גירסה 1.00 – 24.3.2009

תורת הקומפילציה

ניר אדר

1. תוכן עניינים

1. תוכן עניינים 2

2. פתיחה 5

3. אוטומטים ושפות פורמליות – חזרה 6

3.1. שפות פורמליות – מושגים בסיסיים 6
3.2. ביטויים רגולריים, שפות רגולריות 7
3.3. אוטומט סופי 8
3.3.1. תיאור לא פורמלי והגדרה 8
3.3.2. אופן תיאור אוטומט סופי (דטרמיניסטי) 9
3.3.3. אוטומט סופי לא דטרמיניסטי 10
3.3.4. זיהוי שפות רגולריות – על ידי אוטומט סופי 10
3.3.5. משפטים חשובים 10
3.4. דקדוקים חסרי הקשר 11
3.4.1. דקדוק חסר הקשר – הגדרה פורמלית 11
3.4.2. Parse Tree (עץ גזירה) 12
3.4.3. שפה חסרת הקשר 12
3.4.4. תבניות BNF 13

4. מבוא לקומפילציה 15

4.1. התמונה הגדולה 15
4.2. שלב האנליזה 16
4.2.1. ניתוח לקסיקלי (lexical analysis) 16
4.2.2. ניתוח סינטקטי (syntax analysis) 17
4.2.3. ניתוח סמנטי 17

5. שלב 1 - ניתוח לקסיקלי (LEXICAL ANALYSIS) 18

5.1. מושגים בסיסיים 18
5.2. SCANNER ו-SCREENER 19
5.3. המנתח הלקסיקלי ותכונות סמנטיות 20
5.4. כלי LEX 21
5.4.1. רקע כללי 21
5.4.2. פונקצית המנתח הלקסיקלי yylex() 21
5.4.3. קבצי קלט ופלט למנתח הלקסיקלי 22
5.4.4. משתנים גלובלים נוספים 22
5.4.5. מבנה קובץ הקלט ל-lex 22
5.5. פתרון קונפליקטים 25
5.6. LOOKAHEAD 25

6. שלב 2 – ניתוח תחבירי (SYNTAX ANALYSIS) 26

6.1. רקע ומוטיבציה 26
6.2. סוגי ניתוח תחבירי 27

6.2.1. ניתוח Top-Down 28
6.3. שיטת Recursive Descent (פועלת בשיטת Top-Down) 29
6.3.1. קבלת עץ גזירה באמצעות RD 31
6.3.2. בעיות בשיטת RD 31
6.3.3. שיפור בפונקציית הבחירה – select, first, follow 34
6.4. אלגוריתם LL(k) 36
6.4.1. רקע 36
6.4.2. אלגוריתם LL(1) 37
6.5. סיכום שיטות Top-Down 38
6.6. אלגוריתמים LR(K) 39
6.6.1. אלגוריתם LR(0) 40
6.6.2. SLR 54
6.6.3. Canonical LR 57
6.6.4. LALR 58
6.7. היררכיית הדקדוקים וסיכום 59

7. שלב 3 - ניתוח סמנטי 60

7.1. פתיחה 60
7.2. דקדוקי שדות ערך / הגדרות מונחות תחביר 61
7.2.1. הגדרות בסיסיות 61
7.2.2. דוגמא להגדרה מונחית תחביר 62
7.2.3. סיווג תכונות סמנטיות לפי אופן חישובן 63
7.3. אלגוריתמים לחישוב הגדרות מונחות תחביר 64
7.3.1. סוגי הגדרות מונחות תחביר 64
7.3.2. סכימת תרגום 64
7.3.3. אלגוריתם כללי לחישוב הגדרות מונחות תחביר 66

8. YACC 67

8.1. מבנה קובץ הקלט ל-Yacc 67
8.2. חלק החוקים 68
8.3. קישור בין המנתח הלקסיקלי למנתח התחבירי 71
8.4. תכונות סמנטיות ב-YACC 71
8.5. קדימויות אופרטורים 72
8.5.1. דוגמא לבעיה 72
8.5.2. הגדרת הקדימויות 73
8.5.3. איך מערכת הקדימויות עובדת? 74
8.6. הפעלת YACC 74

9. בניית טבלאות הסמלים 75

9.1. Scoping 75
9.2. ארגון הזכרון על ידי המהדר 76
9.3. טבלת הסמלים 77
9.4. סכימת תרגום 78
9.4.1. מבוא 78
9.4.2. דוגמא 79

10. תרגום לשפת ביניים 81

10.1. מבוא 81
10.2. שפת הרביעיות 82
10.3. יצירת קוד הביניים – תרגום לשפת הרביעיות 83
10.3.1. תרגום פשוט 84
10.3.2. תרגום לשפת הביניים - שיטת תוויות נורשות 86
10.3.3. Backpatch 89
11. מקורות 93
11.1. מקורות באנגלית 93
11.2. מקורות בעברית 95

2. פתיחה

מסמך זה מסתמך על מקורות רבים, ובא להציג לקורא את עולם הקומפיילרים. המבנה הכללי של המסמך מנסה לשמור על הסדר המוצג בספר "Compilers – Principles, Techniques and Tools" מאת Aho, Sethi and Ullman. בנוסף המסמך בנוי במידה רבה מאוד גם מסביב לסדר ההוראה בקורס "תורת הקומפילציה" בטכניון. חשוב לציין כי מדובר בסיכום אישי של המחבר, ולא בחומר רשמי של הקורס שאושר על ידי צוות הקורס. סטודנטים בקורס צריכים לשים לב כי המסמך עלול להכיל טעויות ואי דיוקים. כמו כן המסמך אינו מכיל את כל הנושאים הנסקרים בקורס. מקור עיקרי נוסף של מסמך זה הוא ויקיפדיה האנגלית. להעמקה אני ממליץ בחום על קריאת הערכים הרלוונטים בויקיפדיה האנגלית – הכותבים של הערכים הרלוונטים עשו עבודה נפלאה וההסברים ברורים ביותר. רוב הדוגמאות במסמך זה נלקחו מויקיפדיה.

רשימת המקורות המלאה מצורפת בסוף מסמך זה לעיונכם. ברצוני להודות לצוות הקורס 2009 של תורת הקומפילציה: **מר גלעד קוטיאל**, **מר דניאל גנקין ומר גדי אלכסנדרוביץ'** על התשובות הרבות שענו לי לשאלות במהלך למידת החומר. תובנות רבות שסופקו לי במייל ובשיחות הוכנסו למסמך זה.

ולקוראים אאחל: בהצלחה בהכרת עולם הקומפיילרים! אשמח לשמוע אם דפים אלה עזרו לכם, וכן לקבל עדכונים על טעויות והצעות לשיפור.

ניר אדר

מרץ 2009

3. אוטומטים ושפות פורמליות – חזרה

3.1 שפות פורמליות – מושגים בסיסיים

- **אלף בית** הינה קבוצה סופית Σ (של "אותיות").
- **מילה** הינה סדרה סופית של אותיות מעל א"ב נתון Σ.
- **המילה הריקה** (מסומנת ב-ε) איננה מכילה אותיות.
- **אורכה של מילה** יוגדר בצורה הבאה: תהי $w=\sigma_1...\sigma_n$, $n\geq 0$, אזי $|w|$, אורך w , הינו n.

  אורך המילה הריקה ייקבע להיות: $|\varepsilon|=0$.
  - Σ^0 - אוסף כל המילים באורך 0 מעל ה-א"ב Σ.
  - Σ^n - אוסף כל המילים באורך n מעל ה-א"ב Σ.
  - Σ^* - אוסף כל המילים מעל ה-א"ב Σ.
  - Σ^+ - אוסף כל המילים באורך 1 או יותר מעל ה-א"ב Σ.
- **דגש:** אם Σ איננה ריקה, מתקיים כי Σ^* אינסופית (בת מניה). אין חסם סופי על אורך המילים ב-Σ^*, אולם אין מילים באורך אינסופי ב-Σ^*.
- תהי Σ קבוצה סופית. נגדיר **שפה פורמלית** L להיות תת קבוצה כלשהי של Σ^*, כלומר $L\subseteq\Sigma^*$. שפה יכולה להיות **סופית** או **אין סופית**.
- שרשור:
  - **שרשור מילים** הינה המילה הנוצרת מהדבקת שתי מילים w_1,w_2 (בסדר זה). פעולת השרשור מסומנת על ידי $\bullet$, אם כי נהוג לוותר לרוב על סימון הפעולה. יהיו המילים $w_1=\sigma_1...\sigma_n$, $w_2=\sigma_1'...\sigma_n'$ אזי השרשור שלהם, המילה w, תוגדר להיות:
    $$w=w_1\bullet w_2=\sigma_1...\sigma_n\sigma_1'...\sigma_n'$$
  - תהי Σ קבוצה סופית. יהיו $L_1,L_2\in\Sigma^*$ שפות. **השרשור** בין L_1,L_2 יוגדר כך:

$$L_1\bullet L_2=\left\{w\in\Sigma^*\middle|\exists x\in L_1\quad\exists y\in L_2:\quad w=xy\right\}$$

3.2 ביטויים רגולריים, שפות רגולריות

נגדיר ביטויים רגולריים על ידי הקבוצה האינדוקטיבית הבאה:

- בסיס:
  - $\varnothing$ הוא ביטוי רגולרי מעל Σ המתאר את השפה הריקה.
  - ε הוא ביטוי רגולרי מעל Σ המתאר את השפה $\{\varepsilon\}$.
  - לכל $a \in \Sigma$ מתקיים כי a הוא ביטוי רגולרי המתאר את השפה $\{a\}$.
- אם p, q הם ביטויים רגולריים המתארים את השפות הרגולריות P, Q, אזי:
  - $p \mid q$ הוא ביטוי רגולרי המתאר את השפה הרגולרית $P \cup Q$.
  - (pq) הוא ביטוי רגולרי המתאר את השפה PQ.
  - $(p)^*$ הוא ביטוי רגולרי המתאר את P^*

<u>הערה</u>: על מנת לפשט את הסימונים, מגדירים כי * הוא בעל הקדימות הגבוהה ביותר, אח"כ שרשור, ואח"כ סימן האלטרנטיבה.

3.3 אוטומט סופי

3.3.1 תיאור לא פורמלי והגדרה

אוטומט סופי הינו מודל מופשט למערכת המגיבה על קלטים (מתוך קבוצה סופית). תגובת האוטומט: קבלה / דחייה. ניתן לזהות את אוסף סדרות-הקלט המתקבלות עם שפה פורמלית. אוטומט סופי הינו בעל מספר סופי של מצבים שאינו תלוי באורך הקלט. אורך הקלט איננו בהכרח חסום.

אופן הפעולה של האוטומט: בכל שלב, נמצאת המערכת במצב q (מתוך קבוצת מצבים סופית Q).
המצב התחילי מסומן q_0. בכל צעד, קורא הראש את אות הקלט הנוכחית, האוטומט עובר למצב q' (כאשר יתכן כי $q = q'$), והראש עובר לאות הקלט הבאה.

המצב אליו מגיעה המערכת לאחר קריאת האות האחרונה בקלט קובע את התגובה למילת הקלט: מצב מקבל / דוחה.

אוטומט סופי (דטרמיניסטי) A נתון ע"י: $A = \langle \Sigma_A, Q_A, q_{0A}, F_A, \delta_A \rangle$, כאשר:

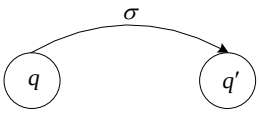
- Σ_A הינה קבוצה סופית, א"ב הקלט.
- Q_A הינה קבוצה סופית לא ריקה של מצבים.
- q_{0A} הינו המצב ההתחלתי.
- F_A היא קבוצת המצבים המקבלים, $F_A \subseteq Q_A$
- δ_A היא פונקצית המעברים, $\delta_A : Q_A \times \Sigma_A \to Q_A$.
  - ב-δ_A אין מעברי-ε.
  - לכל מצב $q \in Q_A$ ואות $a \in \Sigma_A$ יש לכל היותר מעבר למצב יחיד.

אם זהות A ברורה מן ההקשר נשמיט אותה כאינדקס.

F_A יכולה להיות קבוצה ריקה. Q_A איננה ריקה – לפחות q_{0A} שייך אליה.

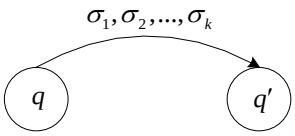
3.3.2. אופן תיאור אוטומט סופי (דטרמיניסטי)

נתאר אוטומט סופי על ידי גרף סופי מכוון, שצמתיו מצבים וקשתותיו מסומנות ע"י אותיות Σ.



אם האוטומט נמצא במצב נוכחי q, ואם אות הקלט הנוכחית היא σ, אזי האוטומט עובר למצב q' (ומקדם את הקלט).

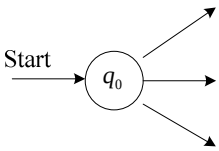
נניח כי ממצב q הקלטים $\sigma_1,...,\sigma_k$ מעבירים אותנו אל מצב q', נוכל נשרטט זאת בקיצור בצורה הבאה:



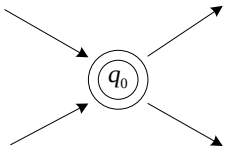
וזאת במקום לצייר k קשתות מקבילות מ-q אל q'.

הערה חשובה: מכל מצב יש בדיוק קשת אחת מסומנת σ, לכל $\sigma \in \Sigma$.

המצב ההתחלתי מסומן על ידי q_0 ועל ידי $start$:



מצב מקבל מסומן על ידי מעגל כפול. מצב מקבל אינו חייב להיות המצב האחרון. מעבר במצב מקבל תוך כדי תהליך החישוב לא אומר כלום על תוצאת החישוב.



3.3.3. אוטומט סופי לא דטרמיניסטי

אוטומט סופי לא דטרמיניסטי הוא הכללה של אוטומט סופי דטרמיניסטי המאפשרת לאוטומט לבחור במספר דרכי פעולה אפשריות עבור קלט נתון בניגוד לדרך הפעולה היחידה אליה מחויב אוטומט דטרמיניסטי.

ההכללה של האוטומט הסופי הדטרמיניסטי מתבטאת בשלוש הרחבות עיקריות:

1. עבור כל מצב של האוטומט ואות קלט נתונה, האוטומט הלא דטרמיניסטי יכול לעבור למספר מצבים, ולא למצב יחיד כאוטומט הדטרמיניסטי.
2. לאוטומט מוספת האפשרות של "**מסע** ε" - מעבר ממצב אחד למשנהו מבלי שתיקלט אות קלט כלשהי.
3. לאוטומט מוספת האפשרות לבחור בין מספר מצבים התחלתיים.

3.3.4. זיהוי שפות רגולריות – על ידי אוטומט סופי

1. הצב וקבל סדרת ביטויים רגולריים טהורים.
2. בנה אוטומט לא דטרמיניסטי Mi לכל Ai המזהה את הביטוי הרגולרי Ri. קיים אלגוריתם שיטתי המבצע זאת.
3. בנה אוטומט משולב M אוטומט זה מזהה את כל הביטויים הרגולריים.

3.3.5. משפטים חשובים

1. לכל אוטומט סופי לא דטרמיניסטי קיים אוטומט דטרמיניסטי שקול.
2. לכל אוטומט סופי דטרמיניסטי קיים אוטומט סופי דטרמיניסטי שקול יחיד בעל מספר מצבים מינימאלי.

3.4 דקדוקים חסרי הקשר

במודל זה כל צעד בנוהל ליצירת מילים הינו שכתוב חלק מהמילה על פי קבוצת כללי שכתוב מוגדרים היטב. התהליך מפסיק כאשר מגיעים **למילה טרמינלית**, כלומר מילה שלא ניתן לשכתב אותה עוד.

נשתמש בסימן $\rightarrow$ לציון כללי שכתוב, ובסימן $\Rightarrow$ לציון הפעלת כללי שכתוב על מילים.

ניתן לציין את יצירת המילה בדרך גרפית על ידי **עץ גזירה**.

3.4.1 דקדוק חסר הקשר – הגדרה פורמלית

דקדוק חסר הקשר G מוגדר ע"י רביעיה $G=(V,T,P,S)$ כאשר:

- V קבוצה סופית לא ריקה של משתנים דקדוקיים.
- T קבוצה סופית לא ריקה של סימנים טרמינליים הזרה ל- V.
- S משתנה התחלתי, $S\in V$.
- P קבוצה סופית של כללי שכתוב (או כללי גזירה) מהצורה $A\rightarrow\alpha$ כאשר $\alpha\in(V\cup T)^*$ ו- $A\in V$.

מוסכמות סימון:

- נשתמש באותיות לטיניות גדולות מתחילת הא"ב כגון A,B,C לציון **סימנים קבועים** מתוך קבוצת המשתנים הדקדוקיים V.
- נשתמש באותיות לטיניות קטנות מתחילת הא"ב כגון a,b,c לציון **סימנים קבועים** מתוך קבוצת הטרמינלים T.
- נשתמש באותיות יווניות קטנות כגון α,β,γ לציון מילים מ-$(V\cup T)^*$.

3.4.2 Parse Tree (עץ גזירה)

Parse Tree (עץ גזירה) מראה בצורה ציורית איך הסימן הפותח של דקדוק חסר הקשור גוזר מחרוזת בשפה. אם למשתנה לא טרמנלי A קיים כלל הגזירה $A \to XYZ$ אז העץ יכול להראות כשורש בעל תווית A עם 3 בנים המסומנים ב- X, Y, Z משמאל לימין.

בצורה פורמלית, בהנתן דקדוק חסר הקשר, עץ הביטוי הוא עץ בעל התכונות הבאות:

1. שורש העץ מסומן על ידי המשתנה ההתחלתי.
2. כל עלה מסומן על ידי טרמינל או על ידי ε.
3. כל צומת פנימי מסומן על ידי משתנה.
4. אם A הוא משתנה שהינו תווית של צומת פנימי ו- $X_1, X_2, ..., X_n$ הם התוויות של הבנים של אותו צומת משמאל לימין, אז $A \to X_1 X_2 ... X_n$ הוא כלל גזירה כאשר $X_1, X_2, ..., X_n$ הם משתנים או טרמינלים. בנוסף, אם $A \to \varepsilon$ אז צומת המסומן A יכול להיות בעל בן יחיד שיסומן ב-ε.

כלל-ε הוא כלל מהצורה $A \to \varepsilon$.

3.4.3 שפה חסרת הקשר

- **גזירה** – סדרה של החלפות של אותיות לא טרמינליות תוך שימוש בחוקי הגזירה
- **שפה** – אוסף ביטויים הנגזרים מהמצב התחילי והמכילים טרמינלים בלבד
- **תבנית פסוקית (sentential form)** – תוצאת סדרת גזירות בה נותרו (אולי) לא-טרמינלים
- **גזירה שמאלית** – גזירה בה מוחלף בכל שלב הסימן השמאלי ביותר (באופן דומה – אפשר להגדיר גם **גזירה ימנית**)

3.4.4. תבניות BNF

תבניות BNF הן תבניות פורמליות המתארות דקדוקים חסרי הקשר. BNF נמצא בשימוש רחב לתיאור דקדוקים של שפות תכנות, פרוטוקולי תקשורת וחלקים של שפות טבעיות.

דוגמא – BNF המתאים למספר שלם (integer):

```
<digit>            ::=   "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9"
<sign>             ::=   "+"|"-"
<unsigned_integer> ::=   <digit>|<digit><unsigned_integer>
<integer>          ::=   <unsigned_integer>|<sign><unsigned_integer>
```

סימונים:

- **הסימון ::=** מציין "מוגדר כ".
- **סוגריים משולשות** (<>)
  - אם ביטוי מוקף בסוגריים משולשות באגף שמאל, זהו השם המוגדר כעת.
  - אם ביטוי מוקף בסוגריים משולשות באגף ימין, זה שימוש בשם שכבר הוגדר.
- **סימנים קבועים** מופיעות בין מרכאות.

קיצורי כתיבה:

- **הסימן |** פירושו "או".
- **שרשור פריטים:** כתיבת פריטים אחד ליד השני

התחביר של BNF עצמה יכול להיות מיוצג גם הוא ע"י BNF:

```
<syntax>         ::= <rule> | <rule> <syntax>
<rule>           ::= <opt-whitespace> "<" <rule-name> ">" <opt-whitespace> "::="
                     <opt-whitespace> <expression> <line-end>
<opt-whitespace> ::= " " <opt-whitespace> | ""  <!-- "" is empty string, i.e. no whitespace -->
<expression>     ::= <list> | <list> "|" <expression>
<line-end>       ::= <opt-whitespace> <EOL> | <line-end> <line-end>
<list>           ::= <term> | <term> <opt-whitespace> <list>
<term>           ::= <literal> | "<" <rule-name> ">"
<literal>        ::= '"' <text> '"' | "'" <text> "'" <!-- actually, the original BNF did not use quotes -->
```

הרחבות ל-BNF: ל-BNF הרחבות ווריאנטים רבים. הרחבה מאוד נפוצה היא EBNF. ניתן לקבל מידע על הרחבה זו בקישור הבא: http://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_Form

הרחבות בהן נשתמש במסמך זה:

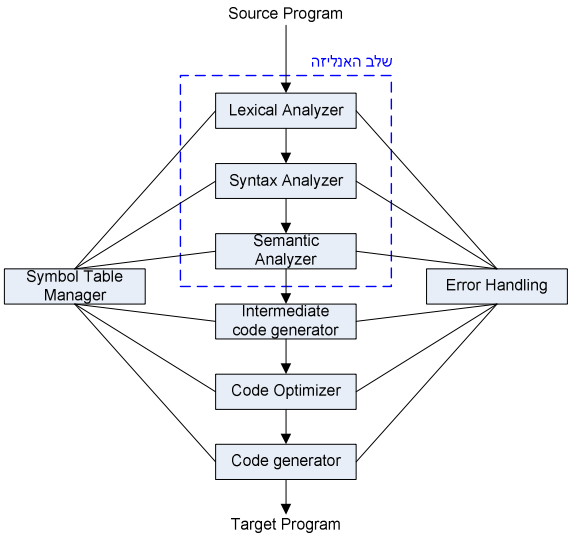
- **איברים אופציונליים** יסומנו ע"י סוגריים מרובעים. לדוגמא: [<item>]
  במקרה זה <item> יופיע 0 או 1 פעמים.
- **הופעה 0 או יותר פעמים** תסומן על ידי סוגריים מסולסלות, לדוגמא: {<letter>}
- **הופעה פעם אחת או יותר** תסומן על ידי האיבר ולאחריו '+'.

4. מבוא לקומפילציה

4.1 התמונה הגדולה

קומפיילר (מהדר) הוא תוכנית הקוראת תוכנית הכתובה בשפה אחת, **שפת המקור**, ומתרגם אותו לתוכנית זהה בשפה אחרת – **שפת היעד**. במידה ויש שגיאות בתוכנית בשפת המקור הקומפיילר מודיע על כך למשתמש.

ההידור מורכב משני חלקים: אנליזה וסינטזה. **בשלב האנליזה** הקוד מחולק לחלקים המרכיבים אותו, ונוצר ייצוג ביניים של הקוד. **בשלב הסינטזה** ניצור את התרגום בשפת היעד מתוך תרגום הביניים.



חלוקה ל-front end ו-back end: נחלק את שלבי ההידור ל-2 חלקים.

- **ה-front end** כולל שלבים (או חלקי שלבים) התלויים בשפת המקור ובגדול לא תלויים בשפת היעד. שלב ה-front-end מביא את התוכנית לייצוג ביניים שנוח לעבוד איתו.
- **ה-back end** מסתמכים על ייצוג הביניים שנוצר על ידי ה-front end (ולא על שפת המקור), ושם נמצאת רוב אופטימיזצית הקוד. שלב ה-back-end מבצע אופטימיזציות ומייצר קוד של שפת המטרה.

4.2. שלב האנליזה

ניתוח תוכנית המקור יוצג ביתר פירוט בהמשך המאמר. תהליך האנליזה מורכב מ-3 שלבים:

א. **ניתוח ליניארי**: בניתוח זה רצף התווים המרכיבים את התוכנית נקרא משמאל לימין ומורכב ל**אסימונים (tokens)** שהם רצפי תווים בעלי משמעות משותפת. שלב זה יכונה בהמשך גם **lexical analysis** או **scanning**.
דוגמא: קריאת רצף אותיות וקיבוצן ל*מזהה* (identifier).

ב. **ניתוח היררכי**: ה-tokens או התווים מקובצים לאוספים מקוננים בעלי משמעות. שלב זה יקרא בהמשך **parsing** או **syntax analysis**.
דוגמא: זיהוי קוד C – בדיקה שכל statement מסתיימת בנקודה-פסיק, בדיקה שהגדרת משתנים היא מהצורה type var_name; וכדו'.

ג. **ניתוח סמנטי**: בדיקות שרכיבי התוכנית מרכיבים יחד ביטויים בעלי משמעות.
דוגמא: בדיקה שלא מבוצעת חלוקה בקבוע אפס, בדיקה שאיננו מציבים ערך בוליאני למשתנה מסוג int וכדו'.

4.2.1. ניתוח לקסיקלי (lexical analysis)

תפקידי המנתח הלקסיקלי:

- מחלק את קוד המקור לאסימונים ("מילים")
- מסנן חלקים שאינם דרושים להמשך הניתוח: למשל, רווחים, ירידות שורה.
- מזהה שגיאות לקסיקליות - מחרוזות שאינן להיות אף אסימון, למשל, "@" בשפת C

מושג האסימון (token): ה-token הוא סוג של מעטפת (struct) שבו אנו עוטפים את מילת הקלט ומתייחסים אליה בשלב הראשון על פי הקבוצה אליה משתייכת המחרוזת ולא על פי ערכה.

דוגמאות: 34 זהו number, var זהו id (משתנה), 23.4 זהו number וכו'. number, id הם tokens.

נקודות:

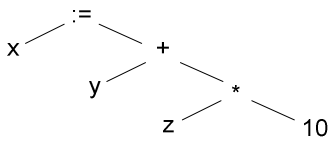
- הניתוח הלקסיקלי מתבצע בעזרת קבוצת חוקים של ביטויים רגולריים שעל פי הם המנתח הלקסיקלי יזהה את המחרוזות שיש בקלט וידע לשים אותן ל-token מתאים.
- מחרוזת ספציפית יכולה באופן עקרוני להשתייך למספר קבוצות ולכן למנתח הלקסיקלי ינתנו חוקי עדיפות לגבי השיוך של המחרוזת וכל מחרוזת תשתייך לקבוצה על פי העדיפות הגבוהה ביותר אליה היא מתאימה.
- לאחר שמסתיים שלב הניתוח הלקסיקלי יגיע שלב הניתוח הסינטקטי.

4.2.2. ניתוח סינטקטי (syntax analysis)

בשלב הניתוח הסינטקטי הופכים את ה-tokens שהוצאו מקוד המקור לייצוג ביניים ממנו נגזור את הפלט.

לעתים ה-tokens מיוצגים על ידי עץ גזירה. סוג נפוץ יותר של עץ בו נשתמש הינו **העץ הסינטקטי (syntax tree)**, בו כל צומת מייצגת פעולה והבנים של הצומת הינם הארגומנטים עליה פועלת אותה הפעולה.

דוגמא: העץ הסינטקטי עבור x := y + z *10



העץ הסינטקטי הוא יצוג דחוס של ה-parse tree בו האופרטורים מופיעים כצמתים הפנימיים, והבנים של כל צומת אופרטור הם אלו עליהם מבוצעת הפעולה.

4.2.3. ניתוח סמנטי

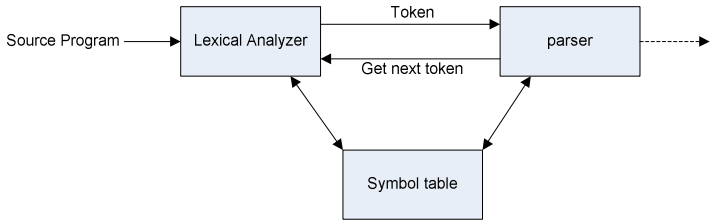
בשלב הניתוח הסמנטי אנו בודקים את תוכנית המקור בחיפוש אחר שגיאות סמנטיות, וכן אוספים מידע לגבי הטיפוסים עבור שלב ייצור הקוד. בשלב זה אנו משתמשים במבנה ההיררכי שנבנה בשלב הניתוח הסינטקטי כדי לזהות את האופרטורים והאופרנדים המרכיבים את הביטויים וההצהרות.

חלק חשוב בשלב זה הינו type checking – המהדר בודק שלכל אופרטורים מוצמדים אופרנדים המתאימים לו, לפי הגדרות השפה.

5. שלב 1 - ניתוח לקסיקלי (lexical analysis)

5.1 מושגים בסיסיים

פרק זה מציג שיטות להגדרת ומימוש מנתחים לקסיקלים. **מנתח לקסיקלי** הוא השלב הראשון שמבצע הקומפיילר. המטרה העיקרית שלו היא לקרוא את תווי הקלט ולייצר כפלט רצף של tokens שבהם ה-parser ישתמש לניתוח סינטקטי. קשר זה, המודגם בשרטוט הבא, גורם לכך שהמנתח הלקסיקלי לרוב ממומש כפונקציה של ה-parser.



מדוע אנו צריכים שלב נפרד למנתח הלקסיקי ולא מממשים אותו כחלק מה-parser?

א. חלוקת המשימה לתתי משימות מאפשרת פישוט של משימת הקומפילציה.
ב. חלוקה כזו מקלה על מימוש שינויים בשפה (למשל הוספת אפשרות להגדרת משתנים בעברית).
ג. אפשרות לשימוש חוזר בגרסאות הבאות של הקומפיילר.
ד. יעילות: שימוש בטכניקות נפרדות כאשר המשימה של חלוקת האסימונים מבוצעת במרוכז.

מושגים בסיסיים:

- **אסימון (token)**: יחידה בסיסית המשמשת כטרמינל בדקדוק שגוזר את שפת התכנות
- **לקסמה (lexeme)**: מחרוזת בקלט שהמנתח הלקסיקלי התאים לאסימון כלשהו
- **תבנית (pattern, regexp)**: ביטוי רגולרי שמגדיר את ההתאמה בין אוסף הלקסמות לאסימון מסוים.

דוגמא:

$$\underbrace{counter}_{Lexeme} \longrightarrow \underbrace{id}_{token} \underbrace{\{name = "counter"\}}_{properties}$$

תפקידי המנתח הלקסיקלי

א. זיהוי מחרוזות בקלט והפיכתן לאסימונים (מילים שמורות, קבועים, אופרטורים, מזהים...).
ב. סינון תבניות מסוימות (למשל הערות, או white-spaces).
ג. זיהוי שגיאות לקסיקליות (למשל הופעה של "$" או "@" בתוך תוכנית, לא במחרוזת או הערה).
ד. ביצוע פעולות סמנטיות (שיוך תכונות לאסימונים).

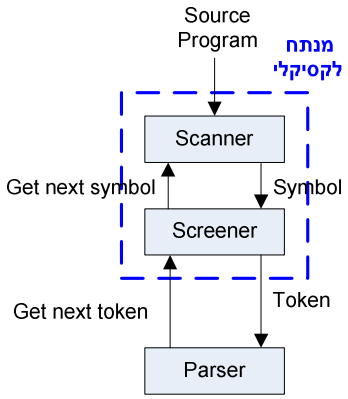
עבודה עם המנתח הלקסיקלי

הקלט של המנתח הלקסיקלי הוא תוכנית שרוצים לנתח, והפלט שלו הוא אסימונים עם תכונות.

5.2. Scanner ו-Screener

המנתח הלקסיקלי עצמו נחלק ל-2 חלקים, ה-scanner וה-screener. בגדול:

- **Scanner** - סורק את הקלט ומחלק אותו לאסימונים (tokens).
- **Screener** - מיסוך, חלק מהאסימונים לא חשובים, למשל הערות, ולכן אין צורך להעביר אותם הלאה. פעולת ה-screening היא סינון חלק מהאסימונים.



תפקידי ה- scanner:

- קריאת הקלט
- הפרדת הקלט ליחידות לקסיקליות (= symbols)
- העברת ה- symbols ל-screener
- טיפול ב- include files ו-macros
- ספירת מספר שורות
- דיווח על symbols לא חוקיים

ה-screener:

- לעיתים – משולב כתוכנה "פתוחה" בתוך ה- scanner (לדוגמא - LEX).
- תפקידו העיקרי של ה-screener – זיהוי ה- tokens ואיסוף אינפורמציה על ה- lexemes עצמן.

5.3 המנתח הלקסיקלי ותכונות סמנטיות

הלקסמות אינן עוברות הלאה לשלבים הבאים בניתוח - המנתח התחבירי אינו צריך אותן, אך המנתח הסמנטי כן צריך מידע עליהן, לדוגמא:

- האם קיים משתנה ששמו x?
- איזו פעולה בדיוק הופעלה?

המנתח הלקסיקלי מחשב לכל אסימון תכונות סמנטיות:

- id {name = "x"}
- op {type = "+"}

5.4. כלי Lex

5.4.1. רקע כללי

כלי לייצור מנתחים לקסיקליים. Lex מקבל הגדרת מנתח לקסיקלי מהמשתמש, ומייצר קובץ C שהוא הקוד של המנתח המבוקש. נשים לב: Lex הוא לא מנתח לקסיקלי.

תכונות המנתח הלקסיקלי ש-Lex בונה:

1. המנתח קורא קלט מ-stdin וכותב ל-stdout.
2. ה-"לב" של המנתח הוא הפונקציה yylex אשר מבצעת את הניתוח הלקסיקלי.
3. אין צורך לכתוב פונקצית main: אם לא נכתוב main, אז lex יספק פונקציה כזו בעצמו. הפונקציה המסופקת תקרא ל-yylex עד שהקלט נגמר.
4. הקובץ הנוצר הינו lexyy.c

5.4.2. פונקצית המנתח הלקסיקלי yylex()

פונים אל המנתח הלקסיקלי על ידי הפונקציה yylex(). זו פונקציה ללא פרמטרים המחזירה ערך int.

תכונות הפונקציה:

1. ערך ההחזרה של yylex הוא int.
2. הפונקציה קוראת את הקלט מהנקודה בה הפסיקה קודם (או תחילת הקלט אם זו הקריאה הראשונה ל-yylex), מזהה אסימונים בזה אחר זה ומבצעת את הפעולה שהמשתמש רשם עבור כל אסימון שמזוהה.
3. הפונקציה חוזרת (return) רק כאשר הפעולה שכתב המשתמש מכילה פקודת return, או כאשר היא מגיעה לסוף הקלט. מקובל: ערך 0 מייצג את סוף הקלט.
4. אם yylex לא מצליחה לזהות אף אסימן בנדוקה הנוכחית בקלט, היא מוציאה את התו הראשון שנקלט מהקלט, מדפיסה אותו לפלט, ומנסה שוב (מהמקום החדש בקלט).

5.4.3. קבצי קלט ופלט למנתח הלקסיקלי

המשתמש יכול להגדיר את קבצי הקלט והפלט שדרכן ניגשת yylex() אל הקלט והפלט.

- yyin הוא משתנה גלובלי המצביע אל קובץ הקלט (משתנה מסוג pointer to FILE). כברירת מחדל הוא מקבל את הערך stdin.
  דוגמא לקבלת קלט מקובץ:

```
yyin = fopen("file.txt", "r");
yylex();
```

- yyout הוא משתנה גלובלי המצביע אל קובץ הפלט (משתנה מסוג pointer to FILE). כברירת מחדל הוא מקבל את הערך stdout.

5.4.4. משתנים גלובלים נוספים

אורך המילה שנמצאה מתאימה לתבנית האסימון:	int yyleng;
תוכן המילה שנמצאה מתאימה לתבנית האסימון:	char *yytext;

תזכורת C: כדי להשתמש בכל אחד מהמשתנים בקבצים אחרים עלינו להשתמש במילת המפתח extern.

5.4.5. מבנה קובץ הקלט ל-lex

מבנה קובץ הקלט ל-lex:

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

לדוגמא:

```
%{
#include <stdio.h>
%}

id [a-z][a-z0-9]*

%%

{id}	{ printf("\nthe id found is: %s\n ",yytext); }

%%

void main(int argc,char *argv[])
{
	if (argc>1) yyin = fopen(argv[1],"r");
	else
	{
		printf("error: you should type input file name as a parameter...");
		exit(0);
	}
	yylex();
}
```

<u>החלק הראשון</u>

- הגדרות של שמות המפשטים את כתיבת הכללים (בחלק השני). הגדרות אלה מכילות הצהרות של שמות כדי לפשט את אפיוני המנתח. הגדרת שם היא מהצורה:

```
name definition
```

ה-name הוא מילה המתחילה באות או קו תחתון ולאחריה אפס או יותר אותיות, ספרות, קווים תחתונים (_) או קווי הפרדה (-). ה-definition מתחיל מהתו הראשון שאינו white space אחרי ה-name וממשיך עד לסוף השורה. בהמשך הקובץ שימוש ב-"{name}" יורחב ל-"(definition)".
דוגמא:

```
DIGIT		[0-9]
ID		[a-z][a-z0-9]*
```

DIGIT מוגדר להיות ביטוי רגולרי המתאים לסיפרה אחת, "ID" מוגדר להיות ביטוי רגולרי המתאים לאות ולאחריה אפס או יותר ספריות ואותיות.

לאחר ההגדרה נוכל למשל לכתוב:

```
{DIGIT}+
```

כדי להתייחס לביטוי בו ספרה אחת או יותר מופיעות ברצף.

קוד בשפת C שנכתב ע"י המשתמש ומועתק כמו שהוא לקובץ המיוצר על ידי flex. קוד זה יופיע לפני הפונקציה ()yylex. כדי להוסיף קוד כזה יש להקיף את הקוד ב- }% בתחילתו וב- {% בסיומו:

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

בדרך כלל שמים בחלק הזה פקודות define, פקודות include, הצהרות על משתנים גלובליים והכרזות שלנו.

<u>החלק השני</u>

חלק זה מכיל סדרה של כללים מהצורה:

```
pattern action
```

- **התבנית (pattern)** היא ביטוי רגולרי. התבנית חייבת להופיע בתחילת השורה (ללא whitespace לפניה). היא מסתיימת בתו הראשון שהוא whitespace (ניתן לשים whitespace כחלק מהתבנית על ידי הקפתו בגרשיים). יתרת השורה מהווה את ה-action.
- **action** הינו קוד בשפת C אותו מספק המשתמש. קוד זה מבוצע כאשר מתגלה התאמה לתבנית.

<u>החלק השלישי</u>

חלק זה מכיל קוד בשפת C שנכתב על ידי המשתמש. הוא מועתק כמו שהוא לסוף הקובץ ש-flex מייצר.

5.5. פתרון קונפליקטים

קונפליקט נוצר כאשר אותו קלט יכול להתאים למספר אסימונים.

הכללים של lex לפתרון הקונפליקט:

1. **המנתח חמדן**: המנתח תמיד מעדיף את הלקסמה הארוכה ביותר שניתן לבחור
2. **סדר התבניות**: אם כלל (1) לא פתר את הקונפליקט, בוחרים באסימון בעל עדיפות גבוהה יותר - האסימון שמופיע ראשון בהגדרות האסימונים.

5.6. Lookahead

הבעיה: לעיתים צריך לסרוק מספר אותיות קדימה על מנת להחליט מהו ה-symbol עליו אנחנו מתכוננים

הפתרון: אות מיוחדת שאינה חלק מהשפה (למשל - /) המציינת את מקום תחילת ה-lookahead, המוחלפת ב-ε בייצוג האוטומט.

6. שלב 2 – ניתוח תחבירי (syntax analysis)

6.1 רקע ומוטיבציה

לכל שפת תכנות יש חוקים המתארים את המבנה הסינטקטי של השפה. בשפת C למשל תוכניות מורכבות מבלוקים המורכבים מהצהרות, וכל הצהרה מורכבת מביטויים.

תחביר של שפה יכול להיות מוגדר על ידי **דקדוק חסר הקשר, או ע"י BNF**. כאשר אנחנו משתמשים בדקדוק ח"ה, כל מילה בשפה היא תוכנית שלמה בקוד המקור.

שימוש בדקדוק חסר הקשר:

- נותן לנו דרך קלה להבנה, אך בנוסף מדוייקת, להגדיר את תחביר השפה.
- בחלק מהמקרים, עבור חלק מהדקדוקים, ניתן לייצר באופן אוטומטי מנתח מתאים לשפה.
- שימוש בדקדוק חסר הקשר מאפשר הרחבות עתידיות קלות לשפה שלנו.

תפקיד המנתח התחבירי: המנתח התחבירי מקבל רצף של tokens מהמנתח הלקסיקלי. תפקידו הוא לוודא שרצף ה-tokens יכול להיווצר על ידי הדקדוק של שפת המקור.

דרישת מהמנתח התחבירי:

- **שלמות**: המנתח מזהה כל מילה בשפה. אם התכנית היא חוקית, המנתח יצליח לזהות אותה ויבנה עץ גזירה.
- **נאותות**: המנתח אינו מקבל מילים שאינן בשפה. אם התכנית לא חוקית, המנתח לא יצליח לזהותה ויודיע על שגיאה.

כל דקדוק חסר-הקשר שקול לאוטומט מחסנית. למה לא בונים אוטומט מחסנית ומשתמשים בו בתור מנתח תחבירי? האלגוריתם הקיים הינו בסיבוכיות $O\left(n^3\right)$ - סיבוכיות לא מעשית.

המטרה שלנו: פענוח ובנית **עץ הגזירה** תוך מעבר בודד על הקלט, משמאל לימין. הקלט של המנתח הוא רצף אסימונים (מילה / תוכנית מחשב) והפלט הוא עץ גזירה למילה / תוכנית.

חשוב לשים לב:

- כאשר אנחנו מדברים על "מעבר על הקלט" אנחנו מדברים בשלב זה על מעבר על האסימונים. כל אסימון נחשב כיחידה בסיסית.
- בכל שלב במהלך הניתוח התחבירי, הקלט w מתחלק ל- x y – כאשר x הוא החלק שכבר קראנו, ו-y החלק שטרם נקרא.

6.2. סוגי ניתוח תחבירי

ישנם 2 סוגים של ניתוח תחבירי:

- **top-down** – מהשורש לעלים (נקרא גם – "ניתוח תחזית" – predictive)
- **bottom-up** – מהעלים לשורש – מעבירים למחסנית, או מחליפים צד ימין בסימן מהצד השמאלי של חוק הדקדוק (shift reduce)

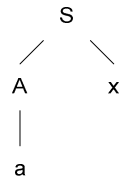
נציג את ההבדל בין הסוגים על ידי דוגמא: נביט בדקדוק הפשוט הבא:

$S \to Ax$
$A \to a$
$A \to b$

נניח שמילת הקלט היא ax, אזי הגזירה המתאימה השמאלית ביותר של הדקדוק הינה:

$S \to Ax \to ax$

במקרה זה, מכיוון שיש לנו רק משתנה אחד, זוהי גם הגזירה הימנית ביותר. עץ הגזירה המתאים הינו:



מנתח bottom-up מתחיל מהטרמינל ומטפס למעלה:

$ax \to Ax \to S$

באופן אינטואיטיבי, מנתח top-down מנסה להרחיב כל nonterminal (כל פעם את הימני ביותר), ואילו מנתח bottom-up מנסה כל פעם להחליף (reduce) את הצד הימני ביותר של הביטוי ב-nonterminals.

6.2.1. ניתוח Top-Down

אופן הפעולה:

1. הניתוח מתחיל מהמשתנה התחילי.
2. מפעילים רצף כללי גזירה.
3. עוצרים כאשר:
   a. מגיעים למילת הקלט $\Leftarrow$ המילה בשפה.
   b. נתקעים (אין או לא יודעים איזה כלל גזירה להפעיל) $\Leftarrow$ המילה אינה בשפה.

הבעיה העיקרית: איזה כלל להפעיל בכל צעד?

הסיטואציה: הגענו למשתנה A, שלו קיימים הכללים $A \to \alpha, A \to \beta$. אנו יודעים איזה משתנה רוצים לגזור (A), השאלה באיזה כלל של A להשתמש.

6.3. שיטת Recursive Descent (פועלת בשיטת Top-Down)

שיטת Recursive Descent היא מקרה פרטי של שיטת Top-Down. Recursive Descent זהו הרעיון הכללי מאחורי משפחה של אלגוריתמים, המאופיינת על ידי:

- האלגוריתם הוא רקורסיבי.
- לכל דקדוק מיוצרת תוכנית מיוחדת ע"מ לנתח אותו.
- כל משתנה (non-terminal) מטופל ע"י פונקציה נפרדת.
- המנתח מתחיל לפעול מהפונקציה המתאימה ל-nonterminal הראשון.
- כל פונקציה מחקה את החוק שעבורו הוגדרה, באופן הבא:
  - terminal מתורגם לקריאת האסימון המתאים מהקלט. (משתמשים בפונקציה match שתוצג בהמשך כדי "לאכול" אסימונים)
  - nonterminal מתורגם להפעלת הפונקציה המתאימה לו. האלגוריתמים שונים ביניהם בהחלטה מה לעשות כאלשר יש מספר חוקי גזירה עבור אותו nonterminal.
- דגש: השאלה איך גוף הפונקציה ייראה פתוחה לגמרי ותלויה במתכנת/במה שמייצר את המנתח. מכיוון שכך, RD היא משפחה של אלגוריתמים.

הרעיון המרכזי: נכתוב לכל משתנה פונקציה ש"תגזור" את כל המילים הנגזרות ממנו.

דוגמא לבחירת התנהגות לאלגוריתם – **ההתנהגות הבסיסית של RD**:

נבחר את כלל הגזירה לפי הסדר הבא:
1. מבין כל הכללים של משתנה A וטרמינל הבא בקלט t:
   a. אם ל-A יש כלל שמתחיל ב-t, בחר בו
   b. אחרת, אם ל-A יש כלל יחיד שמתחיל במשתנה, בחר בו
   c. אחרת, תודיע על שגיאה

אפשרות אחרת, במקום c – **שימוש ב-lookahead**:

אם ישנם כמה חוקי גזירה עבור אותו nonterminal, בוחרים ביניהם בעזרת lookahead.

פונקצית עזר של האלגוריתם:

```
void match(token t) {
     if (lookahead == t)
          lookahead = next_token();
     else
          error;
}
```

פונקצייה זו משמשת כדי לקרוא טרמינלים. ה-lookahead במקרה של RD שנציג הוא של אסימון אחד בלבד. קיימים אלגוריתמים RD שלא נציג להם lookahead גדול יותר.

דוגמא לשימוש. נביט בדקדוק:

$$OP \rightarrow and \mid or \mid xor$$

פונקציה מתאימה תהיה:

```
void OP() {
     if (lookahead = AND)       match(AND);
     else if (lookahead = OR)   match(OR);
     else if (lookahead = XOR)  match(XOR);
     else                       error;
}
```

שיטת lookahead RD: איך מחליטים באיזה כלל להשתמש?

- אם יש כמה כללים שונים לאותו nonterminal, יש לבחור ביניהם.
- עבור כלל A→α, נגדיר: $first(\alpha)$ – רשימת האסימונים שהם ראשונים באחת או יותר גזירות אפשריות הנובעות מכלל זה.
- אם אין חיתוך בין כל ה-FIRSTs עבור הכללים השונים ל-nonterminal נתון, אין כל בעיה ואנו יודעים באיזה כלל לבחור.
- אם יש חיתוך צריך לתקן את הדקדוק או להשתמש ב-lookahead עמוק יותר.

6.3.1. קבלת עץ גזירה באמצעות RD

ציינו כבר כי לצורך השלבים הבאים של הקומפילציה נצטרך יותר מאשר זיהוי וקבלת הקלט – נצטרך עץ גזירה.

נרצה להרחיב את הפונקציות שאנחנו כותבים ב-RD על מנת שיצרו לנו עץ: בכל פעם שנקראת אחת הפונקציות פירוש הדבר ש"איתרנו" צעד בגזירה. בכל צעד כזה ניתן לבצע פעולות שונות - בפרט, ניתן לבנות עץ בעזרת הפעולות הללו.

דרך אחת לקבל עץ מהפונקציות הקיימות:

- כל פונקציה מחזירה רשומה מסוג Node (צומת בעץ).
- כל רשומה כזו מכילה רשימה של בנים.
- בכל קריאה לפונקציה אחרת (או ל-match), מוסיפים את תוצאת הקריאה ל-Node שנבנה כעת.

בפועל בד"כ לא באמת בונים עץ גזירה. יצוג של התוכנית כולה בזכרון יכול להיות מסובך ו"כבד". עם זאת, באופן שקול, ניתן לבצע כל פעולה שהיא בפונקציות השונות המתקבלות באלגוריתם RD (Recursive Descent). פעולות אלה תייצרנה, בסופו של דבר, את הפלט של ה-parser: יצוג הביניים IR.

6.3.2. בעיות בשיטת RD

- <u>לא ניתן לטפל בכל הדקדוקים</u>: אם יש שני כללים שמתחילים במשתנה או אותו טרמינל לא נדע במי מהם לבחור.
- <u>המנתח לא שלם</u>: אינו מזהה את כל המילים בשפה.
  - דוגמא: עבור דקדוק עם שני כללי גזירה $S \rightarrow a \mid Bb$ ו-$B \rightarrow a$, עבור מילת קלט "ab" המנתח יבחר בכלל $S \rightarrow a$ וייתקע.
- <u>רקורסיה שמאלית</u> תגרום לריצה אינסופית: דוגמה: $S \rightarrow Sa \mid a$

פתרונות לבעיות:

1. שינוי הדקדוק לדקדוק שקול:
   a. **ביטול רישות משותפות** (**left factoring**). לדוגמא: $A \to \alpha B_1 \mid \alpha B_2 \mid ... \mid \alpha B_n$

יהפוך להיות:

$$A \to \alpha A'$$
$$A' = \beta_1 \mid \beta_2 \mid ... \mid \beta_n$$

   b. **ביטול רקורסיה שמאלית**. מיד יוצג אלגוריתם שמבצע זאת.
2. בחירת שיטה אחרת.

בעייתיות בפתרון 1:

1. לא מובטח שהבעיות ייפתרו. הפתרון לא נותן מענה לשני כללים שמתחילים ממשתנה שונה.
2. קיבלנו דקדוק "מכוער" ולא אינטואיטיבי.

אלגוריתם להעלמת רקורסיות שמאליות מדקדוק:

Input: Grammar G possibly left-recursive
Output: An equivalent grammar with no left-recursion
Method:
Arrange the nonterminals in some order $A_1, A_2, ..., A_n$
for i:=1 to n do begin

    for s:=1 to i-1 do begin
        replace each production of the form $A_i \to A_S \chi$ by the productions
        $A_i \to \delta_1 \chi \mid \delta_2 \chi \mid ... \mid \delta_k \chi$ where $A_S \to \delta_1 \mid \delta_2 \mid ... \mid \delta_k$ are all the current
        A_S-productions;
    end

    eliminate immediate left recursion among the A_i-productions
end

ביטול רקורסיה שמאלית ישירה (דוגמא):

$$A \to A\alpha \mid \beta$$

יהפוך ל:

$$A \to \beta A'$$
$$A' \to \alpha A' \mid \varepsilon$$

אלגוריתם Left Factoring: (מספר כללים המתחילים עם התחלה זהה ובהמשך יש התפצלות)

Input: Grammar G
Output: An equivalent left-factored grammar
Method:
For each nonterminal A find the longest prefix a common to two or more of its alternatives. If a not equal e, i.e., there is a nontrivial common prefix, replace all the A productions $A \to \alpha\beta_1 \mid \alpha\beta_2 \mid ... \mid \alpha\beta_n \mid \chi$ where χ represents all alternatives that do not begin with α by

$$A \to \alpha A' \mid \chi$$
$$A' \to \beta_1 \mid \beta_2 \mid ... \mid \beta_n$$

Here A'is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix.

6.3.3. שיפור בפונקציית הבחירה – select, first, follow

נגדיר פונקציה: $select: P \to 2^T$ שמגדירה לכל כלל איזה טרמינלים בקלט יגרמו לבחירת כלל זה.

לצורך ההגדרה נגדיר 2 פונקציות עזר, fisrt ו-follow, ובאמצעותן נגדיר את select.

הפונקציות מודגמות עכשיו בהקשר של שיפור ל-RD, אבל גם אלגוריתמים בהמשך ישתמשו בהן, ולכן חשוב להכירן היטב.

הפונקציה first

$$first(\alpha) = \{t \in T \mid \alpha \Rightarrow^* t\beta\}$$

אלגוריתם לחישוב first:

<u>שלב 1</u>: חישוב first עבור כל סימן $X \in (V \cup T)$ בדקדוק.

אתחול:

- לכל טרמינל $t \in T$, הגדר $first(t) = \{t\}$.
- לכל משתנה $A \in V$, הגדר $first(A) = \varnothing$
- לכל משתנה $A \in V$, אם קיים כלל $A \to \varepsilon$, $first(A) = first(A) \cup \{\varepsilon\}$

צעד: כל עוד יש שינויים, לכל כלל $A \to Y_1 Y_2 ... Y_k$ (כאשר $Y_1, Y_2, ..., Y_k \in (V \cup T)$) בצע:

- $first(A) = first(A) \cup first(Y_1)$
- לכל $i > 1$ עבורו $Y_1 Y_2 ... Y_{i-1} \Rightarrow^* \varepsilon$ בצע $first(A) = first(A) \cup first(Y_i)$

<u>שלב 2</u>: חישוב first עבור רצף סימנים $\alpha = x_1 ... x_k$

$$first(\alpha) = \bigcup_{x_1 x_2 ... x_{j-1}} first(\alpha)$$

כלומר: $first(x_1)$ מאוחד עם $first(x_2)$ במקרה ש- x_1 אפיס. כל זה מאוחד עם $first(x_3)$ במקרה ש- x_1, x_2 אפיסים, וכך הלאה.

הפונקציה follow

$$follow(A)=\{t\in T\cup\{\$\}\mid S\$\Rightarrow^{*}\alpha At\beta\}$$

אלגוריתם לחישוב follow:

אתחול:
- לכל משתנה $A\neq S$, נגדיר $follow(A)=\varnothing$.
- עבור המשתנה התחילי S, נגדיר $follow(S)=\{\$\}$.

צעד: כל עוד יש שינויים, לכל משתנה $A\in V$ בצע:
- לכל כלל $Y\to\alpha A\beta$ (שבו A מופיע באגף מצד ימין):
  - $follow(A)=follow(A)\cup first(\beta)$
  - אם $\beta\Rightarrow^{*}\varepsilon$ בצע גם $follow(A)=follow(A)\cup follow(Y)$

הערות:
- $ - סימן סוף קלט (לא חלק מאוסף הטרמינלים). אנחנו מציינים בעצם באתחול כי אחרי המשתנה ההתחלתי מגיע סוף הקלט.
- β זה כל הביטוי בהמשך, לא רק המשתנה הצמוד. חשוב לשים לב כי נופלים בזה בתרגילים.

הפונקציה select

הפונקציה select תוגדר באופן הבא:

$$select(A\to\alpha)=\begin{cases}first(\alpha)\cup follow(\alpha) & \alpha\Rightarrow^{*}\varepsilon\\ first(\alpha) & otherwise\end{cases}$$

נסכם את המשמעויות:
- $select(A\to\alpha)$: טרמינלים שיגרמו לבחור בכלל זה כאשר רוצים להפעיל כלל של A. לצורך חישוב של select, מגדירים פונקציות עזר first ו-follow.
- first(A): טרמינלים שיכולים להופיע בתחילת מילה שנגזרת מ-A.
- first(α): ההרחבה של first לתבניות פסוקיות. כל הטרמינלים שיכולים להופיע בתור האות הראשונה של התבניתפסוקית.
- follow(A): טרמינלים שיכולים להופיע אחרי A בגזירה כלשהי

6.4. אלגוריתם LL(k)

6.4.1. רקע

אלגוריתם LL(k) הוא אלגוריתם בעל התכונות הבאות::

- top-down
- מבוסס טבלה
- סורק את הקלט משמאל (L) לימין
- מניב את הגזירה השמאלית (L) ביותר
- זקוק ל-lookahead בגודל k
- איטרטיבי, בניגוד ל-recursive descent שהוא רקורסיבי

המקרה הפשוט ביותר הוא אלגוריתם LL(1).

שפה נקראת LL(k) אם אפשר לגזור אותה בעזרת LL(k) parser. אלגוריתם RD גוזר שפות LL(k) עבור k בלתי-חסום. בדרך-כלל מנתחים שפות LL(k) בעזרת אלגוריתמים מבוססי-טבלה. אלגוריתמים אלו ידועים בשם LL(k) parsers.

דוגמאות:

1. דקדוק/שפה שאינם LL(k):

$$S \to A \mid B$$
$$A \to aBb \mid c$$
$$B \to aBbb \mid d$$

לכל k, קיימת מילה ארוכה מספיק שלא יהיה ניתן לגזור אותה ב-LL(k).

2. דקדוק ב-LL(k+1) שאינו ב-LL(k):

$$S \to a^k b \mid a^k c$$

ניתן לקבל דקדוק שקול ב-LL(1) באמצעות left-factoring.

6.4.2. אלגוריתם LL(1)

אלגוריתם LL(1) מקבל: קלט, מחסנית וטבלת מעברים, ומוציא עץ גזירה כפלט.

לדקדוק G קיים מנתח LL(1) אם ורק אם לא קיים קונפליקט בדקדוק, כלומר לכל שני כללים $A \to \alpha$ ו- $A \to \beta$ בדקדוק כך ש- $\alpha \neq \beta$, מתקיים $select(A \to \alpha) \bigcap select(A \to \beta) = \varnothing$.

הרעיון של האלגוריתם הוא לחקות בעזרת מחסנית את התנהגות Recursive Descent.

מבני הנתונים של האלגוריתם:

- **מחסנית Q**: מחסנית הניתוח המחזיקה את מה שרוצים עדיין לראות. בראש המחסנית מופיע מה שאנחנו רוצים לראות מיד. המחסנית מאותחלת ב-S, ותומכת בפעולות הרגילות pop, push, top.
- **טבלת המעברים M:** ב-LL(1) משתמשים בטבלה המכתיבה, עבור כל מצב נתון, באיזה כלל גזירה להשתמש. שורות הטבלה: משתנים. עמודות הטבלה: אסימונים אפשריים בקלט. תוכן הטבלה: חוקי גזירה. כאשר אין קונפליקטים, כל תא של M מכיל איבר אחד בלבד. **בניית הטבלה** מבוצעת לפי הכלל הבא:

$$M_{X \in V, t \in T}(X,t) = \begin{cases} X \to \alpha & t \in select(X \to \alpha) \\ error & otherwise \end{cases}$$

האלגוריתם:

אתחול המחסנית: המשתנה (nonterminal) הראשון בדקדוק, ו-$ (סימן לסוף הקלט).
- המחסנית יכולה להכיל אסימונים (terminals) או משתנים. "$" הוא אסימון מיוחד, לצורך זה.

אם בראש המחסנית יש אסימון:
- אם האסימון הבא בקלט אינו זהה: שגיאה.
- אם הוא תואם את הקלט: צרוך את תו הקלט; הסר את האסימון מהמחסנית. (אם האסימון הוא $, סיימנו). **פעולת SHIFT**

אם בראש המחסנית יש משתנה:
- מצא את התא בטבלה M המתאים למשתנה זה ולתו שבראש הקלט.
- אם התא ריק: שגיאה.
- אחרת: הסר את המשתנה מראש המחסנית; הוסף למחסנית את צד ימין של כלל הגזירה שנמצא בטבלה, לפי סדר – החל באסימון/משתנה הימני ביותר וכלה באסימון/משתנה השמאלי ביותר (הוא ישאר בראש המחסנית). **פעולת REPLACE**

דוגמא לבניית הטבלה. נבצע חישוב select על הדקדוק הנתון. נניח כי מקבלים:

$$select(S \to Ab) = \{a\}$$
$$select(S \to bC) = \{c\}$$
$$select(A \to a) = \{a\}$$
$$select(C \to cA) = \{c\}$$

הטבלה M המתאימה הינה:

	a	b	c	$
S	$S \to Ab$	$S \to bC$		
A	$A \to a$			
C			$C \to cA$	

המקומות הריקים מכילים error.

6.5. סיכום שיטות Top-Down

השיטות שראינו, היתרונות והחסרונות של כל אחת מהשיטות:

שיטה	תיאור
RD	המנתח הנוצר אינו בהכרח נכון ואינו בהכרח יעיל
Improved-RD	דורשת select-ים זרים, ויצירת תוכנית מיוחדת לכל דקדוק
LL(1)	שכוחה בכוחה ל-Improved RD, אבל אוטומטית

6.6. אלגוריתמים LR(K)

אלגוריתם LR(k) הוא אלגוריתם בעל התכונות הבאות:

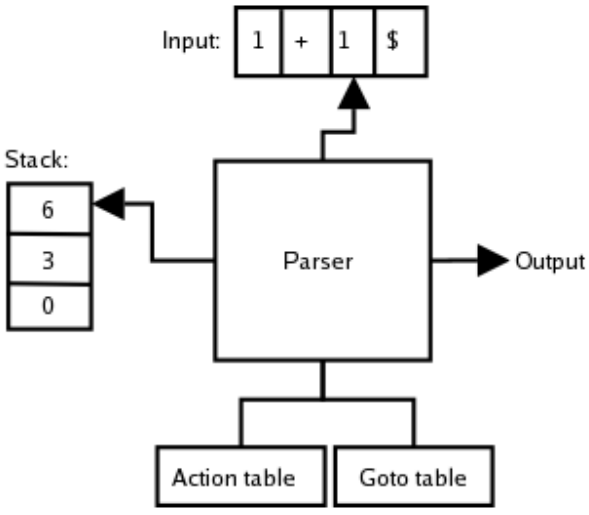
- bottom-up
- מבוסס טבלאות
- סורק את הקלט משמאל (L) לימין
- מניב את הגזירה הימנית (R) ביותר
- זקוק ל-lookahead בגודל k

המקרה הפשוט ביותר הוא אלגוריתם LR(0). אלגוריתם זה פשוט, אך קיימים דקדוקים רבים שהוא לא יצליח לנתח. האלגוריתם LR(1) שיוצג בהמשך מסוגל להתמודד עם רוב שפות התכנות הקיימות.

מכיוון ש-LR קורא את הקלט משמאל לימין, אך צריך לייצר את הגזירה הימנית ביותר, הוא משתמש ברדוקציה כדי לנתח את הקלט. על ידי יצירת הרדוקציה השמאלית ביותר נקבל את הגזירה הימנית ביותר. (נשים לב – בכל פעם אנחנו בוחרים את המשתנה המתאים לחלק השמאלי ביותר. בסופו של דבר המשתנה עבור החלק הימני יבחר אחרון – וכאשר נסתכל top-down נשים לב שזו גזירה ימנית)

שפה נקראת LR(k) אם אפשר לגזור אותה בעזרת LR(k) parser.

<u>תכונה</u>: כל שפה LL(n) היא גם LR(n) – אבל לא להיפך. כלומר, LR מכיל ממש את LL.



הארכיטקטורה של מנתחי LR (השרטוט לקוח מויקיפדיה)

6.6.1. אלגוריתם LR(0)

6.6.1.1. הצגת האלגוריתם

אלגוריתם LR(0) מקבל: קלט, מחסנית, טבלת פעולות וטבלת goto, ומוציא עץ גזירה כפלט. ה-parser הוא אוטומט המגיב בהתאם לרכיבים השונים.

המחסנית:

- המחסנית ב-LR מכילה **מצבים** (מזוהים על-ידי מספר).
- לצורך קריאות, נכלול במחסנית גם משתנים ואסימונים. אין בהם צורך באלגוריתם. לצורך האלגוריתם מספיק לשמור מצבים, אך על מנת להבין יותר את מצב התוכנית מתוך שרטוט המחסנית נגדיר שגם משתנים ואסימונים ימוקמו במחסנית.
- למעט מצב "0" שנמצא בתחתית המחסנית, כל שאר המחסנית תהיה מורכבת מזוגות של מצב+משתנה או מצב+אסימון.
- המחסנית ההתחלתית מכילה רק את מצב "0".

טבלת הפעולות:

- טבלה זו מכתיבה את הפעולה לביצוע בכל שלב.
- שורות הטבלה: מצבים (על-פי מספרים).
- עמודות הטבלה: אסימונים אפשריים בקלט.
- תוכן הטבלה: הוראות ביצוע.

הוראות ביצוע: הוראות הביצוע בטבלת הפעולות יכולות להיות:

- **shift n**
  - מסומנת sn (למשל s1, s2 וכו').
  - פירושה: מעבר למצב n (דחיפה של n למחסנית), העברת תו הקלט הבא למחסנית והסרתו מה-input stream.
- **reduce m**
  - מסומנת rm (למשל r1, r2 וכו')
  - פירושה: זיהינו כלל גזירה מס' m
- **accept**
  - מסומנת acc
  - פירושה: הקלט נגזר בהצלחה.
- **תאים ריקים – שגיאה בקלט**.

טבלת Goto:

טבלת העזר השנייה משמשת במקרים בהם זיהינו גזירה של כלל.

- שורות הטבלה: מצבים
- עמודות הטבלה: משתנים
- תוכן הטבלה: מצבים.

היות שלשתי הטבלאות (goto, action) אותן שורות, מקובל להציג אותן זו-לצד-זו.

האלגוריתם עצמו:

- אתחול המחסנית: "0"
- מצא את action[t, i] (i ראש המחסנית, t אסימון הקלט הבא):
  - אם מצאת shift n:
    - הסר את האסימון t מהקלט,
    - הוסף את t ואח"כ את n למחסנית.
  - אם מצאת reduce m:
    - יהי w מספר התווים בצד ימין של כלל הגזירה מספר m. הסר 2w ערכים מהמחסנית.
    - יהי M המשתנה שגוזר כלל m. הוסף את M למחסנית.
    - מצא את goto[s, M] (s ראש המחסנית החדש) והוסף אותו למחסנית.
    - כתוב m לפלט
  - אם מצאת acc: סיום בהצלחה.
  - אחרת: סיום בשגיאה.
- המשך עד לעצירה.

הערה: קיימת גרסה של האלגוריתם ללא פלט, בה התשובה של האלגוריתם היא דחיה או קבלה בלבד.

6.6.1.2 דוגמא לשימוש במנתח

נגדיר את כללי הגזירה הבאים:

(1) $E \to E * B$

(2) $E \to E + B$

(3) $E \to B$

(4) $B \to 0$

(5) $B \to 1$

טבלה אפשרית עבור כללים אלה: (בהמשך נראה כיצד בונים טבלה כזו)

	טבלת הפעולות (action)					טבלת goto	
מצבים	*	+	0	1	$	E	B
0			s1	s2		3	4
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	s5	s6			acc		
4	r3	r3	r3	r3	r3		
5			s1	s2			7
6			s1	s2			8
7	r1	r1	r1	r1	r1		
8	r2	r2	r2	r2	r2		

הרצה של האלגוריתם עבור הקלט "1+1". (במחסנית מוצגים המצבים בלבד)

State	Input Stream	Output Stream	Stack	Next Action
0	1+1$		[0]	Shift 2
2	+1$		[0,2]	Reduce 5
4	+1$	5	[0,4]	Reduce 3
3	+1$	5,3	[0,3]	Shift 6
6	1$	5,3	[0,3,6]	Shift 2
2	$	5,3	[0,3,6,2]	Reduce 5
8	$	5,3,5	[0,3,6,8]	Reduce 2
3	$	5,3,5,2	[0,3]	Accept

הסבר חלקי:

- המנתח בהתחלה שם במחסנית את המצב ההתחלתי [0].
- התו הראשון בקלט שהמנתח רואה הוא 1. כדי למצוא מה השלב הבא הולכם לטבל ה-action לשורת המצב הנוכחי (המצב הנוכחי, כאמור, הוא זה שבראש המחסנית – במקרה שלנו – 0). הפעולה היא shift 2, ולכן המצב [2] נדחף למחסנית (וגם האסימון '1' – לא מוצג בטבלה לעיל).
- במצב 2 הטבלה אומרת שלא משנה איזה טרמינל אנחנו רואים, צריך לעשות reduce 5. אנחנו רושמים 5 לפלט, מוצאים את מצב 2 מהמחסנית, ודוחפים למחסנית את הערך המתאים שבטבלת goto – עבור מצב 0 ומשתנה B.
- כך ממשיכים עד שמגיעים ל-Accept.

המספרים שנכתבו כפלט [5,3,5,2] הם אכן הגזירה הימנית ביותר שלו.

6.6.1.3. בניית טבלאות LR(0)

הגדרות לקראת בניית הטבלאות:

פריט LR(0)

הגדרה: **פריט LR(0)** (או בקיצור פריט, ובעברית item) הוא כלל גזירה עם "נקודה" מיוחדת בתוכו. מכלל גזירה עם n רכיבים מצד ימין ניתן לקבל n+1 פריטים. למשל, עבור הכלל E → E + B (3 רכיבים מימין) נקבל את 4 הפריטים הבאים:

E → · E + B

E → E · + B

E → E + · B

E → E + B·

כלל ε נחשב **לכלל באורך 0**, לכן מהכלל A → ε ניתן לקבל רק את הפריט:

$$A \rightarrow \cdot$$

פריט אינו באמת כלל גזירה (בכללים האמיתיים אין את הנקודה). **פריט מסמל את מצבו של ה-parser.**

משמעותו: זיהינו את מה שנמצא לפני (משמאל) הנקודה; אנו מצפים כעת למצוא את מה שנמצא אחריה (מימינה).

למשל, הפריט:

E → E · + B

מסמן כי ה-parser זיהה מחרוזת המתאימה ל-E בקלט, והוא כעת מצפה למצוא "+" ואחר-כך מחרוזת המתאימה ל-B.

קבוצות של פריטים

בדרך-כלל לא ניתן לתאר את מצב ה-parser בעזרת פריט יחיד. יתכנו מספר אפשרויות להמשך הגזירה. לכן מציינים את מצבו של ה-parser בעזרת **קבוצה של פריטים**.

סגור של קבוצת פריטים

אם קבוצת הפריטים הנוכחית כוללת מצב שבו הנקודה נמצאת לפני משתנה, קשה לדעת מה צריך להיות הפריט הבא בקלט. לשם כך בונים את קבוצת הסגור של קבוצת הפריטים.

הגדרה: **קבוצת הסגור של קבוצת פריטים**: קבוצת פריטים שבה, עבור כל פריט בקבוצה מהצורה

$A \rightarrow \alpha \cdot B\ \beta$

ועבור כל כלל מהצורה $B \rightarrow \delta$ בדקדוק, גם הפריט

$B \rightarrow \cdot\ \delta$

נמצא בקבוצה.

בניית קבוצת הסגור היא איטרטיבית, משום שגם δ עשוי להתחיל במשתנה. (ההגדרה של הסגור אומרת בעצם – כל עוד האיבר אחרי הנקודה הוא משתנה, המשך להרחיב).

דקדוק מורחב

הגדרה: **דקדוק מורחב** הוא דקדוק שהוסיפו לו כלל יחיד, המבטיח שהגזירה האחרונה היא חד-משמעית בהיותה אחרונה. נרחיב את הדקדוק על-ידי הוספת כלל גזירה $S \rightarrow E$, כאשר:

- S משתנה שלא היה קיים בדקדוק לפני-כן, והוא כעת המשתנה הראשון,
- E המשתנה הראשון המקורי של הדקדוק.

נהוג למספר את הכלל החדש "0". הדקדוק אחרי ההוספה של הכלל:

$(0) \quad S \rightarrow E$

$(1) \quad E \rightarrow E * B$

$(2) \quad E \rightarrow E + B$

$(3) \quad E \rightarrow B$

$(4) \quad B \rightarrow 0$

$(5) \quad B \rightarrow 1$

הערה: הצורך במשתנה הזה הוא לדאוג שכאשר נגיע אליו בסופו של דבר, נדע כי סיימנו את התהליך.

מכונת המצבים והמצב ההתחלתי

- בבניית הטבלאות, אנו בונים למעשה מכונת מצבים דטרמיניסטית שכל מצב שלה מסומן על-ידי קבוצת סגור כלשהי של פריטים.
- המצב ההתחלתי הוא הסגור של הפריט הראשון מהכלל שהוספנו בבניית הדקדוק המורחב: S → · E. זוהי "קבוצת פריטים 0".
- המצבים הבאים:
  - מצא את כל הפריטים במצב הנוכחי, שבהם הנקודה נמצאת לפני סימן נתון כלשהו x. הסימן x יכול להיות אסימון או משתנה. נסמן קבוצת פריטים זו S (תת-קבוצה של המצב הנוכחי).
  - הזז את הנקודה צעד אחד ימינה עבור כל הפריטים ב-S.
  - מצא את הסגור של הקבוצה שהתקבלה.
  - כל clos(S) שנמצא בעזרת הכללים הללו הוא מצב חדש שניתן להגיע אליו מהמצב הנוכחי, כאשר בקלט מופיע x.

דוגמא: עבור הדקדוק שהוצג, קבוצת המצבים 0 הינה:

Item set 0

$$S \to \cdot E$$
$$E \to \cdot E * B$$
$$E \to \cdot E + B$$
$$E \to \cdot B$$
$$B \to \cdot 0$$
$$B \to \cdot 1$$

כעת, עבור הטרמינל '0' נקבל את קבוצה 1:

Item set 1

$$B \to 0 \cdot$$

עבור הטרמינל '1' נקבל את קבוצה 2:

Item set 2

$$B \to 1 \cdot$$

עבור המשתנה E נקבל את קבוצה 3:

Item set 3

$S \to E \cdot$
$E \to E \cdot * B$
$E \to E \cdot + B$

עבור המשתנה B נקבל את קבוצה 4:

Item set 4

$E \to B \cdot$

נשים לב שהסגור לא מוסיף פריטים חדשים בכל המקרים. אנחנו ממשיכים בתהליך עד שלא ניתן למצוא קבוצות נוספות.

עבור קבוצות 1,2, 4 אין המשך מכיוון שהנקודה אינה בקדמת אף סימן. עבור קבוצה 3 אנו רואים שקיימת נקודה לפני הטרמינל * וגם לפני + .

עבור * נקבל את קבוצה 5:

Item set 5

$E \to E * \cdot B$
$B \to \cdot 0$
$B \to \cdot 1$

עבור + נקבל את קבוצה 6:

Item set 6

$E \to E + \cdot B$
$B \to \cdot 0$
$B \to \cdot 1$

מכאן: חשוב לשים לב שאם נמשיך עם 0 או אם 1 נגיע לקבוצות 1 ו-2 בהתאמה. אין להגדיר קבוצות כפולות. עבור המשתנה B נגיע לקבוצה 7:

Item set 7

$E \to E * B \cdot$

מקבוצה 6, עבור הטרמינלים 0 ו-1 נגיע גם לקבוצות 1,2. עבור B נקבל את קבוצה 8:

Item set 8

$E \to E + B \cdot$

בכל הקבוצות הסופיות שקיבלנו אין נקודת לפני סימן, ולכן סיימנו. טבלת המעברים של אוטומט המצבים המתאים נראה כך:

Item Set	*	+	0	1	E	B
0			1	2	3	4
1						
2						
3	5	6				
4						
5			1	2		7
6			1	2		8
7						
8						

בניית טבלאות action ו-goto

בהנתן טבלת המעברים והקבוצות שבנינו נבנה את goto ו-action באופן הבא:

1. העמודות של המשתנים מועתקות לטבלת ה-GOTO
2. העמודות של הטרמינלים מועתקות לטבלת ה-action כפעולות shift.
3. אנו מוסיפים עמודה נוספת לטבלת ה-action עבור $ (סוף הקלט) שמכילה acc עבור קבוצה שמכילה $S \to E\bullet$
4. אם קבוצה i מכילה פריט מהצורה $A \to w\bullet$ וגם $A \to w$ הוא חוק m (כאשר $m > 0$) אז נמלא את כל השורה עבור i בפעולה rm.

ניתן לראות שמתקבלת הטבלה שהצגנו בהתחלה.

6.6.1.4. דוגמא

נציג כעת דוגמא מלאה נוספת לבניית אוטומט וטבלאות LR(0) מתוך דקדוק נתון. הקורא מוזמן לפתור דוגמא זו כתרגיל.

הדקדוק הנתון:

1. S' -> S
2. S -> aAB
3. A-> Ab
4. A-> ε
5. B-> c
6. B-> cAd

נבחין כי אין צורך להוסיף כלל וליצור דקדוק מורחב מכיוון שהכלל 1 הוא כבר כלל המקיים שהגזירה האחרונה היא חד משמעית.

נמצא את קבוצות הפריטים השונות:

$$\text{Item set } 0 = CLOS\left(\left\{S' \to \bullet S\right\}\right) = \left\{S' \to \bullet S, S \to \bullet aAB\right\}$$

מקבוצה 0, עבור טרמינל a נקבל:

$$\text{Item set } 1 = \left\{S \to a \bullet AB, A \to \bullet Ab, A \to \bullet\right\}$$

מקבוצה 0, עבור משתנה S נקבל:

$$\text{Item set } 2 = \left\{S' \to S \bullet\right\}$$

מקבוצה 1, עבור משתנה A נקבל:

$$\text{Item set } 3 = \left\{S \to aA \bullet B, A \to A \bullet b, B \to \bullet c, B \to \bullet cAd\right\}$$

מקבוצה 3, עבור משתנה B נקבל:

$$\text{Item set } 4 = \left\{S \to aAB \bullet\right\}$$

מקבוצה 3, עבור טרמינל b נקבל:

$$\text{Item set } 5 = \{A \to Ab\bullet\}$$

מקבוצה 3, עבור טרמינל c נקבל:

$$\text{Item set } 6 = \{B \to c\bullet, B \to c\bullet Ad, A \to \bullet Ab, A \to \bullet\}$$

מקבוצה 6, עבור משתנה A נקבל:

$$\text{Item set } 7 = \{B \to cA\bullet d, A \to A\bullet b\}$$

מקבוצה 7, עבור טרמינל d , נקבל:

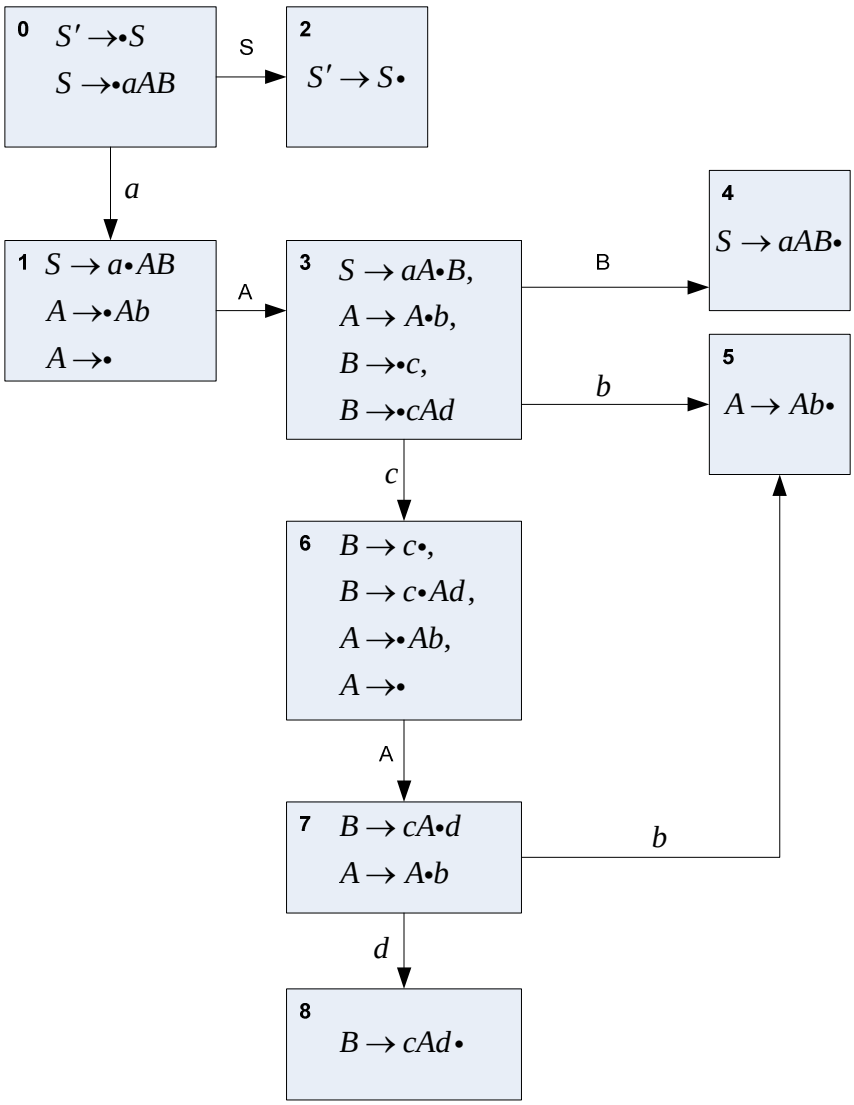
$$\text{Item set } 8 = \{B \to cAd\bullet\}$$

מקבוצה 7, עבור טרמינל b , נקבל שוב את קבוצה 5.

טבלת המעברים של האוטומט:

	a	b	c	d	S	A	B
0	1				2		
1						3	
2							
3		5	6				4
4							
5							
6						7	
7		5		8			
8							

האוטומט עצמו:



טבלאות goto ו-action עבור מנתח LR(0):

	actions					goto		
	a	*b*	*c*	*d*	$	*S*	*A*	*B*
0	s1					2		
1	r4	r4	r4	r4	r4		3	
2					acc			
3		s5	s6					4
4	r2	r2	r2	r2	r2			
5	r3	r3	r3	r3	r3			
6	r5/r4	r5/r4	r5/r4	r5/r4	r5/r4		7	
7		s5		s8				
8	r6	r6	r6	r6	r6			

נשים לב שקיימת התנגשות בטבלה! בנושא זה נעסוק מיד.

6.6.1.5. קונפליקטים בטבלאות

כשמוסיפים הוראות reduce לשורה בטבלת הפעולות, לא מוחקים מה שכבר קיים בשורה זו, ולכן ייתכן שבתאים מסוימים בטבלה יש שני ערכים (או יותר). מקרים כאלה נקראים **קונפליקטים**.

סוגי קונפליקטים:

- **קונפליקט reduce/reduce** נוצר כשבתא אחד יש כמה אפשרויות שונות ל-reduce.
- כשבתא אחד יש גם הוראת reduce וגם הוראת shift, מקבלים **קונפליקט shift/reduce**.

איך הקונפליקטים נוצרים? נביט בבניה שלנו: האוטומט נבנה בדרך שמובטחת להיות דטרמיניסטית. לעומתו – פעולות reduce מוספות ללא התחשבות בתוכן הנוכחי של התאים באותה שורה, ולכן יכולים להיווצר מצבים שתא יכיל כמה פעולות reduce או שילוב של shift ושל reduce.

טענה: ניתן להוכיח כי דקדוק שבו יש קונפליקט הוא לא דקדוק LR(0).

דוגמא לדקדוק בעייתי מסוג shift-reduce (הדוגמא לקוחה מויקיפדיה האנגלית):

$$(1) \quad E \to 1E$$
$$(2) \quad E \to 1$$

אחת מהקבוצות שניתן למצוא היא:

$$E \to 1 \bullet E$$
$$E \to 1 \bullet$$
$$E \to \bullet 1E$$
$$E \to \bullet 1$$

יש התנגשות shift-reduce בגלל שבאותו תא יש גם reduce של כלל 2 וגם shift של הטרמינל 1.

אבחנות:

- הקונפליקט קיים כשהמכונה במצב 1 וקיים האסימון 1 בקלט.
- בלי שום קשר לקלט, כשאנחנו במצב 1, מבצעים reduce לכלל הגזירה $E \to 1$ בגלל הפריט $E \to 1 \bullet$.

דוגמא לדקדוק בעייתי מסוג reduce-reduce:

$(1) \quad E \to A1$

$(2) \quad E \to B1$

$(3) \quad A \to 1$

$(4) \quad B \to 1$

אנחנו מקבלים את הקבוצה הבאה:

$A \to 1\bullet$

$B \to 1\bullet$

בשורה זו יהיה קונפליקט מסוג reduce-reduce.

6.6.2. SLR

כדי לפתור את הבעיה שהוצגה נציג שיפור למנתח, שיכונה SLR.

בהמשך לדוגמא הראשונה, LR(0) לא יכול להכריע בין שני המצבים, אך עם זאת אפשר להגיע להחלטה בקלות: קל לראות שאחרי E לא יכול לבוא האסימון 1. כלומר: $1 \notin follow(E)$. לכן, אם יש 1 בהמשך הקלט לא צריך לגזור E, ומבצעים shift ולא reduce.

נתקן את LR(0) כך:

צעד ה-reduce המקורי בבניית הטבלה:
- לכל מצב עם פריט A → α· , מוסיפים reduce מתאים לכל השורה.

הופך להיות:

- לכל מצב עם פריט A → α· , מוסיפים reduce מתאים בשורה זו, לכל עמודה שהאסימון שבראשה שייך ל-follow(A).

האלגוריתם החדש יכונה **אלגוריתם SLR.** הוא מסוגל לזהות יותר שפות מ-LR(0) ללא קונפליקטים.

נאמר על דקדוק שהוא **דקדוק SLR** אם ניתן לבנות עבורו מנתח SLR נאות ושלם.

האם SLR פותר את הדקדוק שהוצג קודם? הדקדוק:

$(1) \quad E \to 1E$

$(2) \quad E \to 1$

נפתח באופן מלא ונבדוק. הקבוצות השונות:

Item set 0	Item set 1	Item set 2	Item set 3
$S \to \bullet E$	$E \to 1 \bullet E$	$S \to E \bullet$	$E \to 1E \bullet$
$E \to \bullet 1E$	$E \to 1 \bullet$		
$E \to \bullet 1$	$E \to \bullet 1E$		
	$E \to \bullet 1$		

מנתח LR(0) המתאים כולל ההתנגשות:

	action		*goto*
state	**1**	**$**	**E**
0	s1		2
1	s1/r2	r2	3
2		acc	
3	r1	r1	

מנתח SLR המתאים:

ה-follow של E הוא $\{\$\}$ ולכן r1 ו-r2 קיימים רק בעמודת ה-#. התוצאה היא טבלה ללא קונפליקטים:

	action		*goto*
state	**1**	**$**	**E**
0	s1		2
1	s1	r2	3
2		acc	
3		r1	

דוגמא נוספת: נמשיך את דוגמא 6.6.1.4. נבנה את הטבלה המתאימה ל-SLR.

ראשיית נחשב first ו-follow לכל המשתנים:

First

#	S'	S	A	B
1	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
2	$\varnothing$	$\{a\}$	$\{\varepsilon, b\}$	$\{c\}$
3	$\{a\}$	$\{a\}$	$\{\varepsilon, b\}$	$\{c\}$

Follow

#	S'	S	A	B
1	$\{\$\}$	$\varnothing$	$\varnothing$	$\varnothing$
2	$\{\$\}$	$\varnothing$	$\{b,c,d\}$	$\varnothing$
3	$\{\$\}$	$\{\$\}$	$\{b,c,d\}$	$\varnothing$
4	$\{\$\}$	$\{\$\}$	$\{b,c,d\}$	$\{\$\}$

בניית טבלאות goto ו-action עבור מנתח SLR:

	actions					goto		
	a	b	c	d	$\$$	S	A	B
0	s1					2		
1		r4	r4	r4			3	
2					acc			
3		s5	s6					4
4					r2			
5		r3	r3	r3				
6		r4	r4	r4	r5		7	
7		s5		s8				
8					r6			

נשים לב שבעמודה המתאימה לטרמינל a אין אף פעולת reduce – הטרמינל אינו מופיע ב-follow של אף משתנה.

6.6.3. Canonical LR

אלגוריתם Canonical LR מבוסס על מנתח LR בדומה לאלגוריתמים הקודמים. הרעיון של האלגוריתם הוא "לפרק" את המצבים של LR(0) למצבים עדינים יותר, כאלה המכילים גם מידע lookahead.

לשם כך נגדיר מהו פריט LR(1), ונגדיר את פונקצית הסגור עבור פריטי LR(1). מעבר לכך, שאר האלגוריתם נותר ללא שינוי.

הגדרה: **פריט LR(1)** מורכב מזוג סדור: פריט LR(0) ואסימון (או סימן סוף הקלט, $).

מכלל גזירה עם n רכיבים מצד ימין, בדקדוק בו קיימים t אסימונים, ניתן לקבל $(n+1)\cdot(t+1)$ פריטי LR(1).

משמעותו של פריט LR(1): פריט מסמל את מצבו של ה-parser. משמעותו: זיהינו את מה שנמצא משמאל לנקודה; אנו מצפים כעת למצוא את מה שנמצא מימין לה, ולאחר מכאן את האסימון המצורף לפריט.

6.6.3.1. *סגור של פריטי LR(1)*

הגדרה: **קבוצת הסגור של קבוצת פריטי LR(1)**: קבוצת פריטי LR(1) שבה:

- עבור כל פריט LR(1) מהצורה $[A \to \alpha \cdot B\beta, a]$, הפריט נמצא בקבוצת הסגור
- עבור כל כלל מהצורה $B \to \delta$ וכל אסימון b בדקדוק (כולל $), כך ש- $b \in first(\beta a)$, גם הפריט $[B \to \cdot\delta, b]$ נמצא בקבוצת הסגור.

המצב הראשון מתקבל מסגור של הפריט $[S' \to S, \$]$.

6.6.3.2. בניית הטבלאות

בדומה לאלגוריתמים הקודמים, מתחילים מטבלת המעברים של האוטומט.

- הופכים כל מעבר בעמודה של אסימון לפעולת shift.
- עמודות המשתנים הן טבלת ה-goto.
- ה-acc מושם בעמודת $, בשורה של כללים המכילים את הפריט $[S' \to S, \$]$.
- עבור כל מצב המכיל פריט מהצורה [A → α·,a], וכלל A→α שמספרו (m>0) m, שמים reduce m בשורה של מצב זה, בעמודה של אסימון a.

6.6.4. LALR

האלגוריתם האחרון שנציג הוא אלגוריתם LALR. זהו אחד האלגוריתמים הפופולריים ביותר – הוא נותן איזון טוב בין כמות הדקדוקים שהוא מסוגל לנתח ובין גודל טבלאות הניתוח שהוא דורש. כלים רבים כגון YACC ו-BISON מייצרים מנתחי LALR.

בדומה ל-SLR, מנתח LALR זהו שינוי של השיטה ליצירת טבלאות LR(0). בשונה מ-SLR – בעוד ש-SLR משתמש בקבוצת follow כדי ליצר את פעולות ה-reduce, LALR משתמש בקבוצות lookahead. שיטה זו עדיפה מכיוון שהיא לוקחת יותר מההקשר בו נמצא המנתח לצורך החלטה על הפעולה הרצויה.

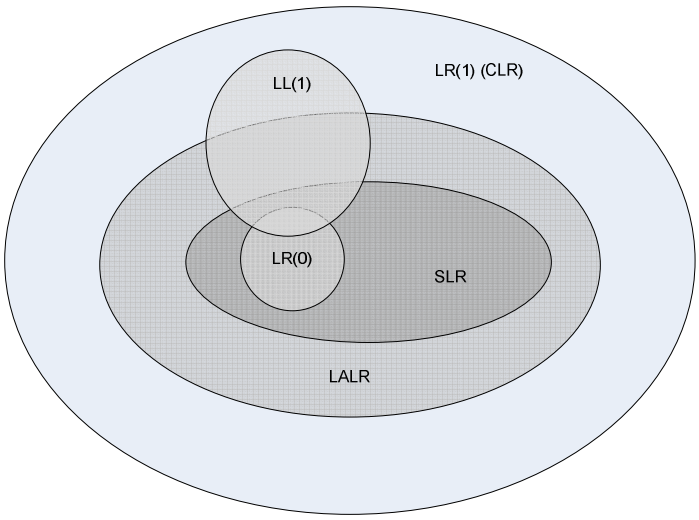
עבור כל פריט LR(1), נקרא לפריט ללא אסימון ה-lookahead בשם core.

דרך הפעולה:

1. בונים אוטומט LR(1).
2. אם בתוך שני מצבים באוטומט LR(1) הכללים זהים בהיטלם על LR(0) (כלומר אותם פריטים את מתעלמים מה-lookahead שבא אחרי הפסיק = ה-core שלהם זהה), מאחדים את המצבים למצב אחד.
3. אם מקבלים קונפליקט אומרים שהאוטומט לא LALR אלא רק LR(1). אחרת יש לנו אוטומט LALR המשמש כמנתח.

6.7 היררכיית הדקדוקים וסיכום

להלן היררכיית הדקדוקים השונים אותם הצגנו.



בניית הטבלה - מתי שמים פעולה X במצב כלשהו ו וטרמינל t

	LR(0)	SLR	LR(1)
shift	מ-I יוצאת קשת המסומנת t		
reduce	ב-I יש $A \to \alpha\bullet$	I מכיל פריט $A \to \alpha\bullet$ וגם $t \in follow(A)$	I מכיל פריט: $A \to \alpha\bullet, t$
accept	I מכיל פריט של $S' \to S\bullet$ וכן $t = \$$	I מכיל פריט של $S' \to S\bullet$ וכן $t = \$$	I מכיל פריט של $S' \to S\bullet, \$$ וכן $t = \$$

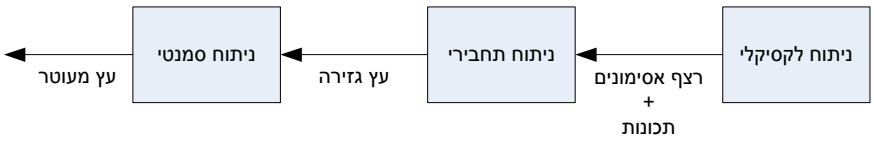
השוואה בין פריט LR(0) לפריט LR(1)

פריט LR(0)	פריט LR(1)
$A \to \alpha\bullet\beta$	$A \to \alpha\bullet\beta, t$
ראינו α	ראינו α
רוצים לראות β	רוצים לראות β
יתכון ואנחנו באמצע גזירה לפי $A \to \alpha\beta$	יתכון ואנחנו באמצע גזירה לפי $A \to \alpha\beta$, אבל בסוף הגזירה אנחנו מצפים לראות t.

7. שלב 3 - ניתוח סמנטי

7.1 פתיחה

תזכורת – שלבי הניתוח:



שלב הניתוח התחבירי והניתוח הסמנטי לרוב מבוצעים ביחד.

המנתח הסמנטי הוא השלב ש-"מבין" את התוכנית. הוא מנצל את הערכים הסמנטיים של הטרמינלים (שחושבו על ידי ה-screener) ואת המבנה ההיררכי של התוכנית (העץ שחושב על ידי הניתוח התחבירי) לשם הסקת מסקנות לגבי קטעים בתוכנית (תת-עצים) כולל התוכנית כולה. בנוסף משמש המנתח הסמנטי להרחבת יכולת הזיהוי של המנתח התחבירי (המוגבל לדח"ה).

קלט: עץ הגזירה של התוכנית (מהמנתח התחבירי), והערכים הסמנטיים של האסימונים (מהמנתח הלקסיקלי).

פלט:

- אם התוכנית חוקית: **עץ גזירה מעוטר** – עץ המכיל ערכי סמנטיים גם עבור החלקים התחביריים של התוכנית (כלומר – מופעים של משתנים בעץ הגזירה).
- אם התוכנית אינה חוקית: הודעת שגיאה מתאימה.

דוגמאות לפעולות סמנטיות:

- וידוא שכל משתנה שיש בו שימוש אכן הוגדר.
- התאמת טיפוסים.
- אי חלוקה באפס.
- דוגמאות רבות נוספות.

הצורך בבדיקות סמנטיות:

- חלק מההגבלות בשפות תכנות אינן ניתנות לביטוי בעזרת דקדוקים חסרי הקשר: התאמת טיפוסים, יחידות, התאמת שמות.
- צריך לטפל גם בתרגום לשפת המטרה, לא רק בזיהוי שפת המקור.

7.2. דקדוקי שדות ערך / הגדרות מונחות תחביר

דקדוקי שדות ערך (attribute grammers) או **הגדרות מונחות תחביר (syntax directed translation)** אלה 2 שמות שונים לפורמלים המשלב ניתוח תחבירי וניתוח סמנטי. שילוב זה מאפשר לבצע בדיקת שגיאות סמנטיות, וחישוב ערכים סמנטיים, בזמן הניתוח התחבירי.

מבנה הגדרה מונחית תחביר:

- "מבנה הצהרתי": הצמדת שדות ערך/תכונות לטרמינלים ולמשתני הדקדוק
- לכל כלל גזירה מצמידים כללים סמנטים, המגדירים ערכי תוכנות של המשתנים על ידי שימוש בערכי תכונות אחרות.

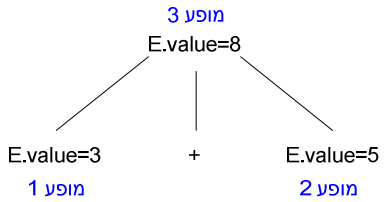
7.2.1. הגדרות בסיסיות

תכונה: טיפוס כלשהו מוגדר מראש, בעל שם מזהה, המוצמד למשתנה או טרמינל בדקדוק מסויים. התכונה מכילה את המידע הסמנטי. התכונה היא טיפוס (ולא ערך).

לדוגמא: למשתנה ID נצמיד תכונה מטיפוס string בשם name, ולמשתנה E נצמיד תכונה בשם value מסוג int.

מופע של תכונה: "instance" – ערך התכונה במופע מסויים של משתנה או טרמינל בעץ גזירה נתון.

לדוגמא – נביט בעץ הגזירה הבא:



הכלל המתאים לעץ זה הוא:

$$E \to E + E$$

כדי להבדיל בין המופעים השונים של העץ נסמפר את המופעים:

$$E \to E_1 + E_2$$

<u>סימון</u>: מופע של תכונה value של המשתנה E יסומן על ידי E.value. מופע התכונה name של המשתנה ID יסומן ID.name.

כלל סמנטי: פקודה המוצמד למקום כלשהו בכלל גזירה (יכול להיות גם באמצע הכלל), המגדירה ערך מופע תכונה על ידי מופעים אחרים של התכונה או של תכונות אחרות. לדוגמא נסתכל על הכלל:

$$A \to BC$$

כללים סמנטיים עבור כלל זה יכולים להיות:

$$A \to B\{B.q = A.p + 2;\}C\{A.p = 3;\}$$

בכלים שנציג בהמשך מסמך זה – הכללים הסמנטיים נכתבים בשפת C.

7.2.2. דוגמא להגדרה מונחית תחביר

<u>המטרה</u>: חישוב והדפסת ערך ביטוי אריתמטי.

<u>תכונות</u>: נצמיד תוכנית מטיפוס int בשם value למשתנה E ולטרמינל num. המנתח הלקסיקלי יספק עבורנו את הערך של התכונה עבור כל מופע של הטרמינל num.

הגדרה מונחית תחביר:

	כלל גזירה	כלל סמנטי
(1)	$S \to E$	print(E.value);
(2)	$E \to E_1 + E_2$	$E.value = E_1.value + E_2.value$
(3)	$E \to num$	$E.value = num.value$

7.2.3. סיווג תכונות סמנטיות לפי אופן חישובן

תכונות נוצרות (synthesized)

תכונות התלויות רק בבנים בעץ. בצורה פורמלית, בהנתן משתנה A, כל הכללים לחישוב A.p הם מהצורה:

$$A \to BCD\{A.p = \langle actions\ on\ B,C,D\ properties \rangle;\}$$

תכונות נורשות (inherited)

תכונות התלויות באבא ובאחים. בצורה פורמלית, בהנתן משתנה A, כל הכללים לחישוב A.p הם מהצורה:

$$B \to CAD\{A.p = \langle actions\ on\ B,C,D\ properties \rangle;\}$$

תכונות של אסימונים, המיוצרות ע"י המנתח הלקסיקלי, נחשבות לתכונות נוצרות מבחינה פורמאלית.

קיימות תכונות שאינן נוצרות או נורשות.

7.3. אלגוריתמים לחישוב הגדרות מונחות תחביר

7.3.1. סוגי הגדרות מונחות תחביר

1. **הגדרות מונחות תחביר מסוג S ("דקדוקי S")**: הגדרות מונחות תחביר המכילות תכונות נוצרות בלבד.
2. **הגדרות L-inherited**: הגדרות בהן כל תכונה נורשת של משתנה V באגף ימין של כלל כלשהו מחושבת על פי:
   - o תכונה נורשת של המשתנה באגף שמאל של הכלל, או:
   - o תכונה כלשהי של משתנה המופיע משמאל למשתנה הנוכחי בכלל.
3. הגדרות כלליות יותר.

מתקיים כי קבוצת ההמ"ת מסוג 1 מוכלת בקבוצת ההמ"ת מסוג 2, ושתיהן מוכלות בקבוצה 3.

הגדרות מונחות תחביר מסוג 1 ו-2 ניתן לחשב באמצעות כלי בשם **סכימת תרגום**, בו שזורים הכללים הסמנטיים בתוך כללי הדקדוק. עבור המ"ת מסוג 3 נשתמש באלגוריתם כללי יותר, שיחשב קודם את סדר חישוב מופעי התכונות, ורק אחר כך את הערכים עצמם.

7.3.2. סכימת תרגום

סכימת תרגום היא מבנה המגדר במדוייק את אופן חישוב התכונות הסמנטיות וסדר חישובן. במהלך ניתוח בעזרת סכמת תרגום הכללים הסמנטיים מתבצעים בדיוק בסדר בו הם כתובים, כלומר, בניתוח כלל גזירה מהצורה הבאה:

$$V \rightarrow \{\ \text{sementic rule 1}\ \}\ V_1 \rightarrow \{\ \text{sementic rule 2}\ \}\ V_2\ \{\ \text{sementic rule 3}\ \}$$

מבוצעות הפעולות הבאות (לפי הסדר):

1. ביצוע כלל סמנטי מס' 1
2. ניתוח (גזירת) המשתנה V_1.
3. ביצוע כלל סמנטי מס' 2.
4. ניתוח (גזירת) המשתנה V_2.
5. ביצוע כלל סמנטי מספר 3.

כדי שהסכמה תהיה תקינה (כלומר, כדי שנהיה מסוגלים לחשב את כל הערכים), נרשום כללים המחשבים תכונות נורשות של המשתנה V_1 לפני הופעתו בכלל הגזירה, וכללים המחשבים תכונות נותרות של המשתנה V רק בסוף כלל הגזירה.

$$V \rightarrow \{ \text{ inherited attributes of } V_1 \} \; V_1 \rightarrow \{ \text{ inherited attributes of } V_2 \} \; V_2 \; \{ \text{ synthesized attributes of } V \}$$

ניתוח סכמת תרגום תקינה מבקר כל מופע של כל משתנה בדיוק פעמים. בביקור הראשון הוא מחשב עבור המשתנה את התכונות שהוא יורש מההורה או מאחיו (השמאליים – L-inherited) בעץ הגזירה. בביקור השני הוא מחשב את כל התכונות הנוצרות של המשתנה, התלויות בתת העץ שהוא גוזר ובתכונות הנורשות שלו.

סכמת תרגום ודקדוקי S

כאמור דקדוקי S זוהי משפחה מצומצמת של דקדוקים ללא תכונות נורשות. דקדוקים אלה מתאימים במיוחד לניתוח תחבירי bottom-up, ללא בנית עץ הגזירה. במקרה זה פעולת הצמצום (reduce) היא "קולב" עליו תולים את ביצוע הפעולות הסמנטיות המתאימות לכלל הגזירה. ניתן למקם את שדות הערך במחסנית הניתוח התחבירי. דוגמא: חישוב ערך ביטוי בוליאני קבוע:

$$EXP \rightarrow const \{ EXP.val = const.val; \}$$
$$EXP \rightarrow (EXP_1) \{ EXP.val = EXP_1.val; \}$$
$$EXP \rightarrow EXP_1 \; or \; EXP_2 \{ EXP.val = (EXP_1.val \;||\; EXP_2.val); \}$$
$$EXP \rightarrow EXP_1 \; and \; EXP_2 \{ EXP.val = (EXP_1.val \;\&\&\; EXP_2.val); \}$$

כל תכונה בדקדוק תלויה רק בבנים – נוצרות. כאמור ניתן לחשב את ערכי התכונות הנוצרות בזמן ניתוח תחבירי בשיטת bottom-up. פעולות סמנטיות יתבצעו במקרה כזה בזמן reduce.

המגבלה בשימוש בתכונות נוצרות בלבד: פעולות סמנטיות יכולות להתבצע רק בסוף כללי הגזירה.

מרקרים: קיים הצורך לבצע פעולות סמנטיות גם באמצע כלל. כדי לפתור את הבעיה אפשר להשתמש ב**מרקרים** (משתני דמה). לדוגמא, אם נניח נרצה לכתוב:

$$A \rightarrow B \{ ... \} C$$

הדרך לכתוב זאת תהיה:

$$A \rightarrow BMC$$
$$M \rightarrow \varepsilon \{ ... \}$$

7.3.3. אלגוריתם כללי לחישוב הגדרות מונחות תחביר

סכימה כללית לשילוב הטיפול הסמנטי במהדר:

1. המנתח התחבירי בודק נכונות תחבירית ובונה את כל עץ הגזירה בזיכרון
2. מעץ הגזירה בונים גרף תלויות המיצג את הקשרים בין המשתנים הסמנטיים
3. מבצעים אנליזה של גרף התלויות על מנת למצוא "סדר טוב" לביצוע הפעולות הסמנטיות – סדר בו כל תכונה תהיה תלויה בערכי תכונות שכבר חישבנו.
4. לאחר שמצאנו סדר תקין - מפעילים את הפעולות הסמנטיות על פי הסדר שנבחר.

בניית גרף תלויות:

המשוואה לחישוב התכונות של A יוצרת תלות של A ב- $B_1, B_2, ..., B_n$. ע"מ לחשב את הערך A יש לחשב תחילה את ערכי $B_1, B_2, ..., B_n$. נסמן זאת כך: $A \leftarrow B_i$

- בהנתן עץ גזירה, ניתן לבנות **גרף תלויות** עבור כל מופעי שדות הערך בעץ.
- דקדוק שדות ערך G הוא **מוגדר היטב** אם לכל מילה בשפה המוגדרת על ידו, גרף התלויות הינו גרף חסר מעגלים מכוונים (הינו DAG). דבר זה מבטיח כי ניתן לחשב את כל התכונות בצורה תקינה.

ניתן לראות את אוסף כל הפעולות הסמנטיות המגדירות כמערכת משוואות עם שדות הערך כנעלמים

- המטרה הכללית של הניתוח הסמנטי היא לפתור את מערכת המשוואות. קיים פתרון כאשר הדקדוק מוגדר היטב.
- **פותר (solver)** – אלגוריתם לפתרון מערכת המשוואות.

8. YACC

Yacc הוא מחולל מנתחים תחביריים (parser-generator). המנתח התחבירי הנוצר ע"י Yacc עובד בשיטת LALR וכתוב בשפת C.

8.1. מבנה קובץ הקלט ל-Yacc

%{ C user declarations %}	**חלק 1:**
Declarations	**חלק 2:**
%%	
Rules	**חלק 3:**
%%	
C user routines	**חלק 4:**

חלק 1:

הגדרות רגילות ב-C. כל מה שמוגדר כאן מוכר ע"י המנתח התחבירי והלקסיקלי.

חלק 2:

הצהרות על אסימונים, עדיפות ואסוציאטיביות של אופרטורים.

```
% token name1 name2 ….
```

משמש להכרזה על השמות שמסמלים tokens כל השמות שלא יוגדרו תחת %token יחשבו למשתנים בדקדוק (non-terminal). כל non-terminal חייב להופיע בצד שמאל של חוק לפחות פעם אחת.

חלק 3:

Rules: חוקי הדקדוק שעבורו יוצר המנתח התחבירי. בכל חוק מכניסים את הפעולות הסמנטיות שיבוצעו בצורת קטע קוד ב-C.

חלק 4:

Programs: רוטינות שמשמשות את הפעולות הסמנטיות

8.2. חלק החוקים

המשתנה שבצד שמאל של החוק הראשון הוא המשתנה התחילי.

ניתן גם להגדיר את המשתנה התחילי באופן מפורש בחלק ההגדרות ע"י:

```
%start   symbol
```

כתיבת חוקים: החוק A→ BCD | EF | G הופך ל:

```
A     :     B  C  D;
A     :     E  F;
A     :     G;
```

ששקול ל:

```
A     :     B  C  D
          | E  F
          | G
          ;
```

תו שיופיע בין גרשיים ('c') יפורש כאסימון.

Action: לכל חוק ניתן לשייך פעולות סמנטיות בעזרת חלק הAction.

לדוגמא:

```
A    :    B C              {          printf("a message\n");
                                       flag = 25;                }
```

כל nonterminal יכול להחזיר ערך. בכדי לגשת לערך המוחזר ע"י הסימבול ה-n בגוף החוק יש להשתמש ב $n.
בכדי לקבוע את הערך המוחזר ע"י החוק יש לבצע: =$$.

דוגמא:

נביט בחוק הבא:

A → B C { some rule } ‘d’ E

אזי:

הסימן	$5	$4	$3	$2	$1	$$
מתייחס ל:	E	‘d’	{ some rule }	C	B	A

כאשר $3 אינו מוגדר!

דוגמאות נוספת:

- נביט בחוק:

```
expr    :       '('  expr  ')'        {  $$ = $2 ;  }
```

הסוגריים המסולסלות קובעות את הערך המוחזר של החוק.

- action באמצע החוק:

```
A       :       B {  $$ = 1;  } C { x = $2;   y = $3;  };
```

תוצאה:

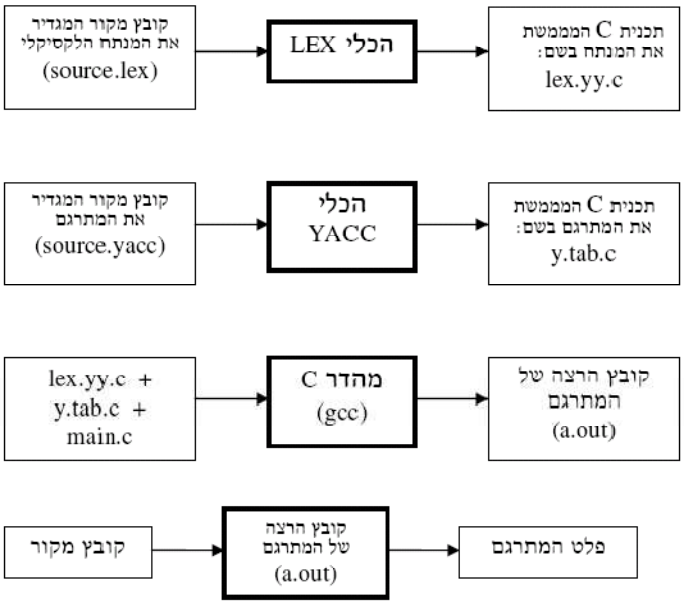
```
x=1
y=val( c )
```

וזאת מכיוון ש-YACC מיצר nonterminal חדש וחוק חדש המתאים למשתנה זה. הYACC מבין את הדוגמא הקודמת כאילו נכתבה:

```
$ACT    :       /* empty */ {  $$ = 1;  } ;

 A      :       B  $ACT  C {   x = $2;   y = $3;  }  ;
```

סכימת השימוש ב- YACC (בעזרת Lex)



שימוש במתרגם הנוצר:

דוגמא לקובץ קלט ל-YACC:

```
%{
#include <stdio.h>
#include <ctype.h>
%}

%token CONST

%%

S : E '\n' { printf("%d\n", $1); }
      ;
E : E '+' T { $$ = $1 + $3; }
      | T { $$ = $1; }
      ;
T :  T '*' F { $$ = $1 * $3; }
       | F { $$ = $1; }
       ;
F : CONST { $$ = yylval; }

%%

#include “lex.yy.c” // refers to the lexical analyzer prepared in, e.g., lex
```

8.3. קישור בין המנתח הלקסיקלי למנתח התחבירי

כדי לתקשר עם המנתח התחבירי, המנתח הלקסיקלי שנוצר על ידי Lex מכיר את הדברים הבאים:

1. מנקודת המבט של Lex כל מחרוזת שהוגדרה באמצעות token X% מתפרשת במנתח הלקסיקלי כקבוע. אם המנתח הלקסיקלי יחזיר את המזהה הקבוע X, המנתח התחבירי יבין שזוהה האסימון X.
2. מנקודת המבט של YACC: המנתח התחבירי מניח את קיומה של הפונקציה yylex() המחזירה בכל פעם קבוע המתאים לאחד האסימונים שהוגדרו על ידי token%.
3. המנתח הלקסיקלי והתחבירי מכירים שניהם את המשתנה הגלובלי yylval (שטיפוסו יוגדר בהמשך) המייצג את ערך אוסף התכונות ל האסימון האחרון שזוהה על ידי המנתח הלקסיקלי.

8.4. תכונות סמנטיות ב-YACC

התכונות של כל המשתנים של דקדוק מסויים הן כולן מאותו טיפוס.
טיפוס זה מוגדר על ידי המקרו YYSTYPE (המגדיר גם את הטיפוס של המשתנה yylval), שברירת המחדל שלו היא int. כדי להגדיר תכונות בעלות טיפוסים שונים עבור סוגי משתנים שונים יש להשתמש ב-struct.

דוגמא להצהרה על ה-struct:

```
typedef struct {
   string        name;
   int           val;
} yystype;

#define YYSTYPE yystype
```

8.5. קדימויות אופרטורים

כאשר אנחנו מגדירים את הדקדוק שלנו יתכן מצב בו יהיו התנגשויות shift-reduce, והוא מצב בו יש לנו ביטויים אריתמטיים. במצב כזה shift זו לא תמיד הפעולה הנכונה לבצע. על ידי הגדרת קדימויות בין אופרטורים נוכל להורות ל-YACC מתי לעשות shift ומתי לעשות reduce.

8.5.1. דוגמא לבעיה

נביט על הדקדוק הדו משמעי הבא:

```
expr:    expr '-' expr
       | expr '*' expr
       | expr '<' expr
       | '(' expr ')'
       ...
       ;
```

דקדוק זה הוא דו משמעי כי למשל לביטוי '3 * 2 – 1' יש 2 עצי גזירה שונים. נניח כי המנתח רואה את האסימונים '1', אז '-' ואז '2'. האם הוא צריך לעשות reduce עם כלל המינוס? התשובה תלויה באסימון הבא. אם האסימון הבא הוא '(' אז התשובה היא חיובית – אנחנו רוצים לעשות reduce. Shift הוא לא חוקי כי אין כלל שיכול לגזור '(2 -'. לעומת זאת - אם הכלל הוא '*' או '<' יש לנו בחירה: ניתן לבחור בין shift ל-reduce – שתי האפשרויות יעשו את העבודה אך יש לבחור ביניהן.

כדי לבחור את האפשרות הנכונה חייבים להתייחס לתוצאות ולמשמעות הפעולה. אם אנחנו מבצעים shift לאופרטור הבא op, התוצאה שנקב ל היא: '$1-(2\ op\ 3)$'. אם נעשה reduce לחיסור לפני ביצוע shift ל-op, התוצאה שתחושב בפועל היא '$(1-2)\ op\ 3$'. הבחירה צריכה להעשות בהתאם ל-op. אם למשל op הוא פעולת הכפל, יש לבצע shift, אבל אם האופרטור הוא '<' יש לבצע reduce.

כאשר הקלט הוא '5 – 2 – 1' האם נרצה לפענח אותו בתור '5 – (2 – 1)' או בתור '(5 – 2) – 1'? לרוב אנחנו נעדיף את האפשרות הראשונה. אפשרות זו היא **אסוציאטיביות שמאלית**. האפשרות ההפוכה, המכונה **אסוציאטיביות ימנית**, מועדפת כאשר האופרטור היה אופרטור השמה, למשל.

8.5.2. הגדרת הקדימויות

הגדרת הקדימויות נעשת על ידי ההצהרות %left ו-%right. כל אחת מהצהרות אלו מכילת רשימת אסימונים, שהם האופרטורים שאת האסוציאטיביות שלהם אנחנו מגדירים.

- %left מגדיר את כל האופרטורים ברשימה כאופרטורים בעלי אסוציאטיביות שמאלית.
- %right מגדיר את כל האופרטורים ברשימה כאופרטורים בעלי אסוציאטיביות ימנית.
- האפשרות השילישית היא %nonassoc המגדירה כי זו שגיאה תחבירית למצוא את אותו האופרטור פעמיים ברצף.

העדיפות היחסית בין האופרטורים נשלטת על ידי הסדר שבו הם מוגדרים. הצהרת ה-%left או ה-%right הראשונה מגדירה את האופרטורים בעלי העדיפות הנמוכה ביותר, ההצהרה הבאה מגדירה אופרטורים עם עדיפות גבוהה יותר, וכך הלאה.

דוגמאות:

בדוגמת הפתיחה שהצגנו, היינו רוצים להגדיר את 3 השורות הבאות:

```
%left '<'
%left '-'
%left '*'
```

3 האופרטורים הם בעלי אסוציאטיביות שמאלית. כפל הוא האופרטור בעל העדיפות הגבוהה ביותר.

בדוגמא יותר מלאה התומכת באופרטורים נוספים היינו מחלקים את האופרטורים לקבוצות בעלות עדיפות שווה:

```
%left '<' '>' '=' NE LE GE
%left '+' '-'
%left '*' '/'
```

- הערה: NE פירושו "לא שווה". אנחנו מניחים שאסימונים עם יותר מתו אחד מיוצגים על ידי שמם ולא הליטרלים המרכיבים אותם.

8.5.3. איך מערכת הקדימויות עובדת?

ההשפעה הראשונה של הכרזת הקדימויות היא להגדיר רמות של קדימויות לטרמינלים הקיימים.

ההשפעה השניה היא הגדרת קדימויות לחוקים השונים: כל חוק מקבל את הקדימות שלו מהטרמינל האחרון שבו.

לבסוף – פתרון של קונפילקטים נעשה על ידי השוואת הקדימויות של החוקים ובהתחשב ב lookahead token – אם עדיפות ה-token גבוהה יותר, הבחירה היא ב-shift. אם הקדימות של החוק גבוהה יותר, הבחירה היא reduce. אם הקדימות של החוק והאופרטור שווה הבחירה נעשית על פי האסוציאטיביות של רמת קדימיות זו.

לא לכל החוקים והאסימונים חובה שתוגדר קדימות. אם לחוק לא מוגדרת קדימות וישנה התנגשות, ברירת המחדל היא shift.

8.6. הפעלת YACC

```
lex pascal.lex
yacc pascal.yacc
gcc -g  y.tab.c –ll

./a.out < input.txt
```

חשוב להוסיף שיגרת טיפול בשגיאות:

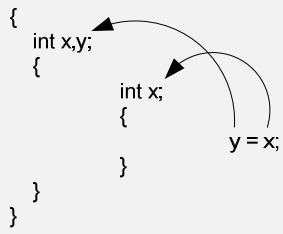
```
yyerror ( char *s )
{
   fprintf(stderr, "%s\n", s);
}
```

9. בניית טבלאות הסמלים

9.1 Scoping

Static Scoping: המופע של המשתנה אליו יש לגשת הוא המופע הקרוב ביותר בשרשרת הקינון הסטטי.

ב-Static Scoping ניתן לשייך כל גישה למופע מסוים בזמן קומפילציה:



Dynamic Scoping: הקביעה אל איזה משתנה ניגש נקבעת בזמן ריצה. המשתנה שהוגדר אחרון (לפי ריצת התוכנית הנוכחית) הוא זה שאליו ניגש.

דוגמא: נביט בקוד דמוי C הבא:

```
int x = 0;
int f() { return x; }
int g() { int x = 1; return f(); }
```

כאשר נעבוד ב-static scoping, קריאה ל-g תחזיר את הערך 0. בזמן ההידור הביטוי x בפונקציה f מוגדר להתייחס ל-x הגלובלי, שאינו מושפע מהמשתנה הלוקלי x של הפונקציה g.

כאשר נעבוד ב-dynamic scoping קיימים במחסנית 2 משתנים בשם x: המשתנה x הגלובלי והמשתנה x של g (שעדיין קיים על המחסנית מכיוון ש-g לא הסתיימה). מכיוון שהמשתנה x של g נמצא בראש המחסנית קריאה ל-g תביא במקרה כזה את הערך 1.

בפועל רוב השפות משתמשות ב-static scoping.

9.2 ארגון הזכרון על ידי המהדר

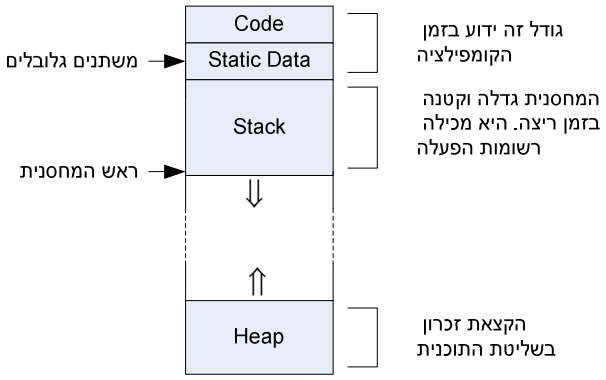
נציג את ארגון הזכרון על ידי המערכת כאשר אנחנו עובדים ב-static scoping.

- לכל משתנה מוקצת כתובת. השאלה מה היא כתובת הזו.
- בנוסף – מה קורה למשתנה לוקלי כאשר התחום שלו מסתיים?

גורמים נוספים המשפיעים על ארגון הזכרון על ידי המהדר:

- האם הפונקציות בשפה יכולות להיות רקורסיביות?
- מה קורה לערכים של משתנים לוקלים כשהקריאה הנוכחית לפונקציה מסתיימת.
- האם פונקציה יכולה להתייחס למשתנים הנמצאים מחוץ לה.
- איך מועברים פרמטרים לפונקציה.
- האם ניתן להקצות זכרון דינמי? מי אחראי לשחרר זכרון כזה?
- נושאים נוספים...

ארגון הזכרון:



הכתובות של משתנים מקומיים:

- בכל כניסה לפונקציה, מוסיפים רשומת הפעלה למחסנית. בכל יציאה, מסירים רשומת הפעלה.
- משתנים מקומיים הם חלק מרשומת ההפעלה של כל פרוצדורה.
- כתובת של משתנה מקומי: S-a, כאשר S היא כתובת ראש המחסנית (משתנה בזמן ריצה), ו-a היא הכתובת היחסית של המשתנה המקומי. הכתובות היחסיות נקבעות בזמן קומפילציה!

9.3 טבלת הסמלים

טבלת סמלים:

- מבנה נתונים בזמן קומפילציה שמחזיק מידע על הסמלים בתכנית. **סמלים**: שמות המשתנים והפונקציות.
- המבנה צריך לתמוך ב-static scoping.
- המבנה צריך לאפשר חיפוש יעיל של סמלים.

מימוש טבלאות הסמלים:

- לכל scope ניצור טבלה נפרדת.
- המבנה המתקבל הוא עץ של טבלאות.
- כדי למצוא משתנה x:
  - חפש בטבלת הסמלים של ה-scope הנוכחי. אם מצאת, עצור. אחרת:
  - חפש בטבלת הסמלים של האבא של ה-scope הנוכחי. אם מצאת, עצור. אחרת המשך את החיפוש עד לשורש.

בניית עץ טבלאות הסמלים:

- הבנייה מתבצעת בזמן קומפילציה.
- נשתמש במחסנית שמחזיקה את שרשרת הטבלאות מה-scope הנוכחי עד לשורש.
- על מנת למצוא משתנה, מספיק לבדוק רק טבלאות במחסנית (כלומר – לטפס במחסנית עד שנמצא את המשתנה).
  - בכניסה ל-scope ניצור טבלה חדשה ונדחוף למחסנית.
  - ביציאה מ-scope נוציא טבלה מראש המחסנית.

9.4 סכימת תרגום

9.4.1 מבוא

נציג כיצד אנחנו מתרגמים דקדוקים המזכירים C ובונים מהם את טבלת הסמלים.

נניח שמבנה הרשומה בטבלת הסמלים הוא:

name type offset

- offset – המיקום היחסי ברשומת ההפעלה של הפונקציה. מתחיל בערך 0.

נשתמש בשתי מחסניות:

- tables: מחסנית טבלאות הסמלים.
  - מחסנית זו תשמור את המסלול הנוכחי בעץ טבלאות הסמלים של הפונקציות. הפונקציה המנותחת כעת תופיע בקצה (top) המחסנית.
- offsets: מחסנית של ה-offset-ים הנוכחיים בכל scope במסלול לשורש.
  - מחסנית של integers.
  - עבור כל איבר ב-tables יופיע איבר במחסנית offsets שיציין את סכום ההיסטים של כל המשתנים שנותחו עד עתה בפונקציה המנותחת.

נניח שקיימות לנו פונקציות הבאות:

- maketable(parent): יוצרת טבלה חדשה ריקה שהיא בת של parent בעץ.
- insert(table, name, type, offset): מכניסה משתנה לטבלת הסמלים.
- push, pop, top: פעולות סטנדרטיות למחסנית.

9.4.2. דוגמא

משתנים:

- Prog: גוזר תכנית שלמה
- St: גוזר פקודה או רצף פקודות (statements)

הדקדוק הינו חלקי ביותר. מוצגים כללים ספורים להצגת הרעיון: (ביטויים טרמינלים מודגשים בקו)

```
Prog    →int main ( ) { St }
St      → varitype id ;
St      → { St }
St      → print id ;
```

כעת נראה איך משולבת טבלת הסמלים:

```
Prog    →int main ( ) { M St }
                {
                        pop(tables);
                        pop(offsets);
                }
M       →ε
                {
                        t = maketable(null);
                        push(t, tables);
                        push(0, offsets);
                }
```

- נשים לב שהסוגריים המסולסלות הראשונות ב-MAIN הן אסימון ולא כלל סמנטי.
- M הוא Marker לאתחול הטבלאות.
- האתחול הראשוני יוצר טבלה שאין לה אב, ודוחף 0 בתור ה-offset ההתחלתי.

נמשיך בדוגמא:

```
St      → { N St }
            {
                    pop(tables);
                    pop(offsets);
            }
N       → ε
            {
                    t = maketable(top(tables));
                    push(t, tables);
                    push(top(offsets), offsets);
            }
St      → varitype id ;
            {
                    insert(top(tables), id.name, vartype.type, top(offsets));
                    top(offsets) += vartype.size;
            }
```

- N הוא מרקר נוסף. בתחילת כל בלוק הוא יוצר את הטבלה החדשה ומוסיף אותה למחסנית.
- בסוף הבלוק מבוצע POP של טבלת הסמלים ושל ה-offset.
- כאשר מגדירים משתנה חדש אנחנו שומרים את מיקומו והמידע עליו בטבלת הסמלים.

לסיום הכלל האחרון: נרצה לבדוק האם משתנה הוגדר כשמנסים להדפיס אותו:

```
St      → print id ;
            {
                    found = false;
                    for (int i = 0; i < tables.size(); i++)
                    {
                            t = tables.get(i);          // get i'th table from top
                            if (t.contains(id.name))
                            {
                                    found = true;
                                    break;
                            }
                    }
                    if (found == false) error("use of undef variable", id.name);
            }
```

10.תרגום לשפת ביניים

10.1. מבוא

עד כה הצגנו את ה-front end של תהליך הקומפילציה. תהליך ה-front-end הוא תלוי שפה. בסוף תהליך זה אנחנו מקבלים את העץ המעוטר, וכעת מתחיל שלב ה-back-end. שלב זה משתמש בנתונים המתקבלים מה-front-end והוא כבר אינו תלוי בשפת המקור.

בשלב התרגום לקוד הביניים אנחנו מתרגמים את הקוד לשפה שאינה שפת היעד, בדרך לביצוע אופטימיזציות נוספות.

מדוע לא לתרגם ישר לשפת היעד?

- תרגום לשפת ביניים מאפשר פיתוח מהיר יותר של קומפיילר למערכת חדשה. נדרש במקרה כזה לכתוב רק Back-end.
- פיתוח אופטימיזציות כלליות הפועלות על קוד הביניים.

10.2. שפת הרביעיות

שפת הביניים אותה נציג: **שפת הרביעיות**. זוהי שפת ביניים דמויית שפת אסמבלר בה כל פקודה מורכבת לכל היותר מארבעה אלמנטים: אופרטור (הפעולה) ושלוש כתובות (האופרנדים). **קוד 3-כתובות** הוא שם נוסף מקובל לשפה זו.

נעבוד עם 5 סוגי פקודות בלבד:

פעולה אריתמטית	t1 := t2 + t3
קפיצה לא-מותנית Label זוהי כתובת קבועה.	goto label
קפיצה מותנית relop זהו אחד מאופרטורי ההשוואה (<=, ==, ...) Label זוהי כתובת קבועה	if t1 relop t2 goto label
תווית ניתן להצמיד יותר מתווית אחת לאותה הפקודה.	Label:
משפט העתקה	x := y

ההגדרה המלאה של שפת הרביעיות מכילה פקודות רבות נוספות: לדוגמא – פקודות לעבודה נוחה עם פונקציות, פקודות לעבודה עם מערכים וכו'. לצורך פשטות ההסבר נדגים את פקודות אלו בלבד.

דוגמא:

הביטוי הבא:

$$a = b + c + d$$

ייוצג באופן הבא בשפת הביניים:

$$t_1 := b + c$$
$$t_2 = t_1 + d$$
$$a := t_2$$

- t_1, t_2 הם משתנים זמניים.
- נשים לב כי t_2 אינו הכרחי באמת. זה משתנה שבאופטימיזציה של הקוד יכול להעלם.

טעויות נפוצות בהבנת התחביר של השפה:

- בשפת ביניים אין else. לא קיים ביטוי ... if ... else
- התנאי בקפיצה אינו מורכב – לא ניתן לעשות && או || בתנאי ה-if.
- ניתן לקפוץ רק לכתובות קבועות. goto x כאשר x משתנה זו אינה פקודה חוקית.

10.3. יצירת קוד הביניים – תרגום לשפת הרביעיות

המטרה: לכתוב תרגום מונחה תחביר שיתרגם את קוד המקור לשפת הרביעיות.

- את תוצאת התרגום של כל חלק בתוכנית נשמור **בתכונה code** של המשתנה המתאים.
- לביטויים נצטרך בנוסף לדעת היכן תשמר תוצאת החישוב. **התכונה place** תחזיק ייצוג של המשתנה אשר בזמן הריצה יחזיק את תוצאת החישוב. גם למשתני התוכנית המקורית (כלומר לטרמינל id) תהיה התכונה place.
- בהתאם לתרגום שנכתוב, ייתכן ונצטרך תכונות נוספות.

הכלים העומדים לרשותנו:

- **הפונקציה gen**: מקבלת מחרוזת המתארת פקודה ומחזירה פקודה.
- **הפונקציה newlabel**: מחזירה מחרוזת המתאימה לתוית חדשה.
- **הפונקציה newtemp**: מחזיר ייצוג (למשל שם) של משתנה זמני חדש.
  השימוש שלנו במשתנה יהיה בקוד הביניים.
- **האופרטור ||**: משרשר קטעי קוד.

דוגמא:

$E \rightarrow E_1 \text{ '+' } E_2$

```
{
        E.place = newtemp();
        E.code  = E1.code || E2.code || gen(E.place '=' E1.place '+' E2.place);
}
```

בעמוד הבא נפרש מעט את הדוגמא.

ראשית אנחנו שומרים מקום עבור התוצאה:

```
E.place = newtemp();
```

המשמעות של הביטוי הבא:

```
E.code  = E1.code || E2.code || gen(E.place '=' E1.place '+' E2.place);
```

היא:

$$t_1 \leftarrow E_1$$
$$t_2 \leftarrow E_2$$
$$t_3 := t_1 + t_2$$

וכך מחושב הערך לתוך E.place.

10.3.1. תרגום פשוט

בהרבה מקרים אפשר להציע תרגום (= קוד ביניים רצוי) כך שלכל קטע קוד (המתאים למשתנה מסויים) נקודת כניסה אחת (בראשיתו) ונקודת יציאה אחת (בסופו). התוצאה היא שבתום ביצוע קטע קוד מבוצע קטע הקוד ששורשר אחריו.

עבור קוד ביניים המקיים תכונה זו אפשר לכתוב תרגום מונחה תחביר בו אין צורך להעביר מידע על תוויות, עובדה המפשטת את התרגום.

קושי המתעורר: ביטוי בו נראה שיש לכאורה יותר מנקודת יציאה אחת:

```
if some_expression then
        some_code_1
else
        some_code_2
```

לכאורה מהבדיקה של some_expression יש יותר מנקודת יציאה אחת: במקרה שהתנאי מתקיים יש לבצע את some_code_1 ובמקרה והתנאי לא מתקיים יש לבצע את some_code_2. השאלה המתעוררת היא מי משניהם יהיה אחרי קטע הקוד של בדיקת התנאי.

הפתרון לשאלה הוא זה: קטע הקוד של בדיקת התנאי **יחשב** את התנאי לתוך משתנה זמני. (לקטע קוד זה יציאה אחת בסופו). קטע הקוד שלאחריו יבדוק את המשתנה הזמני וימשיך לביצוע some_code_1 או some_code_2 על פי ערכו.

התוצאה תראה כך:

```
        <code that evaluate some_expression into t13>
        if t13 == 0 goto L54
        <code for some_code_1>
        goto L55
L54:
        <code for some_code_2>
L55:
```

קצת דוגמאות כיצד נכתוב את הכללים הסמנטיים עבור מבנים מורכבים:

$E \to E_1 \ ' > ' \ E_2$	label tmplabel = newlabel();
	E.place = newtemp();
	E.code = E1.code || E2.code ||
	gen(E.place '= 0') ||
	gen('if' E1.place '<=' E2.place 'goto' tmplabel) ||
	gen(E.place '= 1') ||
	gen(tmplabel ':');
	
	אנו שמים ב-E את הערך 0. אם ה-if לא מתקיים, כלומר התנאי כן מתקיים, אנחנו שמים ב-E את הערך 1.

$S \to \text{if } E \text{ then } S_1 \text{ else } S_2$	label falseLabel = newlabel(), endlabel = newlabel();
	S.code = E.code ||
	gen('if' E.place '== 0' 'goto' falseLabel) ||
	S1.code ||
	gen('goto' endlabel) ||
	gen(falseLabel ':') || S2.code ||
	gen(endLabel ':');
	אם הערך של E הוא false (0) אנחנו קופצים לביצוע S2 ומסיימים.
	אחרת אנחנו ממשיכים סידרתית, מבצעים את S1 וקופצים לסיום.

$S \to \text{while } E \text{ do } S_1$	label beginLabel = newlabel(), endLabel = newlabel();
	S.code = gen(beginLabel ':') || E.code ||
	gen('if' E.place '== 0' 'goto' endLabel) ||
	S1.code ||
	gen('goto' beginLabel) ||
	gen(endLabel ':');

10.3.2. תרגום לשפת הביניים - שיטת תוויות נורשות

לא תמיד ניתן (לפחות לא באופן נוח) לדאוג ליציאה אחת מכל קטע קוד. למשל, כשנרצה לממש break מלולאה או switch. לפיכך אנו מציגים את שיטת התוויות הנורשות.

הרעיון: יש כמה יציאות מקטע הקוד של משתנה כלשהו, ומהיכן בדיוק זה חלק מהסמנטיקה של אותו משתנה. לכן כל משתנה דקדוק יודע מידע זה באופן עצמאי. לעומת זאת – לאן צריך להמשיך מכל יציאה זה חלק מהסנטיקה של הסביבה (האבא) של המשתנה, לכן המשתנה לא יכול להסיק מידע זה לבד.

בשיטת תוויות נורשות כל משתנה מכיל תכונות נורשות בשביל נקודות היציאה. התכונות תהינה מסוג תווית ומשמעותן "היכן להמשיך אם נבחרה נקודת יציאה מסויימת". אבא של המשתנה יוריש לו את התוויות הנ"ל.

כאשר משתמשים בתוויות נורשות אפשר להמנע מחישוב מפורש של ערכי תנאים בוליאניים.

דוגמא

המשתנה B יגזור ביטוי בוליאני. למשתנה B שתי נקודות יציאה אפשריות: אחת true ואחת false בהתאם לערך הביטוי.

$B \to E_1 \text{ '>' } E_2$	B.code =	E1.code || E2.code || gen('if' E1.place '>' E2.place 'goto' B.true) || gen('goto' B.false);
$B \to B_1 \text{ or } B_2$	B1.true = B1.false = B2.true = B2.false = B.code =	B.true; newlabel(); B.true; B.false; B1.code || gen(B1.false ':') || B2.code;

נעמיק בדוגמא השניה:

ראשית נעשה את ההבחנות הבאות:

- אנחנו מדברים על תכונות נורשות, כלומר:

B מקבל כתובת מהאב שלו – לאן ללכת במקרה של הצלחה ולאן ללכת במקרה של כשלון.

- אפשר להסתכל על B.true ו-B.false כעל מצביעים – המכילים כתובת ממנה אפשר להמשיך.

```
B1.true =       B.true;
```

אם B1 מתקיים אז מתקיים B ולכן נקפוץ לאן ש-B.true מצביע.

```
B2.true =       B.true;
```

באופן דומה, אם B2 מתקיים אז מתקיים B ונקפוץ לאן ש-B.true מצביע.

2 השורות הבאות הן הכנה לשורה האחרונה:

```
B1.false =      newlabel();
B2.false =      B.false;
```

מוגדרת תווית חדשה אליה נקפוץ אם B1 לא מתקיים, ומוגדר שאם B2 לא מתקיים נקפוץ לאן ש-B.false מצביע.

השורה הסופית דואגת לנכונות:

```
B.code =        B1.code || gen(B1.false ':') || B2.code;
```

- מבוצע B1. אם B1 מתקיים הרי שנקפוץ ל-B1.true שהוא B.true.
- אם B1 לא מתקיים, אנחנו קופצים לתוית החדשה (שמופיעה מיד אחרי B1). מיד לאחר מכן מבוצע B2.
- אם B2 מתקיים נקפוץ ל-B.true.
- אחרת, נקפוץ ל-B.false.

לצורך השוואה עם הקוד הקודם נציג את דרך התרגום עבור if ו-while בשיטת התוויות הנורשות.

נשים לב כי לקטע קוד של משתנה S צריכה להיות יציאה אחת (עפ"י התכונה next), אלא ש-S יכול לצאת **גם** אל הפקודה הבאה (וזאת בשביל לא לסרבל את התרגום של פקודות בסיסיות).

$S \rightarrow \text{if } B \text{ then } S_1 \text{ else } S_2$	B.true =
	B.false =
	S1.next =
	S2.next =
	S.code =
		newlabel();
		newlabel();
		S.next;
		S.next;
		B.code ||
		gen(B.true ':') || S1.code ||
		gen('goto' S.next) ||
		gen(B.false ':') || S2.code;

$S \rightarrow \text{while } B \text{ do } S_1$	S1.next =
	B.true =
	B.false =
	S.code =
		newlabel();
		newlabel();
		S.next;
		gen(S1.next ':') || B.code ||
		gen(E.true ':') || S1.code ||
		gen('goto' S1.next);

מומלץ לקורא לעבור שורה-שורה בתרגום מונחה התחביר ולראות שהלוגיקה ברורה.

<u>החסרון</u> הגדול של שיטת התוויות הנורשות הוא בכך שהתרגום המתקבל עשוי להיות תרגום בו סדר החישוב לא ידוע בזמן קומפילציה, ולכן הקומפיילר יאלץ למצוא (לכל תוכנית מחדש) סדר חישוב בעזרת בניית גרף תלויות. תהליך זה מאריך את זמן הקומפילציה ומעלה את דרישות הזכרון של הקומפיילר.

הפתרון הוא להמנע מהורשת התוויות. אולם – כבר אמרנו כי המשתנה לא יכול לדעת באופן עצמאי לאן לקפוץ. לכן ההמנעות מהורשה מחייבת היפוך תפקידים:

- כרגע: האב אומר לבן "נקודת היציאה שלך exit מופנית לתוית label".
- המצב החדש: הבן אומר לאב: "אם הייתי יודע לאן exit מופנה, הייתי רושם ערך זה 'כאן, כאן וכאן'. מכיוון שאתה לא מיידע אותי בערך זה, האחריות למלא את החורים שהשארתי היא עליך".

Backpatch .10.3.3

נסכם מה שראינו עד כה:

- "תרגום פשוט" (ללא צורך בהעברת תויות ממקום למקום) אפשרי כאשר כל יחידה תחבירית מתורגמת לקטע קוד עם כניסה אחת (בראשיתו) ויציאה אחת (אל הפקודה שאחריו).
- כאשר צריכים יותר מנקודת יציאה אחת אפשר להעביר את התוויות כתכונות נורשות (שיטת "תויות נורשות").
  - o החסרון של השימוש בתוויות נורשות הוא שהתרגום מונחה התחביר המתקבל עלול לדרוש שימוש בגרף תלויות לצורך הקומפילציה.
  - o חסרון נוסף של תויות נורשות הוא הצורך לצבור את כל הקוד הנוצר בתכונות ורק בסוף להדפיסו.

שיטת backpach (הטלאה לאחור) היא שיטה לייצור קוד ביניים המאפשרת יותר מנקודת יציאה או כניסה אחת לכל קטע קוד, מבלי להשתמש בתכונות נורשות.

גם כאשר שאר התכונות של משתני הדקדוק הן L-inherited (כלומר, ניתן לתאר את התרגום מונחה התחביר כסכימת תרגום) יש לשיטה יתרון: אין צורך לצבור את הקוד שנוצר וניתן להוציאו לפלט באופן מיידי.

הרעיון, כאמור, הוא שבמקום שהאב אומר לבן "נקודת היציאה שלך exit מופנית לתוית label", הבן אומר לאב: "אם הייתי יודע לאן exit מופנה, הייתי רושם ערך זה 'כאן, כאן וכאן'." ואחריות האב היא למלא את חורים אלה.

<u>איך השיטה עובדת?</u>

1. הפונקציה emit מוציאה פקודה ל"פלט". הפקודה מורחבת כך שהיא מסוגלת לשלוח גם פקודות הפנית בקרה (if, goto, קריאה לפונקציה) עם חור במקום יעד.
2. את הכתובות (מספרים סידוריים בקוד הביניים) של הכתובות אנחנו אוספים ב-"**רשימת התחייבויות**", העוברת כתכונה נוצרת מהבן לאב. (כלומר, כאשר הבן יודע מה הכתובת הנכונה, הוא מודיע זאת לאב). לכל נקודת יציאה (למעט "נפילה" לכתובת שאחרי) תהיה רשימת התחייבות משלה. פונקציות עזר עבור ניהול הרשימה:
   a. הפונקציה nextquad מחזירה בכל רגע נתון את הכתובת של הפקודה הבאה (המספר הסידורי של הפקודה הבאה שתכתב). נשתמש בפונקציה זו בתהליך.
   b. הפונקציה makelist(quad) יוצרת רשימת התחייבויות חדשה שמכילה פריט בודד – quad.
   c. הפונקציה merge(list1, list2, …) ממזגת את הרשימות המתקבלות ומחזירה רשימה ממוזגת.

3. פקודות הבקרה השונות ישתמשו בכתובות (ולא בתוויות כפי שהוצג עד כה) לציון כתובת היעד.
4. הפונקציה backpatach מקבלת רשימת התחייבויות (רשימה עם כתובות) וכתובת יעד (אחת) ותמלא את כל החורים (האלמנטים ברשימת ההתחייבויות) בכתובת היעד.
5. אסור לאבד חורים (להשאיר חור ריק) ואסור למלא חור פעמיים.
6. כמו קודם, לביטויים יש את התכונה place, וכמו כן ביטויים בוליאנים אפשר לממש כביטוי רגיל או בהפניות בקרה.

דוגמא – ביטוי בוליאני

ביטוי בוליאני ממומש בהפניות בקרה. ל-B שתי תכונות נוצרות: falselist ו-truelist – רשימת ההתחיבויות עבור נקודת היציאה false ונקודת היציאה true (בשפה של בני אדם: רשימת המקומות בהם צריך לכתוב את הכתובת אליה קופצים במקרה של הצלחה, ורשימת המקומות בהם צריך לכתוב את הכתובת אליה קופצים במקרה של כשלון), בהתאמה.

$B \rightarrow E_1 \text{ '>' } E_2$

```
{
        B.truelist=makelist(nextquad());
        emit('if' E1.place '>' E2.place 'goto ___');
        B.falselist=makelist(nextquad());
        emit('goto ___');
}
```

נעבור שורה שורה על מנת להבין את הכתוב.

הנחה חשובה: E1, E2 כבר נכתבו לפלט. אנחנו תמיד מניחים זאת, ואנחנו שומרים על שמורה זו כל הזמן – כאשר B יסתיים, גם B יכתב כבר לפלט. באמצעות ההנחה אנחנו מסוגלים להשתמש ב-E1, E2.

```
B.truelist=makelist(nextquad());
```

- יוצרים את רשימת ה-truelist של B. הכתובת הראשונה שתצטרך הטלאה היא השורה הבאה.

```
emit('if' E1.place '>' E2.place 'goto ___');
```

- אנחנו רואים שאכן בשורה הבאה יש חור. (___ מסמן עבורנו חור).
- הנחה חשובה – הפונקציה המטליאה backpatch יודעת, בהנתן שורה, להזין את הכתובת למקום הנכון באותה שורה.
- אנחנו בעצם אומרים: בהטלאה הכתובת לקפיצה במקרה שהתנאי מתקיים, תוכנס לכאן.

```
B.falselist=makelist(nextquad());
emit('goto ___');
```

- נשים לב שאנחנו מגיעים לשורות אלו רק אם התנאי לא מתקיים. אנחנו מבצעים goto לכתובת הכשלון, שתמולא בהטלאה.

מאוד מקובל להשתמש במרקרים בשיטת backpatching. דוגמא:

$B \rightarrow B_1 \ or \ Q \ B_2$

```
{
        B.truelist = merge(B1.truelist, B2.truelist);
        B.falselist = B2.falselist;
        backpatch(B1.falselist, Q.quad);
}
```

$Q \rightarrow \varepsilon$

```
{
        Q.quad = nextquad();
}
```

```
B.truelist = merge(B1.truelist, B2.truelist);
```

שורה זו מניחה, כרגיל, ש- B_1, B_2 חושבו ונכתבו כבר לפלט.

```
backpatch(B1.falselist, Q.quad);
```

השורה הזו עושה את "הקסם" של הדוגמא. אבחנות:

- Q.quad מכיל את הכתובת של תחילת B_2.
- מכאן, אם ערכו של B_1 הוא false, אנו קופצים לתחילת B_2. ההטלאה שמה את הכתובת הנכונה ב-falselist של B_1.

דוגמא – מבנה בקרה פשוט

נגדיר ל-S תכונה נוצרת – nextlist – רשימת ההתחייבויות עבור נקודת היציאה next. צריך לשים לב כי גם בדוגמא זו S יכול "ליפול" לפקודה הבאה.

אנחנו רוצים לכתוב את הכללים הסמנטיים עבור:

$S \rightarrow if\ B\ then\ S_1\ else\ S_2$

נשתמש במרקר Q כמו שהוצג קודם, ובמרקר נוסף G.

```
S → if B then Q1 S1 G else Q2 S2
        {
                backpatch(B.truelist, Q1.quad);
                backpatch(B.falselist, Q2.quad);
                S.nextlist = merge(S1.nextlist, S2.nextlist, G.nextlist);
        }

G → ε
        {
                G.nextlist = makelist(nextquad());
                emit('goto ___');
        }
```

נקודות:

- G מופיע בשביל המקרה ש-S נופל לפקודה הבאה.
- המרקרים G ו-Q הם מאוד שימושיים – נשתמש בהם ברוב הדוגמאות של backpatching.

דוגמא נוספת: מבנה הבקרה while:

```
S → while Q1 B do Q2 S1
        {
                backpatch(S1.nextlist, Q1.quad);
                backpatch(B.truelist, Q2.quad);
                emit('goto' Q1.quad);
                s.nextlist = B.falselist;
        }
```

הפקודה emit במקרה זה מטפלת במקרה בו S נופל למטה.

11.מקורות

11.1. מקורות באנגלית

- http://en.wikipedia.org/wiki/Recursive_descent_parser
- http://en.wikipedia.org/wiki/LR_parser
- http://en.wikipedia.org/wiki/Compiler
- http://en.wikipedia.org/wiki/Lexical_analysis
- http://en.wikipedia.org/wiki/Top-down_parser
- http://en.wikipedia.org/wiki/Bottom-up_parsing
- http://en.wikipedia.org/wiki/Backus%E2%80%93Naur_form
- http://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_Form
- http://en.wikipedia.org/wiki/Talk:LR_parser
- http://en.wikipedia.org/wiki/Simple_LR_parser
- http://en.wikipedia.org/wiki/Talk:Simple_LR_parser
- http://en.wikipedia.org/wiki/Canonical_LR_parser
- http://en.wikipedia.org/wiki/LL_parser
- http://en.wikipedia.org/wiki/Parsing_table
- http://en.wikipedia.org/wiki/Comparison_of_parser_generators
- http://en.wikipedia.org/wiki/Context-free_grammar
- http://en.wikipedia.org/wiki/LALR_parser
- http://en.wikipedia.org/wiki/Compiler_design
- http://en.wikipedia.org/wiki/Code_generation_(compiler)
- http://en.wikipedia.org/wiki/Compiler_optimization
- http://en.wikipedia.org/wiki/Intermediate_code_generation
- http://en.wikipedia.org/wiki/Scope_(programming)
- GOLD Parsing System - http://www.devincook.com/goldparser
- "A Yacc Tutorial", Victor Eijkhout, July 2004, http://mirror.sweon.net/madchat/coding/unix/yacc-tutorial.pdf
- Bison Tutorial, Lan Gao, http://www.cs.ucr.edu/~lgao/teaching/bison.html
- Bison 2.3 Manual - http://www.gnu.org/software/bison/manual/html_mono/bison.html
- LR parsing MS/Powerpoint presentation, Aggelos Kiayias, University of Connecticut
- "Compiling and Translation", Essays by F. D. Lewis, 2000, http://www.cs.uky.edu/~lewis/essays/comp-index.html
- A.V. Aho, M. S. Lam, R. Sethi, and J.D. Ullman –"Compilers –Principles, Techniques, and Tools"

11.2. מקורות בעברית

- "תורת הקומפילציה, תרגולים בטכניון", 2004
- "תורת הקומפילציה, תרגולים בטכניון", 2006
- "תורת הקומפילציה, תרגולים בטכניון", 2009
- "תורת הקומפילציה, הרצאות בטכניון", 2009
- "שיטות הידור (קומפילציה), תרגולים בטכניון", 2004
- "שיטות הידור (קומפילציה), תרגולים בטכניון", 2005, http://www.ee.technion.ac.il/courses/046266/compilers_michael/Spring2005/tutorials/
- "הסבר Flex", אופיר אהרן, 2008
- "תורת הקומפילציה, תרגולים בטכניון", סוכם על ידי שיר בן ישראל