כך שמחזיקים את התכונה של סימן דקדוקי בלבד אליו ב-parsing stack.

כדי למש SDD L attributed SDD הכולל תכונות מורשות ע"י bottom up parser את הדקדוק ע"י הוספה marker non terminals (תפקידם של אלו להחזיק תכונות מורשות - זה מוסבר בספר הלימוד למי שמעוניין). אם הדקדוק המתקבל ניתן לניתוח תחבירי ע"י ה-bottom up parser אז הוא יכול לחשב את ערכי כל התכונות בזמן שהוא עושה ניתוח תחבירי. בפרט יש משפט שאומר שאם הדקדוק המקורי היה (1) LL או הדקדוק המתקבל בעקבות הוספה ה-.LR (1) marker non terminals הוא.

רוטינה לחישוב ערכי תכונות בעץ גזירה כאשר ההגדרה מונחת התחביר היא depth first traversal () visit נעביר את השורש של עצ' הגזירה. () visit עושה לקרואיה הראשונה לרוטינה visit () נעביר את השורש של עצ' הגזירה. () visit עושה של העץ ומחשבת את ערכי התכונות בכל策מי.

```

visit (node N) {
    for (each child C of N, from left to right) {
        evaluate inherited attributes at node C;
        visit (C);
    }
    evaluate synthesized attributes at node N;
}

```

עודכן לאחרונה: 6 אפריל 2012

## הטלה לאחר (backpatching) עם bison

נניח שדקודוק עבר שפט תכנות פשוטה כולל בין היתר את כללי הגזירה הבאים:

```

stmt: IF '(' boolexp ')' stmt ELSE stmt
stmt: WHILE '(' boolexp ')' stmt
stmt: ID '=' expression ';'
boolexp: ID RELOP NUM

```

**תאור הבעיה**  
דוגמא: את הקלט

```
if (a > 3) b = 0; else z = (c + 1) * 17;
```

נרצה לתרגם לקוד הביניים הבא:

```

20: ifFalse a > 3 goto 23
21: b = 0
22: goto 26
23: t1 = c + 1
24: t2 = t1 * 17
25: z = t2
26:

```

לשם הנוחיות, ליד כל פקודה נרשם מספירה הסידורי. כאן הנחנו שקטע הקוד מתחילה בפקודה מס' 20.

יעדי הקפיצות הן מספרי פקודות. הבעיה היא שבזמן שמייצרים את פקודות הקפיצה, לעיתים קרובות המספר של פקודת היעד עדין לא ידוע.

פתרון: נשתמש בשלב ראשון בתוויות סימבוליות שהשלב מאוחר יותר יומרו במספרי פקודות.

**פתרון אחר: backpatching (הטלה לאחר)**

הרעיון הוא שכאשר מייצרים פקודות קפיצה שייעda לא ידוע -- בשלב ראשון מייצרים את הפקודה ומשאירים את היעד ריק. מאוחר יותר, כאשר יודעים מה צריך להיות היעד -- משלימים את היעד החסר זהה (backpatching).

לצורך כך נוח לצבור את פקודות קוד הביניים בטבלה ולא לכתוב אותן מיד לקובץ דבר שהיה מסרבב את תהליך השלמת היעדים. בסופו של דבר אפשר לעבור על הטבלה ול כתוב את הפקודות לקובץ.

### **פונקציות ומשתנים גלובליים**

כאמור פקודות בקוד הביניים נצברות בתוך טבלה. המשתנה הגלובלי

```

next_instr
emit () (next_instr
          מחזיק את האינדקס של הכניסה הפנوية הבאה בטבלה. הפונקציה ()
          יוצרת פקודה בקוד ביניים, מוסיפה אותה לטבלה (lezinise zinstr
          ומקדמת את zinstr next באחד.

```

בالمשך נעשה שימוש ברשימות של פקודות קפיצה (שיעדן יותר ריק בשלב ראשון). בפועל רשימות אלו יכילה אינדקסים של פקודות הקפיצה בטבלה. (האינדקס של פקודת קוד בין היתר בטבלה הוא בעצם מספירה הסידורי).

הfonקציה () makelist יוצרת רשימה חדשה של אינדקסים, מכניתה לתווך את האינדקס אותו היא מקבלת כארוגומנט ומחזירה את הרשימה שיצרה.

הfonקציה () merge מחזירה רשימה שהיא מיזוג של הרשימות (של אינדקסים) אותם היא מקבלת כארוגומנט. היא יכולה לקבל שתיים או שלוש רשימות כארוגומנטים.

הfonקציה () backpatch מקבלת שני ארגומנטים. הראשון הוא רשימה של אינדקסים של פקודות קפיצה שיעדן יותר ריק. הארגומנט השני הוא האינדקס של היעד. היא עוברת על הפקודות ברשימה (שנמצאות בטבלה) וממלאת את יעד הקפיצה שלhn באינדקס היעד.

הארוגומנט הראשון של backpatch יכול גם להיות אינדקס של פקודה קפיצה בודדת (במוקם רשימה של אינדקסים שיש בה אינדקס בודד).

ל- backpatch ול- merge ניתן להעביר TUNNUM כארוגומנט. בהקשר זה TUNNUM מייצג רשימה ריקה.

#### **ערכיהם סמנטיים**

בתרגום שלנו, ביטוי בוליאני תמיד מתרגם לפקודת `ifFalse` אחת. פקודה זו קופצת לעד אם התנאי של הביטוי הבוליאני לא מתקיים. כאשר מיצרים פקודה זו יעדיה עדין לא ידוע ולכן נותר ריק. כדי שנוכל להשלים את היעד בממשך, שומרים את מספר הפקודה כערך הסמנטי של `boolexp`.

הערך הסמנטי של ID הוא שם המזהה.  
האסמימון RELOP מייצג אופרטור השוואה (קטן, קטן או שווה, שווה וכ"יו). הערך הסמנטי שלו אומר באיזה אופרטור מדובר.  
הערך הסמנטי של NUM הוא ערכו המספרי.

הערך הסמנטי של expression זה המשתנה שיחזיק את תוצאה חישוב הביטוי.

#### **קוד עברו Bison (פסאודו קוד)**

```
stmt: IF '(' boolexp ')' stmt ELSE skip stmt
      {      backpatch ($3, $7 + 1);
            backpatch ($7, next_instr); }
```

בדוגמא הניל', הקריאה הראשונה ל- () backpatch כתוב את היעד 23 בפקודת ה- `ifFalse` (פקודה מספר 20). הקריאה השנייה כתוב את היעד 26 בפקודת ה- `goto` (פקודה מספר 22).

```

skip: /* empty */ { $$ = next_instr;
                  emit ('goto _'); /* skip false stmt */
}

stmt: ID '=' expression ';' { emit ($1 '=' $3); }

boolexp: ID RELOP NUM { $$ = next_instr;
                        emit ('iffalse' $1 $2 $3 'goto _'); }

```

## משפטי while

### דוגמה

את המשפט הבא

```

while (foo <= 20)
    foo = (foo + 7) * c;

```

נרצה לתרגם הקוד הבינריים כך (נניח שקטע הקוד מתחילה בפקודה מספר 80) :

```

80:   iffalso foo <= 20 goto 85
81:   t1 = foo + 7
82:   t2 = t1 * c
83:   foo = t2
84:   goto 80
85:

```

הנה הקוד עבור :bison

```

stmt: WHILE marker '(' boolexp ')' stmt
      { emit ('goto' $2); /* jump to check loop
                           condition */
        backpatch (makelist ($4), next_instr);
      }

```

בדוגמה הניל', הקריאה ל- backpatch כתובות את 85 כיעד הקפיצה של פקודת - .iffalse

```

marker: /* empty */ { $$ = next_instr; }

```

המשתנה marker תפקידו במקרה זה להזכיר את מספר הפקודה הראשונה בקוד עבור הביטוי הבוליאני (קוד זה כולל פקודת אחת בלבד מסוג iffalso כי הביטויים הבוליאניים בשפה שלנו מאוד פשוטים אבל בשפות המאפשרות כתיבת ביטויים בוליאניים מורכבים הקוד שלהם עשוי לכלול מספר רב של פקודות).

זה אפשר לנו לייצר את פקודת ה- goto (בסוף הלולאה) שתפקידה לkapoz לקוד עבור הביטוי הבוליאני. הערכה : במקרה זה יכולנו לוותר על השימוש ב- marker כי ערכו .boolexp זהה לזה של ה-

נערך 24 אוגוסט 2014

**шиפור קטן בקוד**  
**דוגמא**  
**המשפט**

```

while (c < 100)
    if (b > 7)
        c = c + 8;
    else
        c = c * 2;

```

יתורגם לקוד הבא (נניח שקטע הקוד מתחילה בפקודה מס' 50)

```

50:  ifFalse c < 100 goto 58
51:  ifFalse b > 7 goto 55
52:  t1 = c + 8
53:  c = t1
54:  goto 57
55:  t2 = c * 2
56:  c = t2
57:  goto 50

```

פקודת ה-`goto` (מספרה 54) קופצת לפקודה 57 שהיא עצמה פקודה קופיצה. עדיף שפקודה 54 הייתה קופצת ישיר לעיד הסופי כלומר עיל יותר היה שפקודה 54 תהיה 50 ס' `goto`.

נראה כיצד ניתן ליצור את הקוד העיל יותר. נגיד למשתנה `stmt` ערך סמנטי. זה רשימה של (אינדקסים של) פקודות קופיצה שייעדן אמרור להיות הפקודה שצרכיה להתבצע אחריו סיום ביצוע ה-`stmt`. פקודות קופיצה אלו נוצרו כחלק מהקוד עבור `stmt`.

עד אפשרי של פקודות אלו היא הפקודה הראשונה שמופיעה אחריו הקוד של `stmt`. מאחר וקיים אפשרות שפקודה זו בעצמה תהיה פקודה קופיצה – העדפנו להשאיר את העיד ריק ביןתיים כדי שנוכל לקפוץ ישירות לעיד הסופי.

בדוגמה הנ"ל הערך הסמנטי של ה-`stmt` שמננו נוצר משפט ה-`If-else` – תחיה הרשימה (54) כי העיד של פקודת ה-`goto` במספרה 54 הוא הפקודה הבאה לביצוע אחריו סיום ביצוע משפט ה-`If-else`.

הערך הסמנטי של ה-`stmt` שמננו נוצר משפט ה-`while` כולל יהיה (50) כי העיד של פקודת ה-`ifFalse` במספרה 50 היא הפקודה הבאה לביצוע אחריו סיום ביצוע משפט ה-`while`.

הערך הסמנטי של ה-`stmt` ממנו נוצר המשפט `c = c + 8;` יהיה TNU (המייצג את הרשימה הריקה) כי בקוד שנוצר עבור משפט ההשמה זהה אין פקודות קופיצה לפקודה לביצוע אחריו.

בדוגמא פשוטה זו הערכים הסמנטיים של ה-`stmts` יכולים לפקודת קופיצה אחת לכל היותר. אבל במקרים מורכבים יותר, הערך הסמנטי יהיה רשימה של

מספר פקודות קפיצה.

### לדוגמה במשפט מהצורה

```
if (boolexp1)
    assignment-stmt1
else
    while (boolexp2)
        assignment_stmt2
```

از הערך הסמנטי של ה- `stmt` שגורזר את משפט ה- `If-else` יהיה רשיימה שתכיל שתי פקודות קפיצה: פקודה ה- `goto` שמטרתה לדרג על פני משפט השקר (לולאת ה- `while`) ופקודה ה- `ifFalse` שבודקת את התנאי של לולאת ה- `while`. לשתי הפקודות האלו צריך להיות אותו היעד: הפקודה שאמורה להתבצע אחרי סיום ביצוע משפט ה- `If-Else`.

דוגמה נוספת: אם במקום `assignment-stmt1` הינו רושמים לולאת `while` נוספת אז לרשיימה שהיא הערך הסמנטי של ה- `stmt` המייצג את משפט ה- `If _else If` כולל היתה מתווספת גם פקודה ה- `ifFalse` שבודקת את התנאי של ה- `while` החדש. היעד של פקודה זו הוא אותו היעד של שתי פקודות הקפיצה שהזכרו מוקדם.

### הנה הגרסה המשופרת של הקוד עבור `bison`:

```
stmt: IF '(' boolexp ')' stmt ELSE skip stmt
      { backpatch (makelist ($3), $7 + 1);
        $$ = merge ($5, $7, $8); }

skip: /* empty */ { $$ = next_instr;
                    emit ('goto _'); /* skip false stmt */ }

stmt: WHILE marker '(' boolexp ')' stmt
      { emit ('goto' $2); /* jump to check loop
                           condition */
        /* jumps from loop body
           should go to code for boolean expression:*/
        backpatch (%6, $2);
        $$ = makelist ($4);
      }

marker: /* empty */ { $$ = next_instr; }

stmt: ID '=' expression ';' { emit ($1 '=' $3);
                            $$ = NULL; }
```

```

boolexp: ID RELOP NUM { $$ = next_instr;
                           emit ('ifFalse' $1 $2 $3 'goto _'); }

```

מוסיף עוד כל גזירה לצורך חשלמת הדוגמא הבאה: (עכשו המשתנה ההתחלתי):

```

program: stmt { backpatch ($1, next_inst);
                  emit ("halt");

```

### **דוגמא להרצת המנתח התחבירי שנוצר על ידי bison**

נראה מה קורה בזמן ריצה של ה- parser שנוצר ע"י bison על הקלט

```

while (c < 100)
    if (b > 7)
        c = c + 8;
    else
        c = c * 2;

```

בכל שלב רואים כאן את תוכן המחסנית המשמשת לניתוח תחבירי (ללא מצבים האוטומט) ומעלה – את תוכן מחסנית הערכים הסמנטיים.

הקוד המיוצר מופיע כאן עם מספרי הפקודות לשם הנוחיות.  
אחריו ומלי הגזירה של expression לא נתונים לא ניתן לדעת את פרטי הניתוח התחבירי של ביטויים אבל ברור שקלט כמו  $c + 8$  יצומצם בסופו של דבר ל- .expression

stack contents	look ahead	action
	while	
while	(	shift
1 while marker	(	reduce by marker: epsilon
1 while marker (	ID	shift
1 c while marker ( ID	RELOP	shift
1 c < while marker ( ID RELOP	NUM	shift
1 c < 100 while marker ( ID RELOP NUM	)	shift
1 1 while marker ( boolexp	)	reduce by boolexp: ID RELOP NUM emit: 1: ifFalse c < 100 goto _

1 1	IF	shift
while marker ( boolexp )		
1 1	(	shift
while marker ( boolexp ) IF		
1 1	ID	shift
while marker ( boolexp ) IF (		
1 1 b	RELOP	shift
while marker ( boolexp ) IF ( ID		
1 1 b	NUM	shift
while marker( boolexp ) IF ( ID >		
RELOP NUM	)	shift
1 1 b	)	reduce by boolexp: ID RELOP NUM emit: 2: ifFalse b > 7 goto _
while marker( boolexp ) IF (		
2		
boolexp		
1 1	ID	shift
while marker ( boolexp ) IF (		
2		
boolexp )		
1 1	=	shift
while marker ( boolexp ) IF (		
2 c		
boolexp ) ID		
1 1	ID	shift
while marker( boolexp ) IF (		
2 c		
boolexp ) ID =		
1 1	;	c + 8 are shifted and then reduced to expression code emitted: 3: t1 = c + 8
while marker ( boolexp ) IF (		
2 c t1		
boolexp ) ID = exp		
1 1	ELSE	shift
while marker ( boolexp ) IF (		
2 c t1		
boolexp ) ID = exp ;		

1 1 while marker ( boolexp ) IF (	ELSE	reduce by stmt: ID = expression ; emit: 4: c = t1
1 1 while marker ( boolexp ) IF (	ID	shift
2 NULL boolexp ) stmt ELSE		
1 1 while marker ( boolexp ) IF (	ID	reduce by skip: epsilon emit: 5: goto _
2 NULL 5 boolexp ) stmt ELSE skip		
1 1 while marker ( boolexp ) IF (	\$	c = c * 2 are shifted c * 2 are reduced to expression emit: 6: t2 = c * 2 then ; is shifted reduce by stmt: ID = expression ; emit: 7: c = t2
2 NULL 5 NULL boolexp ) stmt ELSE skip stmt		
1 1 (5) while marker ( boolexp ) stmt	\$	reduce by stmt: IF '(' boolexp ')' ' ' stmt ELSE skip stmt backpatch((2), 5 + 1);
(1) stmt	\$	reduce by stmt: WHILE marker ( boolexp) stmt emit: 8: goto 1 backpatch ((5), 1)
program		reduce by program: stmt backpatch ((1), 9) emit: 9: halt

בש"כ הקלט

```
while (c < 100)
    if (b > 7)
        c = c + 8;
```

```
else  
    c = c * 2;
```

תורגם לקוד הבינאים:

```
1:  ifFalse c < 100 goto 9  
2:  ifFalse b > 7 goto 6  
3:  t1 = c + 8  
4:  c = t1  
5:  goto 1  
6:  t2 = c * 2  
7:  c = t2  
8:  goto 1  
9:  halt
```

הוכן ב-24 למאי 2011

### הגדרה מונחת תחביר : תרגום משפטי השמה לקוד בינוני

```
S --> id = E
    S.code = E.code ||
    gen (id.name '=' E.addr)

E --> E1 + E2

    E.addr = newtemp ();
    E.code = E1.code || E2.code ||
    gen (E.addr '=' E1.addr '+' E2.addr)

E --> E1 * E2

    E.addr = newtemp ();
    E.code = E1.code || E2.code ||
    gen (E.addr '=' E1.addr '*', E2.addr)

E --> ( E1 )

    E.addr = E1.addr
    E.code = E1.code

E --> id
    E.addr = id.name
    E.code = ''
    יוצר הקוד הבא : דוגמא : עבור הקלט
t1 = b + c
t2 = d + f
t3 = t1 * t2
a = t3
```

### bison מימוש עם

```
stmt : ID '=' expr { emit ($1 '=' $3); }

expr : expr '+' expr
      { TMP t = newtemp ();
        emit (t '=' $1 '+' $3);
        $$ = t; }
```

```

expr : expr '*' expr
    { TMP t = newtemp ();
      emit (t '=' $1 '*' $3);
      $$ = t; }

expr : '(' expr ')'
expr : ID { $$ = $1; }

```

### דוגמה להרצת המנתח התחביבי שנוצר על ידי bison

בכל שלב נראה כאן את תוכן שתי המחסניות: המחסנית המשמשת לניתוח התחביבי והמחסנית שבה נשמרים הערכים הסמנטיים. (E קיצור של (expr

stacks contents	remaining input	Action
a	a = (b + c) * (d + f) \$	
ID	= (b + c) * (d + f) \$	Shift
A	(b + c) * (d + f) \$	Shift
ID =	\$	
a	b + c) * (d + f) \$	Shift
ID = (	+ c) * (d + f) \$	Shift
a b	+ c) * (d + f) \$	Shift
ID = ( ID	+ c) * (d + f) \$	reduce by E : id
a b	c) * (d + f) \$	Shift
ID = ( E	c) * (d + f) \$	
a b	) * (d + f) \$	Shift
ID = ( E +	) * (d + f) \$	Shift
a b c	) * (d + f) \$	Shift
ID = ( E + ID	) * (d + f) \$	reduce by E : id
a b c	) * (d + f) \$	
ID = ( E + E	) * (d + f) \$	reduce by E : E + E
a t1	) * (d + f) \$	emit: t1 = b + c
ID = ( E	* (d + f) \$	Shift
a t1	* (d + f) \$	reduce by E : ( E )
ID = E	(d + f) \$	Shift
a t1	(d + f) \$	Shift
ID = E *	d + f) \$	Shift
a t1	+ f) \$	Shift
ID = E * (	+ f) \$	Shift
a t1 d	+ f) \$	Shift
ID = E * ( ID	+ f) \$	reduce by E : id
a t1 d	+ f) \$	
ID = E * ( E	f) \$	
a t1 d	f) \$	Shift

---

ID = E * ( E +			
a   t1   d   f	) \$		Shift
ID = E * ( E + ID			
a   t1   d   f	) \$		reduce by E : id
ID = E * ( E + E			
a   t1   t2	) \$		reduce by E : E + E
ID = E * ( E			emit: t2 = d + f
a   t1   t2	\$		Shift
ID = E * ( E )			
a   t1   t2	\$		reduce by E : ( E )
ID = E * E			
a   t3	\$		reduce by E : E * E
ID = E			emit: t3 = t1 * t2
stmt	\$		reduce by stmt : ID = E
			emit: a = t3
			Accept

---

### קוד ביניים עבור משפטי בקרה

הגדירה מונחת **תחביר** נשתמש בתכונות הבאות (כולן נוצרות).  
**stmt --> stmt.code** -- קוד הביניים עבור stmt (כאן הקוד יכול רק פקודה אחת  
**boolexp --> boolexp.code** -- קוד הביניים עבור boolexp (כאן הקוד יכול רק פקודה אחת  
**boolexp --> boolexp.falseLabel** -- התוית הסימבולית שאליה קופצים כאשר התנאי  
**boolexp --> boolexp** לא מתקיים.

```

boolexp --> id relop num

        boolexp.falseLabel = newlabel ();
        boolexp.code =
            gen ('ifFalse' id.name relop.op num.val
                  goto' boolexp.falseLabel)

stmt --> if boolexp then stmt1
        stmt.code = boolexp.code ||
                    stmt1.code ||
                    gen (boolexp.falseLabel ':')

stmt --> if boolexp then stmt1 else stmt2

        after = newlabel ();
        stmt.code = boolexp.code ||
                    stmt1.code ||
                    gen ('goto' after) ||

```

```

        gen (boolexp.falseLabel ':') ||
        stmt2.code ||
        gen (after ':')

stmt --> while boolexp do stmt1
        begin = newlabel ();
        stmt.code = gen (begin ':') ||
                    boolexp.code ||
                    stmt1.code ||
                    gen ('goto' begin) ||
                    gen (boolexp.falseLabel ':')

stmt --> id = num
        stmt.code = gen (id.name '=' num.val)

```

דוגמה: המשפט הבא

```

while a < 10 do
    if b >= 7 then
        x = y + z
    else
        x = c + d

```

יתורגם ל-

```

L4:    iffFalse a < 10 goto L1
        iffFalse b >= 7 goto L2
        t1 = y + z
        x = t1
        goto L3
L2:    t2 = c + d
        x = t2
L3:    goto L4
L1:

```

מימוש ההגדרה מונחת הת לחבר עם bison

```

boolexp: ID RELOP NUM
{ LABEL falselabel = newlabel ();
emit ('iffFalse' $1 $2 $3 'goto'
      falselabel);
$$ = falselabel; }

stmt: IF boolexp THEN stmt
     { emit ($2 ':'); }

```

```

stmt: IF boolexp THEN stmt skip_stmt ELSE
      { emit ($2 `:`); }
      stmt
      { emit ($5 `:`); }

skip_stmt: /* empty */
{ LABEL afterlabel = newlabel ();
  emit ('goto' afterlabel);
  $$ = afterlabel; }

stmt: WHILE beginloop boolexp DO stmt
    { emit ('goto' $2);
      emit ('$3 `:' ); }

beginloop: /* empty */
{ LABEL beginlabel = newlabel ();
  emit (beginlabel `:' );
  $$ = beginlabel; }

stmt: ID '=' NUM ';' { emit ($1 '=' $3); }

```

### דוגמא להרכבת המנתח התחבירי שנוצר על ידי bison

בכל שלב נראה כאן את תוכן שתי המחסניות: המחסנית המשמשת לניתוח התחבירי והמחסנית שבה נשמרים הערבים הסמנטיים. (b קיצור של boolexp)

stacks contents	remaining input	Action
WHILE	while a < 3 do if b > 7 then c = 8 ;\$	
L0 WHILE N	a < 3 do if b > 7 then c = 8 ;\$	Shift reduce by N : epsilon emit: L0:
L0 a	< 3 do if b > 7	Shift

WHILE N ID	then c = 8 ;\$	
L0 a < WHILE N ID RELOP	3 do if b > 7 then c = 8 ;\$	Shift
L0 a < 3 WHILE N ID RELOP NUM	do if b > 7 then c = 8 ;\$	Shift
L0 L1 WHILE N b	do if b > 7 then c = 8 ;\$	reduce by B : id relop num emit: ifFalse a < 3 goto L1
L0 L1 WHILE N b DO	if b > 7 then c = 8 ;\$	Shift
L0 L1 WHILE N b DO IF	b > 7 then c = 8 ; \$	Shift
L0 L1 b WHILE N b DO IF ID	> 7 then c = 8 ; \$	Shift
L0 L1 b > WHILE N b DO IF ID RELOP	7 then c = 8 ;\$	Shift
L0 L1 b > 7 WHILE N b DO IF ID RELOP NUM	then c = 8 ;\$	Shift
L0 L1 L2 WHILE N b DO IF b	then c = 8 ;\$	reduce using B : id relop num emit: ifFalse b > 7 goto L2
L0 L1 L2 WHILE N b DO IF b then	c = 8 ;\$	Shift
L0 L1 L2 c WHILE N b DO IF b then ID	= 8 ; \$	Shift
L0 L1 L2 c WHILE N b DO IF b then ID =	8 ; \$	Shift
L0 L1 L2 c 8 ;\$ WHILE N b DO IF b then ID = NUM		Shift
L0 L1 L2 c 8 \$ WHILE N b DO IF b then ID = NUM ;		Shift
L0 L1 L2 WHILE N b DO IF b then s	\$	reduce using S : id = num ; emit: c = 8

---

L0 L1	\$	reduce using
WHILE N b DO s		S : if B then S
		emit: L2:
S	\$	reduce using
		S : while N B do S
		emit: goto L0
		L1:
		Accept

---

הקלט בדוגמה מתורגם אם כן לסדרת הפקודות הבאה:

```

L0:
    ifFalse a < 3 goto L1
    ifFalse b > 7 goto L2
    c = 8
L2:   goto L0
L1:

```

יש כאן מספר שאלות מסווגים שונים שעסוקות בתרגום משפטי switch לקוד בינוניים.

### דוגמא: יצור קוד עם bison בעזרת backpatching

השאלה למועד מבחינה 2010B1

השאלה עוסקת בתרגום של משפטי switch פשוטים לקוד בינוניים.

הנה חלק מבדיקה המיועדת ל- **bison** והמתאר את פקודות ה- **switch**. זה חלק מבדיקה גדויל יותר הכלול כלי נזירה עבור סוגים נוספים של משפטיים ועבור ביטויים.

```
stmt   :  SWITCH caselist DEFAULT ':'  stmt
;
caselist  :      caselist CASE NUM ':'  stmt  BREAK ;
;
caselist :  '(' expression ')'
;
switch  ( (a+b) * 17 )
case 3:    c = (c + 1) * 20;
            break;
case 10:   a = 0;
            break;
case 25:   b = 15;
            break;
default:
            g = 19;
```

המשמעות של המשפט דומה למשמעות שלו בשפת C. כמו בשפת C, הפקודה `break` משמעותה שיש לסיים את ביצוע משפט ה-`switch`.

הנה תרגום אפשרי לקוד בינוני של המשפט הניל:

```
(100) t1 = a + b
(101) t2 = t1 * 17
(102) ifFalse t2 == 3 goto 107
(103) t3 = c + 1
(104) t4 = t3 * 20
(105) c = t4
(106) goto 114
(107) ifFalse t2 == 10 goto 110
(108) a = 0
(109) goto 114
(110) ifFalse t2 == 25 goto 113
(111) b = 15
(112) goto 114
(113) g = 19
```

כאן הנחנו שהפקודה הראשונה מספירה 100 (מספריה הפקודות אינן חלק מהפקודות).

יש להוסיף לכללי הגזירה הניל פעולות סמנטיות (בפסאודו קוד) הנחוצות לצורך התרגום לקוד בינוניים.

בפתרון יש להניח שפקודות בקוד הביניים נצברות בתוך טבלה. המשתנה глובלי `next_instr` מחזיק את האינדקס של הכניסה הפנوية הבאה בטבלה. הפקודה `()` יוצרת פקודה בקוד בינוניים, מוסיפה אותה לטבלה (לכניסה `(next_instr)`) ומקדמת את `next_instr` באחד.

הfonkcija () makelist יוצרת רשימה חדשה של אינדקסים (לתוך הטבלה הנ"ל), מכניתה לתוכה את האינדקס אותה היא מקבלת כารוגומנט ומוחזירה את הרשימה שיצרה.

הfonkcija () merge מקבלת שתי רשימות של אינדקסים (לתוך הטבלה הנ"ל) ומוחזירה רשימה שהיא המיזוג של שתיהן.

הfonkcija () backpatch מקבלת שני ארגומנטים. הראשון הוא רשימה של אינדקסים של פקודות קופיצה שייעדן יותר ריק. הארגומנט השני הוא האינדקס של היעד. היא עוברת על הפקודות ברשימה (שנמצאות בטבלה) וממלאת את יעד הקופיצה שהן באינדקס היעד.

לפונקציות הנ"ל ניתן להעביר כארוגומנט גם את הערך NULL המיציג בהקשר זה רשימה ריקה של אינדקסים.

הערך הסמנטי של expression הוא המשתנה שיחזיק (בזמן ריצת התוכנית) את תוצאה חישוב הבטוי. הערך הסמנטי של האסימון NUM זה המספר.

בפתרון אין צורך להגדיר ערך סמנטי למשתנה stmt.

רמז: ניתן להוסיף לדקדוק את כלל הגזירה

```
marker : /* empty */ { $$ = next_instr; }
```

ולהשתמש במשתנה marker בכלל הבא:

```
caselist : caselist
          CASE NUM ':'
          marker
          { add action here; }
          stmt
          BREAK
```

' ; '

כאן נרמז על מקום אחד שבו ניתן להוסיף פעולה סמנטית (action).

### פתרונות

```
stmt : SWITCH caselist DEFAULT ':' stmt
       { backpatch ($2.jumplist, next_instr); }

caselist : caselist
          CASE NUM   ':'
          marker
          { emit ("ifFalse $1.result == $3 goto _"); }
          stmt
          BREAK
          ' ; '
          { $$ .result = $1.result;
            $$ .jumplist = merge($1.jumplist,
                                  makelist (next_instr));
            emit ("goto _");
            backpatch (makelist ($5),
                        next_instr);
          }

caselist : '(' expression ')'
          { $$ .result = $2;
            $$ .jumplist = NULL; }
```

```
marker : /* empty */ {    $$ = next_instr; }
```

הערך הסמנטי של `marker` הוא `struct caselist` הכלל שתי שדות: השדה `result` הוא המשתנה שיחסוק את תוצאה הביטוי (המשתנה `t2` בדוגמה). השדה `jmpList` הוא רשימת אינדקסים של פקודות קפיצה שיעדן אמרור להיות הפקודה שתתבצע אחרי סיום ביצוע משפט `case`. בדוגמה הניל, `jmpList` עברו `caselist` המציג את כל ה-`case` - אם יכלול את האינדקסים 106, 109, 112.

היעד של כל הקפיצות האלו הוא 114.

#### SHIPOR BEKOD HANOTZER

אפשר לקבל קוד יעיל יותר במקרים מסוימים (זה לא נדרש בשאלת) אם נגדיר למשתנה `stmt` ערך סמנטי. זה רשימה של (אינדקסים של) פקודות קפיצה שיעדן אמרור להיות הפקודה שצריכה להתבצע אחרי סיום ביצוע ה-`stmt`. פקודות קפיצה אלו נוצרו חלק מהקוד עבור `stmt`.

נזכיר לדוגמא: במקרים מסוימים, הפקודה 114 (שלא רואים אותה כaan) תהיה פקודה קפיצה. זה יכול לקרות אם למשל משפט ה-`switch` הוא הגוף של לולאת `while`. אז פקודה 114 תהיה פקודה קפיצה לקוד שבודק את תנאי הלולאה (שיכול להופיע לפני הקוד של הגוף הלולאה). שימוש מושכל בערך הסמנטי של `stmt` מאפשר לייצר קוד שיקפו מפקודות 112, 109, 106, ישר לקוד שבודק את תנאי הלולאה במקום לקפוץ לפקודה 114 ומשם לקפוץ לקוד לבדיקת התנאי. לכן בזמן יוצר הקוד עבור משפט ה-`switch`, ישאר את ידי הקפיצות בפקודות 112, 109, 106 ריקים ונשמר את האינדקסים האלו כערך הסמנטי של `stmt` (ה-`stmt` שלו מוצמצמים את משפט ה-`switch`). זה מאפשר לנו למלא את ידי הקפיצות בהמשך.

הנה התוכנית עם השינויים המתבקשים:

```

stmt : SWITCH caselist DEFAULT ':' stmt
      { $$ = merge ($2.jumplist, $5); }

    :
   粲 עושים של שלוש רשימות כולל 7 $ : merge

caselist : caselist
          CASE NUM ':'
          marker
          { emit ("ifFalse $1.result == $3 goto _");
            stmt /* $7 */
            BREAK
            ';'
            { $$ .result = $1.result;
              $$ .jumplist = merge ($1.jumplist, $7,
                                    makelist (next_instr));
              emit ("goto _");
              backpatch (makelist ($5),
                         next_instr);
            }
          }

caselist : '(' expression ')'
          { $$ .result = $2;
            $$ .jumplist = NULL; }

marker : /* empty */ { $$ = next_instr; }

```

### דוגמא: יצור קוד עם תוויות סימבוליות בעזרת .bison

ההבדל מהדוגמה הקודמת היא שכן יודי הקפיצות הן תוויות סימבוליות ולא מספרי פקודות.  
כאן אין צורך ב-.backpatching

נתבונן לדוגמא במשפט הבא (זה אותו המשפט שהופיע בדוגמה הקודמת):

```
switch ( (a+b) * 17 )  
{  
    case 3:    c = (c + 1) * 20;  
                break;  
  
    case 10:   a = 0;  
                break;  
  
    case 25:   b = 15;  
                break;  
  
    default:  
        g = 19;
```

הנה תרגום אפשרי לקוד בינרים של המשפט הניל' (זה מאד דומה לתרגום בשאלת הקודמת רק שכן כאמור משתמשים בתוויות סימבוליות):

```
t1 = a + b  
t2 = t1 * 17  
ifFalse t2 == 3 goto L2  
t3 = c + 1  
t4 = t3 * 20  
c = t4  
goto L1  
L2:  
ifFalse t2 == 10 goto L3  
a = 0  
goto L1
```

L3:

```
iffFalse t2 == 25 goto L4  
b = 15  
goto L1
```

L4:

```
g = 19
```

L1:

הנה תוכנית עבר bison שמבצעת את התרגומים. הפונקציה emit () משמשת כאן להדפסת פקודות בקוד בינוני לקובץ.

```
stmt : SWITCH caselist DEFAULT ':' stmt  
       { emit ($2.exitlabel ':' ); }  
  
caselist : caselist  
          CASE NUM ':'  
          skiplabel  
          {emit ('iffFalse' $1.result '==' $3 'goto'$5); }  
          stmt  
          BREAK ';'   
          { emit ('goto' $1.exitlabel);  
            emit ($5 ':')  
            $$ = $1; /* same as: $$ . result = $1 . result;  
                         $$ . exitlabel = $1 . exitlabel; */  
          }  
          ;
```

```

caselist : '(' expression ')' {
    $$ .result = $2;
    /* in the example, L1 is generated here */
    $$ .exitlabel = newlabel () ; }

skiplabel: /* empty */ { /*this generates L2, L3 & L4
    in the example*/
    $$ = newlabel ();
}

```

הערך הסמנטי של `caselist` הוא `struct` הכיל שני שדות: `result` הוא המשתנה שיחזק את תוצאת הביטוי (`t2` בדוגמה). `exitlabel` היא התוויות אליה קופצים כדי לצאת משפט ה-`L1` (בדוגמה).

#### דוגמא לשאלה על הגדרה מונחת תחביב

גם שאלה זו עוסקת בתרגום משפטי `switch` לקוד בינאים. הקוד הנוצר כמו בדוגמה הקודמת (עם תוויות סימבוליות). גם כללי הבדיקה כמו בדוגמה הקודמת. כמו קודם, כללי גזירה אלו הם חלק מדווק גדוול יותר המתאר סוגים נוספים של משפטיים וביטויים.

יש לכתוב כלליים סמנטיים המתארים את התרגום של משפטי `switch` לקוד בינאים. לכל הגירה של `stmt` ( $\text{stmt} \rightarrow \text{SWITCH } (\dots)$ ) יש לשיעץ כלל סמנטי המתאר כיצד יש לחשב את התכונה `stmt.code` -- התרגום של `stmt` לקוד בינאים.

הчисוב הזה יחייב להגדיר תכונות נוספות. יש לרשום גם כלליים סמנטיים לחישוב תכונות אלו. כלליים אלו יש לשיעץ לכל הגירה הנתונים של `.caselist`.

יש להשתמש בתכונות הבאות (איןכם נדרשים לכתוב כללים סמנטיים לחישובם – זה לא אפשרי מאחר וככל הגירה של expression(expression.expression).expression.result

-- expression.code -- המסתנה שיחזיק את תוצאת חישוב הביטוי.  
-- NUM.val -- המספר אותו מייצג האסימון.

בנוסף לכך, יש להשתמש בפונקציה הבאה:

הפונקציה newlabel ממחישה תוית סימבולית חדשה (...L1, L2, L3 ...)

בכללים הסמנטיים יש להשתמש בסימון (...) gen כדי ליצור פקודת חדשה בקוד ביןיהם  
ובסימון | כדי לציין שרשור של קטעים של קוד ביןיהם.

#### פתרונות

stmt : SWITCH caselist DEFAULT ':' stmt <sub>1</sub>	stmt.code =  caselist.code     stmt <sub>1</sub> .code     gen (caselist.exitlabel ":")
caselist : caselist <sub>1</sub> CASE NUM ':' stmt BREAK ';'	caselist.result = caselist <sub>1</sub> .result  caselist.exitlabel = caselist <sub>1</sub> .exitlabel   skiplabel = newlabel ()  caselist.code = caselist <sub>1</sub> .code     gen ("ifFalse" caselist <sub>1</sub> .result "=="  NUM.val "goto" skiplabel)     stmt.code

	<pre> gen ("goto" caselist1.exitlabel)    gen (skiplabel ":") </pre>
caselist : '(' expression ')'	<pre> caselist.result = expression.result caselist.exitlabel = newlabel () caselist.code = expression.code </pre>

בדוגמה לעיל, זה `expression.result` .L1 זה זה `caselist.exitlabel` ה-

הנה פתרון נוסף שבו `caselist.exitlabel` היא תכונה מורשת. בפתרון הקודם כל התוכנות חן תכונות נבנות. שימו לב שرك הכללים לחישוב `caselist.exitlabel` השתנו.

stmt : SWITCH caselist DEFAULT ':' stmt1	<pre> caselist.exitlabel = newlabel () stmt.code = caselist.code    stmt1.code    gen (caselist.exitlabel ":") </pre>
caselist : caselist1 CASE NUM ':' stmt BREAK ';'	<pre> caselist.result = caselist1.result caselist1.exitlabel = caselist.exitlabel  skiplabel = newlabel ()  caselist.code = caselist1.code    gen ("iffFalse" caselist1.result "==" NUM.val "goto" skiplabel)    stmt.code    </pre>

	<pre>gen ("goto" caselist1.exitlabel)    gen (skiplabel ":")</pre>
caselist : '(' expression ')'	<pre>caselist.result = expression.result caselist.code = expression.code</pre>

עודכן 17 אפריל 2015 (הוסר הסבר על ערך סמנטי של mid rule action כי אין בזה שימוש כאן)

```

int lookahead;

union {
    char name[100];
    enum op op;
    int ival;
} lexicalValue;

void match(int token) {
    if (lookahead == token)
        lookahead = yylex();
    else
        error();
}

void stmt() {
    switch (lookahead) {
        case WHILE:
            stmt -> WHILE '(' boolexp ')' stmt
            match(WHILE);
            match('(');

            LABEL cond = newlabel();
            emit (cond ":");
            LABEL exitlabel = boolexp();

            match(')');
            stmt();
            emit ("goto" cond);
            emit (exitlabel ":" );
            break;

        case IF: /* stmt -> IF '(' boolexp ')' stmt ELSE stmt */
            match(IF);
            match('(');

            LABEL elselabel = boolexp();
            match(')');
            stmt();
            LABEL exitlabel = newlabel();
            emit ("goto" exitlabel);

            match(ELSE);
            emit (elselabel ":" );
            stmt();
    }
}

```

```
        emit (exitlabel ":");

break;

...
/* stmt */

LABEL
boolexp ()
{
    if (lookahead != ID)
error();
/* boolexp -> ID RELOP NUM */
char id[100];
strcpy (id, lexicalValue.name);
match (ID);

enum op relop = lexicalValue.op;
match (RELOP);

int num = lexicalValue.ival;

match (NUM);

LABEL falselabel = newlabel();
emit ("ifFalse" id relop num "goto" falselabel);

return falselabel;
}
/* boolexp */
```

יש כאן דוגמאות לייצור קודBINים בטכניות שונות. כל הדוגמאות עוסקות במשפט reverse שימושי בהמשך. שאלת על משפט זה הופיעה בבחינה 2012B2.

### השאלה מהבחן

השאלה עוסקת בתרגום לקודBINים של משפטי מסוג חדש בשפת תכנות פשוטה.  
התרגום יעשה ע"י recursive descent parser.

הנה כלל הגיירה הרלונטי (הטרמינלים כתובים באותיות גדולות או כסימן בודד המוקף בגרשיים):

```
stmt --> REVERSE expression ' ( ) ' BEGIN stmt stmt END
```

בדקذוק יש כללי גזירה נוספים עבור stmt ועבור expression שאינם מובאים כאן.

**המשמעות של המשפט:** מחשבים את הביטוי המופיע בסוגרים אחרי המילה השמורה reverse

אם ערכו גדול או שווה לאפס אז שני המשפטים מבוצעים בסדר הרגיל.  
אם ערך הביטוי הוא שלילי אז שני המשפטים מבוצעים בסדר הפוך: קודם מבוצע המשפט השני ואחריו זה מבוצע הראשון.

### נתבונן בדוגמה:

```
reverse (a + b * 17)
begin b = k + 5; c = (c + 3) * 17 end
```

הנה תרגום אפשרי לקודBINים של המשפט הנ"ל.

```
t1 = b * 17
t2 = a + t1
if t2 < 0 goto L2
L1: t3 = k + 5;
    b = t3
    if t2 < 0 goto L3
L2: t4 = c + 3
    t5 = t4 * 17
    c = t5
    if t2 < 0 goto L1
L3:
```

יש לכתוב (בפסאודו קוד) חלק מ- recursive descent parser שיתרגם משפטי reverse לקודBINים. **יש לכתוב רק את החלק הרלונטי של הפונקציה עבור המשתנה :stmt**

```
void stmt ()
{
    switch (lookahead) {
        case IF:    ... /* do not write code here .../
        case WHILE: /* do not write code here .../
        case REVERSE:
            /* complete this code */
            ...
    }
}
```

```
    }
}
```

הקוד שעליכם לכתוב יעשה שימוש בפונקציה `expression` (אותה אין לכתוב). הפונקציה `expression` עושה ניתוח תחבירי של ביטוי ומיצרת קוד בינים עבורו. היא מחזירה את המשתנה שבו תອחSEN תוצאה חישוב הביטוי.

המנתח התחבירי צריך להשתמש במשתנה הגלובלי `lookahead` ובפונקציה `()` (`expression` אין לכתוב).

הפונקציה `()` מקבלת איסימון כארגוומנט. היא משווה אותו לו-`lookahead`. אם הם שוים היא מתקדמת בקלט (ומעדכנת את `lookahead`) אחרת היא קוראת לפונקציה שפטפלת בשגיאות.

הניחו גם שכל סוג של איסימון ישנה הגדרה כמו לדוגמה :

#define REVERSE 300

בפתרון אפשר להשתמש בפונקציות הבאות :  
הפונקציה `emit` יוצרת פקודה בקוד ביןיהם ומדפסה אותה לקובץ.  
הפונקציה `newlabel` מחזירה תווית סימבולית חדשה (`L1, L2, L3 ...`).  
הפונקציה `()` יוצרת משתנה זמני חדש (`t1, t2, t3 ...`) ומחזירה אותו.  
אין להגדיר משתנים גלובליים נוספים בפתרון.

## פתרון

```
void
stmt  ()
{
    switch (lookahead) {
        case IF: ...
        case WHILE : ...
        case REVERSE:
            /* stmt --> REVERSE  '(' expression ')'
               BEGIN stmt  stmt END

               LABEL stmt1_label = newlabel (); /* L1 in the
example */
               LABEL stmt2_label = newlabel (); /* L2 in the
example */
               LABEL exit_label = newlabel (); /* L3 in the
example */

            match (REVERSE);

            match ('(');
            VARIABLE result = expression ();
            match (')');

            match (BEGIN);
```

```

    emit ("if " result "< 0 goto" stmt2_label);
    emit (stmt1_label ":");

    stmt ();
    emit ("if " result "< 0 goto" exit_label);

    emit (stmt2_label ":");

    stmt ();
    emit ("if " result "< 0 goto" stmt1_label);

    match (END);
    emit (exit_label ":");

    return;
    . . .
}

```

### פתרון עם bison

הנחה: הערך הסמנטי של expression זה המשתנה שיחזיק את התוצאה של הביטוי (t2 בדוגמה הנ"ל)

```

stmt --> REVERSE
labels
'(' expression ')'

BEGIN
{ emit ("if" $4 "< 0 goto" $2(stmt2_label));
  emit ($2(stmt1_label ":")); }

stmt

{ emit ("if" $4 "< 0 goto" $2(exit_label));
  emit ($2(stmt2_label ":")); }

stmt

{ emit ("if" $4 "< 0 goto" $2(stmt1_label));
  emit ($2(exit_label ":")); }
END

labels: /* empty */ { $$stmt1_label = newlabel();
                     $$stmt2_label = newlabel();

```

```
    $$ .exit_label = newlabel(); }
```

### פתרונות נוסף עם .bison

הפעם עם backpatching. יודי הקפיצות הם מספרים ולא תוויות סימבוליות.

הנחה: כמו קודם, הערך הסמנטי של expression זה המשתנה שיחזיק את התוצאה של הביטוי (t2 בדוגמה הנ"ל)

```

stmt --> REVERSE
    ' ( ' expression ' ) '


BEGIN
marker /* $6 */
{ emit ("if" $4 "< 0 goto" _); }
stmt


marker /* $9 */
{ emit ("if" $4 "< 0 goto" _); }


stmt

END

{ emit ("if" $4 "< 0 goto" $6 + 1);
backpatch (makelist ($6), $9 + 1);
backpatch (makelist ($9), next_instr); }

marker: /* empty */ { $$ = next_instr; }

```

### פתרון עם AST (Abstract Syntax Tree)

נניח כאן שהנתני הוא ביטוי בוליאני ולא ביטוי (expression) כמו לעלה (הערך של ביטוי בהקשר שלנו הוא מספרשלם בעוד שהערך של ביטוי בוליאני הוא true או false (או יציג של אלו למשל 1 ו-0)).

הקריאה ל- genBoolExp (truelabel, falselabel) מייצרת קוד עבור הביטוי הבוליאני שkopf ל- truelabel מתקיים וkopf ל- falselabel כאשר אין מתקיים. ו- .FALL\_THROUGH יכולם להיות גם falselabel

נניח שמשפט reverse מיוצגים כך:

```
class reverseStmt : public Stmt
{
```

```

public:
    /* constructor not shown */
void genStmt ();

BoolExp *_condition;
Stmt *stmt1, *stmt2;
}

reverStmt::genStmt () // pseudo code
{
    stmt1_label = newlabel();
    stmt2_label = newlabel();
    exit_label = newlabel();

    _condition->genBoolExp (FALL_THROUGH,
                           stmt2_label);

    emit(stmt1_label ":");

    _stmt1->genStmt ();

    _condition->genBoolExp (FALL_THROUGH,
                           exit_label);

    emit(stmt2_label ":");

    _stmt2->genStmt ();

    _condition->genBoolExp (FALL_THROUGH,
                           stmt1_label);

    emit(exit_label ":");

}

```

בפתרון זהה הקוד של הביטוי הבוליאני משוכפל שלוש פעמים ובכל ביצוע של המשפט הביטוי הבוליאני יחולש שלוש פעמים. זה לאiesel וזה עלול גם להיות לא נכון כי יתרן שהיחסובים השונים יניבו תוצאות שונות. למשל התרחיש הבא אפשרי: התנאי הוא  $3 < a$  והתנאי הזה מתקיים בהתחלה. לכן מבצעים את המשפט הראשון. אבל המשפט זה עלול לשנות את ערכו של  $a$  כך שהתנאי  $3 < a$  כבר יפסיק להתקיים. כתוצאה לכך אחרי ביצוע המשפט הראשון נקבע ל-  $\text{exit\_label}$  והמשפט השני לא יבוצע בכלל.

הפתרון (אותו לא נראה כאן) הוא לחשב את התנאי הבוליאני רק פעם אחת ולשמור את התוצאה שלו במשתנה זמני אותו נבדוק בהמשך בהתאם לצורך.

## מחסנית הקריאה לפונקציות

בכל פעם שמתבצעת קריאה לפונקציה מוקצת עבורה בזיכרון רשומה. הרשומה נדחפת למחסנית (המכונה run time stack) וכאשר הפונקציה חוזרת הרשומה מוסרת מהמחסנית (עושים לה pop). מחסנית היא מבנה נתונים מתאים לצורך זה כי הפונקציה האחורונה שנקרה היא הראשונה שהוזרת וכאשר היא חוזרת אין עוד צורך ברשומה שלה. רשומה שモקצת לפונקציה מכונה activation record או stack frame.

בכל רגע במהלך הריצת תוכנית (תהליך) ה-frame של הפונקציה שכרגע רצה נמצא בראש המחסנית. מתחתיה נמצא ה-frame של הפונקציה שקרה לפונקציה הנוכחית. מתחתיה נמצא ה-frame של הפונקציה שקרה לפונקציה הנוכחית וכן הלאה.

אחד הרегистרים של המעבד מצבייע למקום קבוע בתוך ה-frame של הפונקציה שכרגע רצה (כלומר ה-frame שבראש המחסנית).  
רегистר זה מכונה לעיתים activation record pointer או frame pointer (בקיצור ARP).  
כשהפונקציה שרצה ניגשת ל-frame שלה היא עושה את זה דרך ה-frame pointer ולכן הוא מכיר את ה-frame קובע את סדר "השדות" בו. offset של כל שדה ייחסית לו. הקומפיילר ייחס את אמ' משתנה מקומי a של הפונקציה שכרגע רצה נמצא ב-offset מיינוס 20 frame pointer. למשל אם המשתנה המקומי a של הפונקציה שקרה ב- offset 17 ל- a הקוד ייחסית לכטובה המשוררת ב-frame pointer או כדי לכתוב למשל את הערך 17 ל- a ימייצר הקומפיילר יפחית 20 מהערך של ה-frame pointer כדי לקבל את הכתובת של a וואז יכתוב 17 לכטובה זאת.

## מידע שנשמר ב-frame

הארגון הפנימי של frame ומה נשמר בו תלויים בשפת התכנות, בקומפיילר ובמעבד.

הנה רשימה של דברים שעשוים להישמר ב-frame :

ארוגמנטים לקריאה לפונקציה

כתבות החזרה

הערך המוחזר

משתנים מקומיים, משתנים זמינים

רегистרים שערכם נשמר כשוראים לפונקציה כדי שניתן יהיה לשחזר אותם כשהיא חוזרת.

(בינהם ה-frame ה- "הישן" שהצבייע לו. frame של הפונקציה שקרה לפונקציה הנוכחית

זה נקרא ה- dynamic link).

static link (על כך בהמשך)

לעתים חלק מהדברים הנ"ל נשמרים ברегистרים במקומות ב-frame מסיבות של יעילות כדי גישה לרגיסטר מהירה יותר מגישה ל-frame שנמצא למרחב הכתובות של התħallid וכן נמצא בזיכרון הראשי. (כמו כל דבר בזיכרון הראשי הוא עשוי להיות נמצא ב-cache או אولي בדיסק כפי שלומדים במערכות הפעלה אבל בכל מקרה גישה לרגיסטר מהירה יותר).

### calling convention, call sequence

הfonקציה הקוראת (caller) והfonקציה שקוראים לה (callee) משתפות פעולה כדי להקים את ה- stack frame החדש וandi "נקות אחריו" הפונקציה הנkerait כשהיא חוזרתandi שהfonקציה הקוראת תוכל המשיך ל谋求.

שתי הפונקציות (הקוראת והנקראית) צריכות להסכים על חלוקת העבודה ביניהם. הן פועלות לפי ה- calling conventions ("מוסכמות הקריאה") שມפרטות היכן הפונקציה הקוראת צריכה לכתוב את הארגומנטים, היכן היא צריכה לכתוב את כתובות החזורה, היכן הפונקציה הנקראית צריכה לכתוב את הערך שהיא מ傳ירה ומה חלוקת העבודה כשםוקם frame חדש וכאשר הוא מוסר מהמחסנית.

באופן כללי - calling convention תלוי בקומפイルר. אבל אם שני קומפילירים שונים משתמשים באותו מוסכמו אז קריאות לפונקציה יעבדו כשרה גם אם ה- callee וה- caller יקומפלו ע"י קומפイルר שונה (ולפעמים גם אם שתי הפונקציות נכתבו במקור בשפות תכנות שונות (ש망טורגמות לשפת מכונה של אותו מעבד)).

לדוגמא ה- calling convention יכול לקבוע שלושת הארגומנטים הראשונים של פונקציה כתבו ע"י ה- caller לשולחה רגיסטרים מסוימים (למשל r4, r2, r3) שם ימצא אותם ה- callee. אם יש ארגומנטים נוספים הם יכתבו ע"י ה- caller למחסנית.

- calling convention קובע גם מה לעשות עם ארגומנטים שלא ניתן לכתוב אותם לרגיסטרים (heap structs או systems). במקרה כזה ה- caller יכול לכתוב את הארגומנט לזיכרון (אפשר ל- callee ואת הכתובת של הארגומנט הוא יכול לרשום לרגיסטר או למחסנית שם הוא ימצא ע"י ה- .callee).

הקוֹמְפִילֵר מיפויים את הקוד שיבוצע ע"י ה- caller וה- callee בזמן הקריאה והחזרה מפונקציה. זה כולל קוד שיבוצע ע"י ה- caller לפני הקריאה ואחרי החזרה ממנה (זה ה- call sequence) וקוד שיבוצע ע"י ה- callee כשהוא מתחילה לרוץ (ה- prologue) ולפני שהוא חוזר (ה- epilogue). בין ה- prologue וה- epilogue, ה- callee מבצע את הקוד של גוף הפונקציה שלו. יש שכוללים במונח call sequence גם את ה- prologue וה- epilogue.

הנה דוגמא למה שעשוי להיות כולל ב-`call sequence`. (הפרטים המדויקים משתנים מקומפיילר לקומפיילר והם מותאמים למעבד לו מיועד הטעוד).

הפונקציה הקוראת מבצעת את הפעולות הבאות:

1. שומרת על המחסנית רגיסטרים שהיא אחראית על שמירתם (caller saved registers) ושהיא תזדקק לערכם בהמשך (אחרי החזרה מהקריאה).
2. דוחפת על המחסנית את ה-frame pointer (שמצוין ל-caller של ה-frame).
3. מחשבת את הארגומנטים ודוחפת אותם למחסנית (או כותבת אותם לרגיסטרים).
4. דוחפת את ה-static link החדש (ישמש את ה-callee) למחסנית.
5. דוחפת את כתובת החזרה למחסנית (או כותבת אותה לרגיסטר מסוים).
6. קופצת לקוד של הפונקציה הנקראית

כשהפונקציה שקוראים לה מתחילה לróż היא מבצעת בתחילת את הפעולות הבאות (זה ה-*prologue*):

1. שומרת על המחסנית רגיסטרים שהיא אחראית על שמירתם (callee saved registers) ושהיא כתובות לתוכם בהמשך.
2. נותנת ערך ל-frame pointer שיישמש אותה במהלך ריצתה.
3. מקצת מקום על המחסנית למשתנים המקומיים שלה (יתכן שחלקם יהיו ברגיסטרים במקום על המחסנית).

ואז הפונקציה שקוראים לה מבצעת את הקוד של גוף הפונקציה (תרגום של הקוד שהמתכוна כתב).

לפני שהיא חוזרת הפונקציה שנקרה מבצעת את ה-*epilogue*:

1. שומרת את הערך המוחזר על המחסנית (או ברגיסטר מסוים) שם הפונקציה הקוראת נמצא אותו.
2. משחזרת את הרגיסטרים שהיא שמרה (callee saved registers) כלומר כותבת אותם בחזרה לרגיסטרים המתאימים.
3. משחזרת את ה-frame pointer היישן (של הפונקציה שעשתה את הקריאה).
4. קופצת לכתובת החזרה.

לאחר החזרה מהפונקציה, הפונקציה שבצעה את הקריאה מבצעת את הפעולות הבאות:

1. משחזרת רגיסטרים שהוא שמרה (caller saved registers).
2. מסירה (עשיה *pop*) מהמחסנית את הארגומנטים, את כתובת החזרה (ואות ה-static link) שכתבה למחסנית.
3. ממשיכה לróż

כאמור חלוקת העבודה בין ה-caller וה-callee תלויים בקומpileר. באופן כללי ככל שה-callee יושה יותר זה עשוי לחסוך בזכרוントופס הקוד. כי קוד שמופיע ב-prologue וב-*epilogue* (כלומר מבוצע ע"י ה-callee) יופיע רק פעם אחת בעוד שקוד שבוצע ע"י ה-caller יופיע בכל מקום

שבו מתבצעת קריאה ל- callee.

יש דברים שרק ה- caller יכול לעשות: הוא זה שמחשב את הארגומנטים וכותב אותם במקום מסוים (על המחסנית או ברגיסטרים). הוא זה שכותב את כתובת החזרה למקום מסוים.

ויש גם דברים שרק ה- callee מסוגל לעשות: הוא זה שמקצת מקום על המחסנית עברו המשתנים המקומיים שלו כי הוא ידוע לכמה מקום הוא זוקק. בשפה כמו C ה- caller לא מכיר את המשתנים המקומיים של ה- callee ולכן לא ידוע כמה זכרון הם יתפסו. יתר דיק, כאשר הקומפיילר מקמפל את הקוד של ה- caller, הקוד של ה- callee בדרך כלל לא זמן (הוא יכול להיות בקובץ אחר) ולכן הקומפיילר לא מכיר את המשתנים המקומיים של ה- callee ולא יכול לייצר קוד שמקצת להם מקום.

שמירת רגיסטרים: בחלק גדול מהמעבדים, ה- caller וה- callee חולקים את אותם הרגיסטרים. זה אומר שה- callee עשוי לכתוב לרגיסטר ולדרוז ערך שהיה שמור בו שה- caller יזדקק לו בהמשך כשימוש לרווחי החזרה מהפונקציה.

לדוגמא נאמר שה- caller מחשב את הביטוי  $(g + b)^*$ . כך נראה הביטוי ב- source code אבל אחרי התרגום לשפת מכונה נניח שה- caller מחשב קודם את  $b^*$  ושומר את התוצאה ברגיסטר 4. עכשו הוא מבצע קריאה לפונקציה  $g$  (זה ה- callee) מתוך כוונה לחבר את הערך ש-  $g$  תחזיר לערך שנשמר ב- 4. אבל יוכל להיות שהפונקציה  $g$  כותבת לרגיסטר 4 ו אז יש חשש שנאבד את הערך של  $b^*$  שנשמר בו.

לכן בזמן הקריאה לפונקציה יש לשמור את ערכי הרגיסטרים בזיכרון ולהחזיר אותם (כלומר לכתוב את הערכים שנשמרו בחזרה לרגיסטרים) כוחזרים מהפונקציה. כך הפונקציה שביצעה את הקריאה תוכל להמשיך ולהשתמש בערכים שכתבה לרגיסטרים לפני הקריאה.

בעקרון, כל אחד מהצדדים (ה- caller וה- callee) מסוגל לשמור ולהחזיר את הרגיסטרים. יש rules叫做 calling conventions בהם מטלה זאת מתחלת בין שתי הפונקציות: ה- caller אחראי על שמירת (וחזוק) רגיסטרים מסוימים (אלו הם ה- caller saved registers) וה- callee אחראי על שמירה של רגיסטרים מסוימים אחרים (אלו הם ה- callee saved registers).

היתרון של חלוקת העבודה בין שני הצדדים הוא שnitן לפעם לחסוך עבודה: ה- caller לא צריך לשמור את כל הרגיסטרים שהוא אחראי עליהם אלא רק את אלו שהוא יזדקק להם בהמשך. ה- callee לא צריך לשמור את כל הרגיסטרים שהוא אחראי עליהם אלא רק את אלו שהוא עשוי לכתוב לתוכם.

(הערה: יש מעבדים שיש להם מה שמכונה register windows: מספר קבועות של רגיסטרים. ה- caller משתמש בקבוצה אחת של רגיסטרים וה- callee משתמש בקבוצה אחרת. כך חוסכים את הצורך בשמירה ומחזור של רגיסטרים. אבל זה מחייב תמייה של החומרה. במקרה שיש הרבה פונקציות שנקראו וטרם חזרו (כל אחת מהן משתמשת ב- window register). אז יתכן שכבר לא יהיו register windows פנויים ונאלצים לשמר ולשזר רגיסטרים).

### inlining

זו שיטה של אופטימיזציה: הקומpileר מחליף קריאה לפונקציה בקוד של גוף הפונקציה עצמה. זה כמובן אפשרי רק אם הקוד של הפונקציה זמין לקומpileר בזמן שהוא מקمل את הקוד שקורא לפונקציה. זה בעיקר שימושי כאשר מדובר בפונקציה קטנה כי הקוד של גוף הפונקציה משוכפל בכל מקום שעושים לה inlining. למה זה אופטימיזציה? כי זה חוסך את הצורך להקים stack frame לפונקציה ולנקוט אחרת. זה גם מאפשר לקומpileר לעשות אופטימיזציות שלוקחות בחשבון גם את גוף הפונקציה הנקראית וגם את הקוד שמקיף את הקריאה לה.

דוגמא:

```
int max(int a, int b) {
    return (a>b) ? a : b;
}

...
int i, j, k;
...
i = max(j, k); // this can be inlined
```

או הקומpileר יכול לipyiter קוד עבור השורה המסומנת כאילו היה כתוב:

i = (j > k) ? j : k;

כאן אין צורך ב- call sequence. זה גם עשוי לאפשר אופטימיזיות נוספת.

לדוגמא אם הקומpileר יודיע שבזמן הקריאה לפונקציה max הערך של j בהכרח 10 והערך של k בהכרח 8 אז הוא יכול לבצע אופטימיזציה נוספת ולהחליף את השורה الأخيرة ב- i = 10 =

### קינון של פונקציות (lexical scoping)

יש שפות תכנות (למשל Pascal, D) המאפשרות קינון של פונקציות. שפת C לא מאפשרת זאת.

זה אומר שכמו שניתן להגדיר משתנה מקומי בתוך פונקציה ניתן גם להגדיר פונקציה בתוך פונקציה. השם של הפונקציה המקוונת יהיה מוכर רק בתחום הפונקציה שמקיפה אותה

(כולל בתוך פונקציות אחרות שמקוננות בתוך הפונקציה המקיפה). כלל ה- scope האלו נקראים lexical scoping והם דומים לכללים עבור שמות אחרים (של משתנים למשל).

פונקציה מקוננת יכולה לגשת למשתנים המקומיים שלה, למשתנים מקומיים של הפונקציה שמקיפה אותה, למשתנים המקומיים של הפונקציה שמקיפה אותה וככ' הלאה.

כדי לתמוך בזה יש לאפשר לפונקציה לגשת בזמן ריצחה לא רק ל- stack frame שלה אלא גם ל- stack frames של הפונקציות שמקיפות אותה כי שם נמצאים המשתנים המקומיים.

לצורך כך יש לשמר בתוך כל frame מצביע ל- frame של הקיראה המאוחרת ביותר לפונקציה שמקיפה את הפונקציה הנוכחית ב- source code. המצביע הזה נקרא static link או access link. מעקב אחר שרשרת ה- static links מאפשר לפונקציה הנוכחית להגיע ל- frames של כל הפונקציות שמקיפות אותה ב- source code.

עבור כל גישה למשתנה מקומי של פונקציה שמקיפה את הפונקציה הנוכחית, הקומpileר מייצר קוד שעוקב אחר שרשרת ה- static links עד למוגעים ל- frame הרלוונטי ושם נמצא המשתנה המקומי ב- offset ידוע יחסית למצביע ל- frame.

הערה: בנוסף ל- static link נשמר בכל frame גם ה- frame pointer זה ה- dynamic link של הפונקציה שקרה לפונקציה הנוכחית. הוא נשמר כדי שנitin יהיה לשחזר את ה- frame pointer הישן כשחזררים מהפונקציה.

דוגמא :

```
int main() {  
    int k; // local to main  
  
    void f() {  
        int local_f; // local to f  
        void r1() {  
            int local_r1; //local to r1  
            void r2() { int local_r2; ... }  
            ... // body of r1  
        } // end of r1  
        ... // body of f  
    } // end of f
```

```

void g() { ... }

... // code for main

} // end of main

```

כאן ההפונקציה `g`, `f` מוקוננת בתוך `main`. `r1` מוקוננת בתוך `f` ו- `r2` מוקוננת בתוך `r1`.  
 לכל פונקציה יש רמת קיון (nesting level): רמת הקיון של `main` היא 0. רמת הקיון של `f` ו- `g` היא 1. רמת הקיון של `r1` היא 2. `r2` היא ברמת קיון 3.

הטבלה הבאה מראה איזה פונקציות מכירות את השמות שמופייעות בדוגמה. השמות מתיחסים למשתנים מקומיים או לפונקציות. שני סוגי השמות כפופים לאותם כלליים של scoping (הכללים שגדירים באיזה חלק מהתוכנית מוכרת כל הגדרה של שם). באופן כללי הגדרה של שם מוכרת בתוך כל הפונקציה שבה היא מופיעה (או בתוך כל הבלוק בו היא מופיעה) בלבד בבלוקים פנימיים בו מוגדר השם שוב ואז בתוך הבלוק הפנימי השם מתיחס להגדרה המקומית. (אין בדוגמה הגדרות שונות של אותו שם).

משתנה או פונקציה	פונקציות המכירות את המשתנה (ויכולות לגשת אליו) או את הפונקציה (ויכולות לקרוא לה)
<code>k</code> (משתנה מקומי של <code>main</code> )	כלון
<code>f</code> (משתנה מקומי של <code>f</code> )	<code>r1, r2</code> ומילוקו בתוך <code>r1, r2</code>
<code>r1</code> (משתנה מקומי של <code>r1</code> )	<code>r1, r2</code>
<code>r2</code> (משתנה מקומי של <code>r2</code> )	רקבן
<code>f</code> (פונקציה <code>f</code> )	כלון
<code>g</code> (פונקציה <code>g</code> )	כלון (בנהחה ש- <code>g</code> מוכר בתוך כל <code>main</code> גם בשורות שמופייעות לפני ההגדרה שלו)
<code>r1</code> (פונקציה <code>r1</code> )	<code>f, r1, r2</code>
<code>r2</code> (פונקציה <code>r2</code> )	<code>r1, r2</code>

נניח לדוגמה שברגע מסוים סדר הקריאה לפונקציות הוא כזה (ואף אחת מהפונקציות עד לא חזרה):  
`main` -> `f` -> `f` -> `r1` -> `r2` -> `g`

כלומר `main` קראה ל- `f` ו- `f` קראה לעצמה ברקורסיה ואז `f` קראה ל- `r1` וכן הלאה.

הנה תוכן מחסנית הקריאה ברגע זה. כאן אנו מניחים שהמחסנית גדולה לפני מטה.  
 עבור כל frame מצוין לאן מצביע ה- static link שלו ולאן מצביע ה- dynamic link.  
 בכלל הקריאה הרקורסיבית ל- `f` יש כרגע שני stack frames של `f` על המחסנית (כל אחד מהם עם עותקים נפרדים של המשתנים המקומיים, כתובות חזרה אחרת וכן הלאה). כדי להבחין ביניהם נוספו אינדקסים: `f1, f2`. שימוש לבשה static link של `r1` מצביע ל- `f1` frame של `r2` כי זה ה- frame של הקריאה המאוחרת יותר ל- `f`.

stack frame	static link	dynamic link
main		
f1	main	main
f2	main	f1
r1	f2	f2
r2	r1	r1
g	main	r2

## virtual function tables (single inheritance)

### תאoor הבעייה - dynamic dispatch

מקובל בשפות שהן Object Oriented כמו Java או C++ כאשר מפעילים על אובייקט -- הגרסה של ה- method שתופעל תלויות בטיפוס של האובייקט.

דוגמא (הקוד הוא תערובת של C++ ו- Java אבל הכוונה אמורה להיות ברורה) :

```
class Base {  
    int a, b, c;  
    int foo() { printf("calling Base::foo\n"); }  
    int bar() { printf("calling Base::bar\n"); }  
}
```

```
class Derived extends Base {  
    int d, e;  
    // override bar:  
    int bar() { printf("calling Derived::bar\n"); }  
    int lama() { printf("calling Derived::lama\n"); }  
}
```

```
void myfunction(Base b) {  
    b.foo();  
    b.bar();  
}
```

```
Base b1 = new Base();  
Derived d1 = new Derived();
```

עכשו הקריאה `myfunction(b1)` תביא לכך שבפלט יופיע

```
calling Base::foo;  
calling Base::bar
```

הקריאה (d1 myfunction() ) תביא לכך שבפלט יופיע

```
calling Base::foo;  
calling Derived::bar
```

ניתן להבחין בין הטיפוס הסטטי של `b` (הפרמטר של `myfunction`) לבין הטיפוס הדינמי שלו.

הטיפוס הסטטי הוא הטיפוס שידוע כבר בזמן קומpileציה. - במקרה זה הטיפוס הוא `Base`.  
הטיפוס הדינמי של `b` הוא הטיפוס של האובייקט ש- `b` מצביע אליו. טיפוס זה בדרך כלל לא ידוע לקומpileר והוא ידוע רק בזמן שהתוכונית רצתה. הטיפוס הדינמי של `b` הוא שקובע לאיזו גירסה של הפונקציה `bar` כאשר קוראים ל- `b.bar()` . ולכן החלטה לאיזו גרסה של `bar`LK קרווא נעשית רק בזמן ריצה. זה קוראים dynamic dispatch ("שיגור דינמי") של פונקציות. dynamic dispatch נעשית ב- Java כברירת מחדל. ב- C++ יש להזכיר על method שהוא virtual כדי שהיא תשוגר באופן דינמי.

בහמשך נקרא ל- methods המשגורות באופן דינמי "פונקציות וירטואליות" .  
זה המינוח המקובל ב- C++ (virtual functions)

מהו המנגנון שמאפשר dynamic dispatch ? לצורך כך ניתן להשתמש ב- virtual function tables (vtbl) . יש לה שמות נוספים : function tables .

עבור כל מחלקה (class) קיים בזמן ריצה virtual function table . זו טבלה שבה שמורות הכתובות של כל ה- virtual functions של המחלקה. כאשר מחלקה D יורשת ממחלקה B אז המיקומים של הפונקציות הווירטואליות בטבלה של D זהים למיקומים שלהם בטבלה של B (יתכן שבטבלה של D יהיו פונקציות נוספות שלא היו קיימות במחלקה B).

בහמשך לדוגמה שמויפהה לעלה ה- Base virtual function table יראה כך :

address of Base::foo
address of Base::bar

(הסימון Base::foo פרשו הפונקציה foo שמוגדרת במחלקה Base. הסימון שאול משפט (C++)

ה- Derived virtual function table של מחלקה יראה כך :

address of Base::foo
address of Derived::bar
address of Derived::lama

חשוב ש- `foo` ו- `bar` נמצאות במקומות קבועים בשתי הtablאות (בשתי הtablאות `foo` בכניסה הראשונה ו- `bar` בכניסה השנייה) והקומפיאילר יודע עבר כל פונקציה וירטואלית מה מיקומה בtablה -- שהרי הקומפיאילר הוא שיזכר את הtablה. באופן כללי הקומפיאילר דואג לכך שככל פונקציה וירטואלית תהיה במקומות קבועים בתABLה של מחלקה ובtablאות של כל המחלקות שיורשות ממנה שירות או בעקביפין.

בהמשך לדוגמה הנ"ל אם נגדיר מחלוקת נוספת:

```
class DD extends Derived {
    int f;
    // override foo:
    int foo() { printf("calling DD::foo\n"); }
    int lo() { printf("calling DD::lo\n"); }
}
```

אז ה- `virtual function table` של מחלוקת `DD` יהיה:

address of DD::foo
address of Derived::bar
address of Derived::lama
address of DD::lo

שים לב שהמקום של `foo` ו- `bar` כמו בשתי הtablאות הקודמות (והמקום של `lama` כמו בתABLה של `Derived`).

כל אובייקט השיך למחלוקת שיש לה פונקציות וירטואליות מכיל מצביע לtablת הפונקציות הווירטואליות של המחלוקת שלו (כאמור יש רק TABלה אחת כזאת לכל מחלוקת). המצביע הזה מאוחSEN

במקומות ידוע בתוך האובייקט למשל בתחילת האובייקט.

אובייקט מסווג נראה כך:

address of virtual function table for Base class
a
b
c

אובייקט מסוג Derived יראה כך :

address of virtual function table for Derived class
a
b
c
d
e

אובייקט מסוג DD יראה כך :

address of virtual function table for DD class
a
b
c
d
e
f

עכשו ניתן להבין איך פונקציות משוגרות באופן דינמי. למשל את הקריאה `-() bar()` כאשר `b` הparameter הפורמלי של `myfunction` הנ"ל והטיפוס (הסתטי) של `b` הוא `Base` (או `Derived`) הקומpileר מתרגם הקוד שפועל כך :

גישה לכינסה השנייה (הכינסה של `bar`) בטלת הפונקציות היררכיות שמצוין אליה מאוחסן בתחילת האובייקט `b`. שם נמצא את הכתובת של הפונקציה שיש לבצע. קופץ לשם.

הקוד הזה יעבד נכון במקרה ש- `b` מצביע לאובייקט מסוג `Base` והוא יעבד נכון גם אם `b` מצביע לאובייקט מסוג `Derived` או `DD` (או כל מחלקה אחרת שירושת מ- `Base` שירותים או בעקיפין) כי בכל הנסיבות המתאימות הכתובת של `bar` ("הכוונה") תופיע בכינסה השנייה.

בזומה לכך, קרייה ל- `()foo`.`b` תתרגם ע"י הקומpileר לקוד שニיגש לכינסה הראשונה בטבלת הפונקציות הירטואליות.

#### מחיק

המנגנון הזה הואiesel מבחינת זכרון אם כי הוא מחייב הוספת מצביע לטבלת הפונקציות הירטואליות לכל אובייקט (שבמחלקה שלו יש פונקציות וירטואליות).  
קרייה לפונקציה וירטואלית היא יקרה רק במעט יותר מאשר קרייה לפונקציה רגילה כי היא מחייבת גישה לפונקציה דרך טבלת הפונקציות הירטואליות. גם בנייתו של אובייקט חדש היא יקרה רק במעט (צריך לאחסן מצביע לטבלת הפונקציות הירטואליות בתוך האובייקט).

#### הערות

המנגנון שתואר כאן עובד עבור **single inheritance** כלומר כל מחלקה יכולה לרשף רק מחלקה אחת. כדי לתמוך ב- **multiple inheritance** כלומר במקרה שבו מחלקה אחת יורשת מספר מחלקות ניתן להשתמש במנגנון דומה אבל הפתרון קצר יותר מורכב. לעומת דומה לו- Java כי מחלקה יכולה למשם מספר `.interfaces`

המנגנון של קרייה ל- `method` באמצעות `reference` ל- `interface` שונה מהמנגנון שתואר כאן.

#### הערה על פרישה של השדות באובייקטים (single inheritance) בשפות כמו C++ או Java

השדות של אובייקט בדרך כלל מאוחסנים בתוך האובייקט לפי הסדר בו הוגדרו (ראו דוגמאות מעלה).

כאשר מחלקה D יורשת שדות ממחלקה B או באובייקט מסווג D יופיעו קודם השדות של B ולאחר מכן השדות של D.

הנה שוב המבנה של אובייקט מסווג `Derived` בזיכרון (במהלך לדוגמה שלנו) :

address of virtual function table for Derived class
a
b
c
d
e

קודם מופיעים השדות שהתקבלו בירושה מהמחלקה Base (השדות a, b, c מכון השדות שהוגדרו במחלקה Derived d, e). זה חשוב שהשדות שהתקבלו בירושה מופיעים באותו offset יחסית לתחילת האובייקט באובייקטים מסווג ובאובייקטים מסווג (או באובייקטים מכל מחלוקת אחרת שיורשת ישירות או בעקיפין מ- Base Derived).

נתבונן למשל בקוד הבא:

```
void myFunction2 (Base p) {  
    print(p.c);  
}
```

הקומפיאילר יודע שהשדה c נמצא ב- offset 12 יחסית לתחילת האובייקט וכך הוא יכול ליצור קוד שניגש ל- c ("גש לכתובת  $c + 12$ "). (בהתבהה שהכתובת של טבלת הפונקציות הוריטואליות והשדות b, a, c יתofsים כל אחד 4 בתים אז c אכן יהיה ב- offset 12).

הנקודה היא ש- c נמצא באותו offset גם אם הטיפוס הדינמי של c הוא Base וגם אם הוא Derived (או כל מחלוקת אחרת שיורשת מ- Base ישירות או בעקיפין) ולכן הקומפיאילר יוכל ליצור קוד שיעבוד בכל המקרים האלו.

## static linker

בשפות כמו C, C++ יוצר קובץ הרצה עבור תוכנית כרוכּ בביצוע הצעדים הבאים:

1. תרגום של כל קובץ C (בנפרד) לקובץ בשפת assembly -- זה נעשה ע"י הקומpileר.

2. תרגום של כל קובץ בשפת assembly (בנפרד) לקובץ בשפת מכונה -- relocatable object code assembler. את זה עשויה ה- Windows קבצים כאלו הם בעלי סיומת .obj. (על Unix ו- Linux הם בעלי סיומת .o).

3. "מייזוג" של כל הקבצים של התוכנית (המכילים relocatable object code) לקובץ הריצה אחד (בעל סיומת .exe). על Windows שם הוא כולל linker (כלומר קוד בשפת מכונה). את זה עשויה ה-

ה-linker הוא היחיד שרוואה את כל קובצי התוכנית כי הקומpileר והאסמלבר פועלם על כל קובץ בנפרד. ה-linker גם מקבל רשימה של ספריות הכלולות פונקציות (בצורה של relocatable object code).

כדי להרכיב את התוכנית, מערכת הפעלה תתען את קובץ הריצה מהדיסק לזכרון הראשי. בפועל יתכן שתתען רק חלקים מהתוכנית לזכרון הראשי וחלקים נוספים (דפים) יטענו בהמשך ריצת התוכנית לפי דרישת הלומר כאשר התוכנית תשתמש בהם. אבל זה עניין של מערכת הפעלה.

כאשר עובדים עם סביבת פיתוח (למשל Visual Studio) ו- "מקמפלים" assembly את התוכנית או מבצעים בעצם את כל הצעדים הניל (קימפול, linking) אם כי אפשר לבצע רק חלק מהצעדים למשל לייצר .object code בלי לייצר assembly code.

## code and data segments

כל קובץ הכלל object code כולל בתוכו code segment ו- data segment. ה- code segment כולל את פקודות התוכנית בשפת מכונה. ה- .data segment כולל data segment

למשל נתבונן בתוכנית

```
int global = 17;
void main() {
    if (global > 0)
        printf("hello world\n");
}
```

או ה- data segment יכול במקרה זה בין היתר את הערך 17 (שמשם לאייחול משתנה גלובלי) ואת המחרוזת "hello world".

## מה עושה ה- linker

ה- linker יוצר את ה- code segment של קובץ ההרצה ע"י שרשרו ה- code segments של הקבצים להם הוא עושה linking. דבר דומה הוא עשוועה עם ה- .data segments

ה- linker עושה relocation ("מיקום מחדש")

ה- linker עושה external symbol resolution

## relocation

כאשר ה- assembler מייצר קוד בשפת מכונה -- הקוד כולל כתובות יחסיות לתחילת ה- code segment (או יחסית לתחילת ה- data segment אבל בהמשך נתיחס רק ל- code segment -- די בכך כדי להבין את העיקרו).

למשל אחת הפקודות (בשפת assembly) יכולה להיות `JUMP label42` כלומר קבוע לתווית `label42`. ה- assembler יתרגם פקודה זאת לפקודה המתאימה בשפת מכונה. נניח לדוגמה שפקודה זאת תופיע הכתובת 150 (קבוע לככתובת 150) כי ה- assembler ימפה את `label42` לככתובת 150. הכתובת 150 היא כתובה יחסית לתחילת ה- code segment של הקובץ עליועובד ה- assembler (נקרא לו `a`).

נניח לדוגמה שה- code segment של a יופיע ב- offset 3000 ב- linker segment של קובץ ההרצה אותו בונה ה- . (SEGMENT זה מורכב מהSEGMENT של a וSEGMENTים של קבועים נוספים להם נעשה linking).

ה- linker ישנה את הכתובת 150 לכתובת  $3000 + 150 = 3150$ . זה נקרא relocation.

ה- assembler מצרף לקובץ שהוא מייצר מידע Shiapshar linker לאתר את כל הכתובות להם יש לעשות relocation. המידע הזה אומר דברים כמו: בכתובות 1000, 2200, 1700 יש לעשות relocation. לעיתים המידע הזה מאורגן

כ- relocation bit map: ה- bit map הוא "מפה" של כל ה- code segment: הסיבית הראשונה ב- bit map מייצגת את המילה הראשונה בסegment, הסיבית השנייה מייצגת את המילה השנייה וכן הלאה. סיבית דלוכה משמעותה שבמילה המתאימה בסegment נמצאת כתובות שיש לעשות לה relocation.

### external symbol resolution

בתוכנית C המורכבת ממספר קבועים, בקובץ אחד יכולה להופיע קרייה לפונקציה המוגדרת בקובץ אחר והוא יכול גם לגשת למשתנה גלובלי המוגדר בקובץ אחר.

שמות של פונקציות ומשתנים גלובליים כאלו מכונים בהקשר של ה- linker external symbols --.

ה- assembler שנטקל ב- external symbols לא יכול להחליפם בכתובות כי הוא לא יודע מה תהיינה הכתובות (הרוי הוא עובד בכל פעם על קובץ אחד ולא רואה כרגע את הקבצים בהם מוגדרים ה- external symbols). זה תפקידו של ה- linker "לפטור" את ההתייחסויות ל- external symbols כלומר להחליפם בכתובות.

ה- assembler מייצר טבלת סמלים (external symbol table) עבור כל קובץ שהוא מתרגם ומצרף אותה לקובץ שהוא יוצר (עם ה- relocatable object code).

## הנה דוגמא ל- external symbol table

external symbol	type	address
main	entry point	100 code
foo	entry point	480 code
printf	reference	500 code
bar	reference	800 code
my_global	entry_point	40 data

יש שני סוגי של כניסה בטבלה: reference ו- entry point.

כניסה מסוג entry\_point מתייחסות לשמות של פונקציות או משתנים גלובליים המוגדרים בקובץ הנוכחי (הקובץ שטבלת הסמלים נמצאת בו).

בדוגמה רואים שהפונקציה main מוגדרת בקובץ והקוד שלה מתחילה בכתובת 100 ב- code segment (100 יחסית לתחילת הsegment).

גם הפונקציה foo מוגדרת בקובץ והקוד שלה מתחילה בכתובת 480 ב- code segment.

משתנה my\_global הוא משתנה גלובלי שמואוחסן ב- data segment בכתובת 40.

ה כניסה מסוג reference מתייחסים לשמות של פונקציות או משתנים גלובליים שהקובץ הנוכחי מתייחס אליהם אבל כתובותם לא ידועה (כי הם מוגדרים בקובץ אחר או על כל פנים ה- assembler מקווה שהם מוגדרים בקובץ אחר ...)

בדוגמה רואים שיש התייחסות ל- printf בכתובת 500 ב- code segment (כנראה שיש שם קפיצה לקוד של הפונקציה printf).

יש התייחסות ל- bar בכתובת 800 ב- bar .code segment (יכול להיות פונקציה או משתנה גלובלי).

ה-linker עובר על כל טבלאות הסמלים של הקבצים להם הוא עושה reference והוא מחליף כל reference לא פטור בכתובת המתאימה.

למשל הוא "פותר" את ההתייחסות ל- bar הניל ע"י חיפוש של entry point עם השם bar בטבלאות הסמלים של הקבצים האחרים. נניח שכניתה של entry point bar נמצאת בטבלת הסמלים של קובץ אחר

(נקרא לו c) ושם רשום ש-`bar` מוגדר בכתובת 600 יחסית לתחילה -`code` (של c). עוד נניח שה-`offset` של ה-`code segment` של c ב-`code segment` של קובץ ההרצה הוא 3000. מכיוון ה-`linker` מסיק שהכתובת של `bar` היא  $3000 + 600 = 3600$  לכן במקום שיש התייחסות ל-`bar` ה-`linker` יכתוב 3600.

חלק מה-`external references` נפתרים כאשר ה-`linker` מוצא את השם המתאים בספריה. ספריה כוללת `object code` של פונקציות (או משתנים גלובליים). ה-`linker` מקבל רשימה של ספריות בהן הוא אמור להשתמש -- זה נשלט ע"י המתכנת. בשיעורים linking לתוכניות C או הספריה הסטנדרטית של C (শכללת למשל את הפונקציות `printf` ו-`strcpy`) תהיה כלולה ברשימה אבל ניתן להוסיף ספריות נוספות בהתאם לצורך. המתכנת יכול לשולוט על הסדר שבו נעשו הקבצים אחרים של התוכנית ובספריות כאשר ה-`linker` מחפש external symbol (הסדר לפעמים חשוב כי אותו שם עשוי להיות מוגדר במספר מקומות).

במקרה שה-`linker` לא מוצא הגדרה של פונקציה (או משתנה גלובלי) שיש אליה external reference על כך הודעת שגיאיה.

#### דוגמא בקובץ נפרד (גס ב- moodle)

הדוגמה נלקחה מהספר Modern Compiler Design 2<sup>nd</sup> Edition by Grune et. al Unix בדוגמה עושים linking לשני קבצים a.o ו- b.o (כאמור קבצים עם סיוםת o. ב- Unix כוללים relocatable object code).

בדוגמה רואים רק את ה-.code segments. זה של a.o גודלו 1000 בתים וזה של b.o גודלו 3000 בתים. בדוגמה ה-`linker` בנה את ה-`code segment` של קובץ ההרצה ע"י שרשור של הסגמנטים של a.o ו- b.o בסדר זה (ובהמשך הסegment של a.o).`printf.o`

לכן ה-`offset` של הסegment של a.o בסegment הממזוג הוא 0 וזה של הסegment של b.o הוא 1000. לכן ה-`linker` עשה relocation לכתובות היחסיות של b.o ע"י הוספה של 1000 לכל כתובת. למשל כתובת 1600 הפכה לכתובת 2600.

בסגמנט של o.a הופיע reference לשם printf או אולי הקומpileר שינה את השם ל \_printf (הוסיף קו תחתון בהתחלה) אבל לא נכנס לוזה כאן).

ה-linker חיפש את ההגדרה של printf ומצא אותה בספריה הסטנדרטית של C. הוא צרעף ל-code segment של קובץ ההרצה את ה-code segment של printf.o שנמצא בספריה. שמו לב שהוא לא צירף את הקוד של כל הפונקציות שנמצאות בספריה אלא רק את הקוד של הפונקציה שהוא נזקק לה.

ה-linker עשה את החישוב הבא:

ה-code segment של printf.o מתחילה ב-offset 4000 בסגמנט של קובץ ההרצה. הקוד של printf מתחילה ב-offset 100 יחסית לתחילת הסגמנט של printf.o (המידיע זהה נמצא בספריה). לכן הכתובת של printf היא  $4000 + 100 = 4100$ .  
לכן במקום שיש התיחסות ל-linker בסגמנט של o.a ה-linker כתוב 4100.

### dynamic linker

ה-linker שתואר כאן נקרא static linker כי הוא פועל לפניהם שהתוכנית רצה.  
dynamic linker לעומת זאת פועל בזמן שהתוכנית רצה. בפעם הראשונה שתוכנית רצה (תחיליך) תקרה לפונקציה (לה עושים dynamic linking) -- יבוצע linking של הקוד של הפונקציה לקוד של התחיליך הרץ. יש לכך מספר יתרונות. אחד מהם הוא חיסכון בשטח איחסון על הדיסק.

למשל אם עושים static linking לфункция printf או כל קובץ הרצה שמשתמש בה יכול עותק של הקוד של printf. לעומת זאת אם משתמשים ב-dynamic linking או ניתן להזין רק עותק אחד של הקוד של printf על הדיסק. (ואם מספר תהליכי רצים משתמשים ב-printf אז בעיקרן ניתן להזין רק עותק אחד של printf בזיכרון).

בנוסף לכך אם רוצים לשדרג את printf לגרסה משופרת (printf נראה לא דוגמא טובה מבחינה זאת כי משדרגים אותה לעיטים רוחוקות אם בכלל) אז ניתן לשדרג את העותק הבודד שנמצא על הדיסק במקום לעשות static linking חוזר לכל קובצי הרצה שצריכים להשתמש בגרסה המשופרת.

נכתב 22 יוני 2018

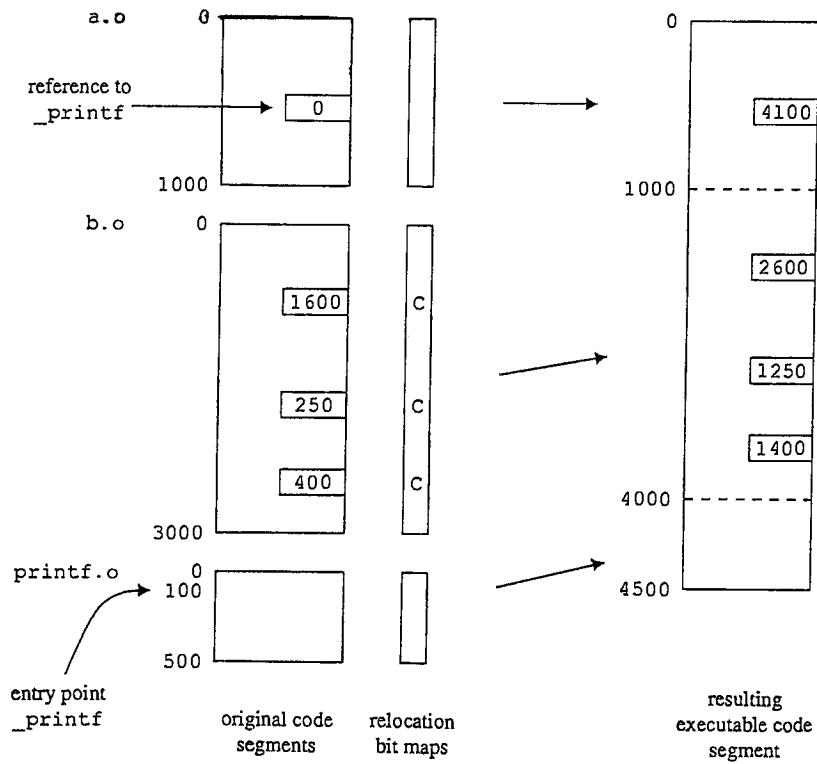


Fig. 8.1: Linking three code segments

External symbol	Type	Address
options	entry point	50 data
main	entry point	100 code
printf	reference	500 code
atoi	reference	600 code
printf	reference	650 code
exit	reference	700 code
msg_list	entry point	300 data
Out_Of_Memory	entry point	800 code
fprintf	reference	900 code
exit	reference	950 code
file_list	reference	4 data

Fig. 8.4: Example of an external symbol table

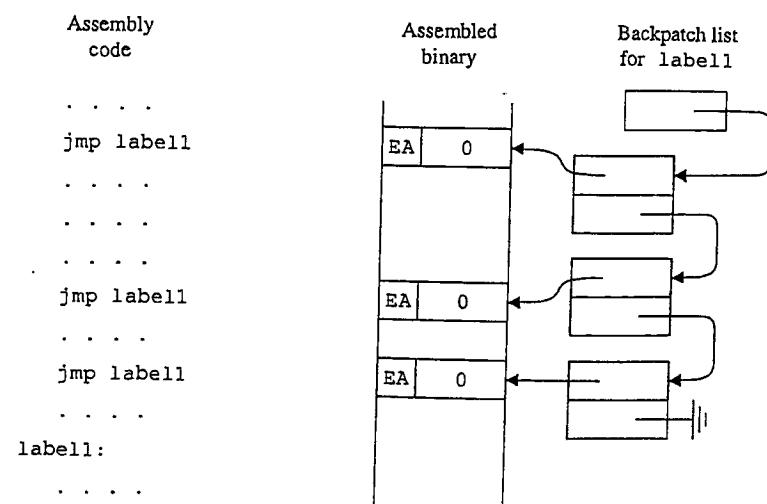


Fig. 8.3: A backpatch list for labels

## חלקים שונים של הקומפיילר מגלים שגיאות מסווגים שונים

המנתח הלקסיקלי (שתי קידו לזהות אסימונים בקלט) מוציא הודעת שגיאה אם בקלט מופיעים סימנים שאינם מזוהים כאיסימון חוקי.

למשל אם בתוכנית C יופיע  $a = b + c \$$

או כשהמנתח הלקסיקלי יבחן בסימן  $\$$  שאינו איסימון חוקי הוא יוציא הודעת שגיאה.ysisical יש בקלט הערת שנסתחה ולא נסגרה, המנתח הלקסיקלי הוא שיבחין בכך. (בשפות רבות, המנתח הלקסיקלי הוא החלק היחיד של הקומפיילר שראה את ההערות. הוא אינו מעביר אותן להלאה -  
.parser

במקרה שהקלט אינו מתאים לדקדוק, או במקרים אחרות, כאשר בקלט יש syntax error -- ה- parser הוא שיבחין בכך ויזיא הודעת שגיאה. הנה 2 דוגמאות ל- syntax errors בשפה C :  
 $c = a + b$  (כאן הבעיה היא שחרר נקודת פסיק בסוף המשפט כי שכלי הדקדוק מחיבים).  
דוגמא נוספת :

while (a == b); else y = 0;

בשורה האחורונה הבעה היא שמופיע else מבלי שיש if תואם. גם זה בנייגוד כללי הדקדוק.

המנתח הסמנטי מגלה שגיאות סמנטיות בתוכנית. מדובר בהפרה של כללים שונים של שפת התכנות

(כללים שאינם נוגעים ל- syntax ולא להגדלה של אסימונים חוקיים).

דוגמאות לשגיאות סמנטיות :

-- שימוש במשתנה מבלי שהוגדר קודם

-- הגדלה של אותו משתנה פעמיים באותו ה- scope.

-- גישה לשדה שהוגדר private מקום בו הגישה הזאת אסורה למשל ב- Java. (סביר

שב- Java טעות זאת מתגלית לא בזמן הקומpileציה אלא מאוחר יותר, בזמן ריצה, אם אין טעה)

-- קריאה לפונקציה עם מספר שגוי של ארגומנטים.

-- קריאה לפונקציה עם ארגומנט מטיפוס לא נכון (למשל אם הפונקציה הוכרזה כ-

`void foo(int);` ומעבירים ל- `foo` מצביע כארוגמנט).

-- הפעלה של אופרטור על אופרנדים מטיפוס לא מתאים. למשל `3 * k` זו שגיאה סמנטית אם

`k` הוגדר כמצביע כי לא ניתן (בשפה C) להפעיל את האופרטור \* על מצביע (הכוונה לאופרטור הבינרי \* המופיע כפלו).

בדיקות שהתוכנית חוקית מבחינת השימוש בטיפוסים נקראות type checking. ה- checker הוא חלק מהמנתח הסמנטי. שלושת הדוגמאות האחורונות שהוזכרו לעלה (קריאה לפונקציה עם מספר שגוי של ארגומנטים או עם ארגומנטים מטיפוס לא מתאים והפעלה של אופרטור על אופרנדים מטיפוס שגוי) הן שגיאות שמתגלות במהלך ה- type checking.

משמעותו של שפה תכנות (למשל Python) בבחן הקומpileר לא עשו type checking וטיעות שקשורת לTYPESIMS מתוגלו רק בזמן ריצה. זה חסרונו של Python. (באופן כללי עדיף שהקומpileר יגלה כמה שיותר שגיאות אם ישן כלו).

בנוסף לסוגי השגיאות שהוזכרו, יתכנו גם סוגים נוספים של שגיאות.  
למשל, בשפת C אם עושים #include "foo.h" ולא קיים קובץ foo.h או ה-preprocessor הוא שיגלה את השגיאה.

אם בתוכנית מופיע ביטוי כמו  $(a+b)/c$  ובזמן שהקומpileר מנסה את התוכנית לצורכי אופטימיזציה הוא מגלח שהחסכום של  $a+b$  חייב להיות 0 אז הקומpileר יכול להוציאו על כך הודעה שגיאה. (אבל בדרך כלל חלוקה באפס תוגלה רק בזמן ריצה).

אם בקובץ C מסוים יש קריאה לפונקציה (שהחтиימה שלה הוכרזה קודם כנדרש לפי כללי C) אבל הפונקציה לא מוגדרת באף מקום אז לא הקומpileר אלא ה-linker הוא שיגלה את השגיאה.

יש שגיאות שהקומpileר (למרבה הצער) לא יכול לגלו. למשל התיאוריה אומרת שלא ניתן באופן כללי לגלו את כל הלוואות האינסופיות. (הבעיה "האם קוד נתון מכיל לולאה אינסופית?" היא בעיה שאינה ניתנת להכרעה). שגיאות בלוגיקה של התוכנית גם הן לא מוגלוות ע"י הקומpileר. כדי, יש לא מעט שגיאות שמוגלוות רק בזמן ריצה.

הערה: השגיאות הן תלויות שפת תכנות. לכל שפת תכנות יש את האיסימונים שלה, הדקדוק שלה, הכללים הסמנטיים שלה ...