שפות תכנות 234319

פרופ' יוסי גיל

הפקולטה למדעי המחשב

הטכניון - מכון טכנולוגי לישראל

סיכום הרצאות 1 ו 2.

רשמה: **איריס קלקה**[1]

תוכן העניינים

[1] נערך על ידי יוסי גיל, תוך שימוש גם ברשימותיהם של רועי דיבון ומתן סבג.

אדמיניסטרציה ועניינים טכניים

למרצה של הקורס יש מנהג מגונה: הוא מוכן לענות בחפץ לב על שאלות אישיות, ויהיו אלו הקנטרניות ביותר[2], אבל פניות ישירות אל המרצה בנושא החומר של הקורס, או במגוון של נושאים פרוצדרוליים ואדמינסטרטיביים תתקלנה בדרך כלל בהתעלמות, תהיה הפניה ענינית ככל שתהיה. הסיבה היא שפניות בנושאים מסוג זה, מקומם ברשות הרבים, בדרך כלל בקבוצת הפייסבוק של הקורס. פניה פומבית נותנת מידע ועוזרת לסטודנטים אחרים, ומאפשרת גם תגובה של סטודנטים אחרים בקורס.

שעות הקבלה של המרצה הן בתיאום של 24 שעות מראש, באמצעות מערכת הזמנות אלקטרונית.

נהלי הקורס המלאים והמחייבים זמינים לכל דיכפין, ואילו מבארים היטב עניינים כגון דרישות קדם, דרישות צמודות, תרגילי הבית, מבנה המבחן, וכיוצ"ב.

ספרי לימוד וחומר עזר

קורס זה נבנה על יסודה של המהדורה השניה של ספרו של David Watt נושא השם "*Programming Languages Concepts and Paradigms*". אבל, במקומות רבים מאוד נרחיב מעבר לכתוב בספר. ספר הלימוד הזה מתבסס על שלוש שפות תכנות:

- Pascal
- ML
- Ada

הספר מעט מיושן, ולעיתים משתמש במונחים באורח אידיוסינקרטי מעט. אמנם קיימת לו מהדורה שלישית, אך לא נשתמש בה כיוון שהיא זונחת את שפת התכנות ML לטובת שפת התכנות Haskell.

ניתן למצוא את השקפים של המהדורה השלישית באינטרנט, וחומר נוסף:

http://www.dcs.gla.ac.uk/~daw/books/PLDC/

ניתן גם לרכוש את הספר כאן:

http://preview.tinyurl.com/watt2

[2] השאלה: "שותפי העבדקן צבע זקנו בשני, צבע המזכיר לי אנשובי, אשר איני יכול לשאת את ריחו. מה עלי לעשות?" היא לגיטימית!

מוטיבציה

למידה פרוצדרולית בניגוד ללמידה דקלרטיבית

מומחים מבדילים בין למידה פרוצדרולית, ובין למידה דקלרטיבית. בלמידה דקלרטיבית, אנו לומדים מושגים ועובדות, שהבנתם ועיבודם דורשת מאמץ קוגניטיבי לא פשוט: משפטים, הוכחות, תהליכים מורכבים כמו אינטגרציה בחלקים, הגדרות, פרוטוקולים, אלגוריתמים וכו', הם דוגמאות ללמידה דקלרטיבית. לימוד נהיגה או שחיה, הוא לימוד פרוצדורלי: הידע הנדרש לשם הפעלת כלי רכב מנועי אינו רב. אבל, יש צורך להתאמן רבות כדי להגיע בו להישגים.

הלימוד בקורס הזה הוא לימוד פרוצדורלי: אמנם ניתקל בהרבה הגדרות, אבל הבנתן אינה דורשת בדרך כלל מאמץ חשיבתי. מה שנרצה להרכיש לסטודנטים בקורס היא היכולת להפעיל את ההגדרות לשם יעול הלימוד של שפות תכנות חדשות, רכישת אופני מחשבה חדשים, יכולת להבין ולהשתתף בשיח המקצועי בנושא של שפות תכנות, וגם, היכולת לפתח בעצמך שפות חדשות.

לשם כך, מומלץ מאוד מאוד להשתתף בהרצאות: לשאול שאלות, לסכם תוך כדי הרצאה, ולהשתתף בלמידה באורח אקטיבי. ישנם סיכומים שונים ברשת, וישנן אף הקלטות וידאו של מהדורה (ישנה מאוד של הקורס), אבל, אין תחליף לנוכחות, ולביצוע אמיתי של תרגילי הבית. (אגב, הקלטות הוידאו של הקורס התגלו כמלכודת פתאים מסוכנת: ישנו מתאם גבוה בין כשלון בקורס, ובין הסתמכות על ההקלטות שאינן מעודכנות כלל.)

מדוע ללמוד "שפות תכנות"?

קורס זה מטרתו לסייע לסטודנטים ללמוד ולהכיר במהירות שפות תכנות חדשות, להגיע לשליטה עמוקה יותר בשפות התכנות שהם מכירים ובהם הם עובדים, ובעיקר, "להגמיש את המחשבה". כדי להסביר את המושג "להגמיש את המחשבה" נספר שבאחד המחקרים, נבדק הַמִּתְאָם בין התפוקה של מתכנתים, ובין פרמטרים שונים, ובהם השכר, שנות הניסיון, ההשכלה, התפקיד, ועוד. התברר כי החזאי הטוב ביותר של תפוקה היה מספר שפות התכנות השונות אותם הם מכירים.

כמובן, מִתְאָם אינו בהכרח מעיד על סיבתיות. אדרבא, יבוא ד"ר איפכא מסתברא ויאמר שֶׁהַמִּתְאָם הגבוה בין תפוקה ובין הפוליגלוטיות הוא תוצאה של סיבתיות הפוכה: מתכנתים שתפוקתם גבוהה יותר, הם כאלו שעיתותיהם בידיהם, ולכן הם מתפנים יותר ללמוד שפות חדשות. ובכל זאת, אין זה מופרך להניח קשר חשוב בין שתי התופעות. שפות תכנות שונות מייצגות דרכי מחשבה שונות, אופנים שונים לבטא אבסטרקציות, ולהתמודדות עם בעיות תכנותיות. ידע רחב בשפות תכנות מסייע על כן למתכנת לבחון בעיות מזוויות שונות. הבחינה הזו יכולה לאפשר לפתור בעיות שונות באמצעות שפות שונות. וגם אם שפת התכנות מוכתבת ואינה ניתנת לשינוי, דרכי החשיבה השונות יכולות להציע דרכים שונות לפתור את הבעיה גם במסגרת שפת התכנות הנתונה.

התיזה של ספיר-וורף

הרעיון שהשפה מעצבת את המחשבה אינו חדש. ברומן מדע בדיוני מאת סמואל ר. דילייני אשר נכתב בשנת 1966 , אחד הצדדים למלחמה בין גלקטית מפתח נשק סודי בדמות שפה טבעית בשם בבל-17 (כשם הרומן). תכונתה של שפת בבל-17 היא שהיא כופה דרך מחשבה כזו על בני המחנה היריב הלומדים אותה, שהם חשים צורך לשנות את התנהגותם ולבגוד במחנה שלהם.

בבל-17 הוא רומן בידיוני, אבל הוא נסמך על רעיון מדעי חשוב: "התיזה של ספיר-וורף", אשר פותחה בראשית המאה ה-20 על ידי הבלשן אדוארד ספיר ותלמידו בנימין לי וורף. לפי תיזה זו, אופן מחשבתו של אדם מושפע מאוד מהשפה בה הוא דובר, ובמיוחד משפת אמו. גם מבלי להתעמק בתורת הבלשנות, לא קשה להשתכנע בתיזה

זו. כולנו יודעים שבעברית יש יותר ממאה שמות לעיר ירושלים, ושבערבית יש 99 שמות שונים לאלוהים, ויש להניח כי העושר הלשוני הזה משפיע על דרך המחשבה. הנה שתי דוגמאות נוספות:

- אנו יודעים כי בעברית יש שלושה זמנים: עבר, הווה, ועתיד, כאשר צורת ההווה, הקרוייה גם "בינוני" בשפה היא מנוונת. (בערבית, יש שני זמנים בלבד: עבר ועתיד.) לעומת זאת, בשפה הלטינית יש שישה זמנים (tenses), הכוללים למשל זמן לציון מאורע שאירע בעבר ומהווה עבר בעבור מאורע שיקרה בעתיד.
- בלטינית יש שתי מילים נבדלות לדם, זה אשר זורם בגוף נקרא sanguis ואילו זה אשר זב ממנו נקרא cruor.

גם כאן, אין זה מופרך לחשוב כי האבחנה בין זמנים שונים ובין שני סוגי הדם השונים מכתיבה דרכי חשיבה שונות.

אנו נגלה בקורס כי שפות התכנות השונות מבטאות ומכתיבות דרכי חשיבה שונות, לעיתים מתוך רצון מודע של מתכנן השפה להשפיע על דרכי החשיבה של המתכנת. מסתבר למשל שיוקיהירו מצומוטו אשר פיתח את שפת התכנות רובי באמצע שנות התשעים של המאה הקודמת, ראה לנגד עיניו את התיזה של ספיר-וורף.

לימוד שפות תכנות בקורס

ישנו קושי מובנה לבנות וללמד קורס בשפות תכנות: קל לדבר על "גמישות מחשבתית" כמשאת נפש, אבל לא ברור כיצד ניתן להרכיש אותה לסטודנטים אשר מתקשים בהגמשת זמנם כך שיכיל את מערכת הלימודים, העבודה והפנאי גם יחד.

לדעת אחדים, קורס בשפות תכנות הוא קורס שבו הסטודנטים ילמדו שתיים-שלוש (ולעיתים אחת) שפות תכנות מתקדמות, מעניינות, ובכך ירגילו אותם בדרכי מחשבה שונות. אחרים ילמדו את הקורס כקורס תיאורטי, בדומה לקורס ב"אוטומטים ושפות פורמליות", וישנה גם גישה המציעה למנות את המושגים השונים ולפרט את דרך מימושים בשפות התכנות השונות.

הגישה אשר בה ינקוט קורס זה היא גישה משולבת. ראשית, נלמד כמה וכמה שפות תכנות, ובראש ובראשונה את השפות הבאות:

- **ML** - כדי להדגים שאפשר לכתוב תכניות גם מבלי להשתמש במשתנים, ואת צורת החשיבה הפונקציונלית.
- **Prolog** - כדי להדגים שאפשר לכתוב תכניות גם מבלי להשתמש בפקודות.
- **Pascal** - כדי להעשיר את הרקע ההיסטורי, כדי להבין את השפעותיה על שפות התכנות של ימינו, וכדי להנגיד אותה עם שפת התכנות המוכרת יותר, C.

שנית, נציב בקורס גישה שיטתית יותר לניתוח שפות תכנות, ובפרט קריטריונים שונים לניתוחן. נדגים את הגישה הזו על שפות תכנות שונות, ונתרגל אותה לשם לימוד שפות תכנות חדשות. במופע זה של הקורס, הסטודנטים ילמדו "כהרף עין" את השפות AWK, Dart, Go ועוד. כמובן, בתחילה יקח זמן לסטודנטים לרכוש את המיומנות ללמוד שפות חדשות "כהרף עין", אבל יש לקוות כי היכולת הזו תיתפתח במהלך הקורס.

פרדיגמות

מי אשר למד שתיים שלוש שפות תכנות "רגילות" כגון C ו Bash, עלול לחשוב כי לימוד של שפת תכנות חדשה הוא פשוט: יש ללמוד את הדקדוק, ובפרט, כיצד מגדירים משתנים, כיצד כותבים פקודות תנאי, פקודות לולאה, וכיוצ"ב.

בקורס זה נגלה כי הדברים מורכבים מעט יותר: שפות תכנות נחלקות לְפָרָדִיגְמוֹת, כאשר פָּרָדִיגְמָה היא דרך מחשבה וסגנון תכנות, אשר שפת התכנות מכתיבה למתכנת. המושגים שתוארו מעלה, מופיעים רק בפרדיגמה אחת.

הפרדיגמות העיקריות

ארבעת הפרדיגמות העיקריות הן:

1. **הפרדיגמה הצִוּוּיית (Imperative)** הנקראת לפעמים גם הפרדיגמה הפרוצדורלית, ואשר השפות C, Bash ופסקל הן דוגמאות שלה.
   - בפרדיגמה זו נמצא משתנים, ביטויים, פקודות, כולל פקודות תנאי ופקודות לולאה, פונקציות ופרוצדורות.
   - תוכנית הכתובה בשפה אימפרטיבית יש בכל רגע **מצב** (State) המוגדר על ידי תוכן הזיכרון (אוסף המשתנים ה"חיים") ונקודת הריצה הנוכחית.
   - תיכנות בפרדיגמה זו נעשה באמצעות ציווי: כתיבת **פקודות** אשר מצוות על התכנית לשנות את מצבה.
   - פקודות בשפה אימפרטיבית יכולות להיות מאוגדות ב**פרוצדורות**.
   - אחת הסיבות לכך שהפרדיגמה האימפרטיבית היא הנפוצה ביותר, היא ששפות המכונה הן אימפרטיביות. תרגום תכנית בשפה אימפרטיבית לשפת מכונה יכול להיות על כן יעיל במיוחד.
2. **הפרדיגמה הפונקציונלית** אשר השפות LISP, Haskell, ML ו Scheme הן דוגמאות שלה.
   - בפרדיגמה זו אין משתנים כלל ואין פקודות.
   - את מקום פקודת התנאי מחליף ביטוי מותנה, את הלולאות מחליפה רקורסיה. בפרק התירגול של הקורס שבו נלמד את שפת ML נגלה באורח מפתיע כי ניתן לתכנת ללא שימוש במשתנים כלל.
   - כתוצאה מהעדר המשתנים והפקודות, לתכנית אין "מצב" בזמן ריצה, פרט לנקודות החישוב הנוכחית.
   - שם הפרדיגמה נובע מכך שפונקציות מהוות את הדרך העיקרית לתיאור החישוב. תכנית בפרדיגמה זו מרבה לחשב ביטויים באמצעות קריאה לפונקציות, להעביר פונקציות כפרמטר לפונקציות אחרות, וכתיבת פונקציות אשר מייצרות ומחזירות פונקציות אחרות.
   - נשים לב לכך שבהעדר פקודות, כל שפונקציה יכולה לעשות הוא להחזיר את ערכו של ביטוי, אשר יכול להיות מחושב באמצעות קריאה לפונקציות אחרות, או באמצעות האופרטורים היסודיים.
   - שפות פונקציונליות הן לרוב אלגנטיות במובן זה שיש מספר קטן של מושגים אותם יש להבין כדי לתכנת בשפה, וגם במובן זה שהן מאפשרות ביטוי תמציתי של דרך החישוב. באופן טיפוסי תכנית לפתרון בעיה מסויימת בשפה פונקציונלית תהיה קצרה בכמה מונים מתכנית דומה בשפה ציווית.
3. **פרדיגמה מונחית העצמים** אשר השפות C++ ו Java הן דוגמאות שלה, אף שאינן שפות תכנות מונחות עצמים טהורות. דוגמאות לשפות שהן מונחות עצמים טהורות יותר הן Eiffel ו Smallatalk. פרדיגמה

זו תילמד בקורס ההמשך 236703 - תכנות מונחה עצמים.

- כדי להדגים את צורת החשיבה השונה בפרדיגמה זו, נסתכל בקטע קוד המקביל לפקודת תנאי בשפת Smalltalk:

```
a < b
  ifTrue: [^'a is less than b']
  ifFalse: [^'a is greater than or equal to b']
```

- בפרדיגמה מונחית העצמים, כל החישוב כולו הוא העברת הודעות לעצמים, אשר מגיבים להם בחישוב קצר בהתאם לטיבו של העצם המקבל. כך, `a < b` שנראה כביכול כביטוי, אינו ביטוי כלל וכלל, אלא שליחת ההודעה `<` עם הפרמטר `b` לעצם `a`. בתגובה להודעה זו העצם `a` יחזיר את העצם `true` או העצם `false`.
- לתוצאה המוחזרת תשלח ההודעה

`ifTrue: … ifFalse: …`

אשר לה שני ארגומנטים, אשר הם בלוקים המוכנים לחישוב.

- `[^'a is less than b']`
- `[^'a is greater than or equal to b']`

כעת, העצם `true` יחשב את הארגומנט הראשון, ויתעלם מהשני, ואילו העצם `false` יתעלם מהארגומנט הראשון ויחשב את השני.

4. **הפרדיגמה הלוגית** – אשר שפת התכנות Prolog מהווה דוגמא שלה. אף בפרדיגמה זו אין פקודות, משתנים, תנאים, ולולאות, אבל בניגוד לפרדיגמה הפונקציונלית, אין בפרדיגמה הזו הגדרת דרך חישוב. לעומת זאת, על המתכנת להגדיר תוך שימוש בלוגיקה את מטרת החישוב, ו"מָנוֹעַ" פנימי של שפת התכנות יפעל לשם השגת מטרה זו.

החלוקה לפרדיגמות אינה תמיד חדה וברורה. בנוסף לפרדיגמות העיקריות, יש פרדיגמות נוספות אשר ההבדלים ביניהן הם לעיתים שנויים במחלוקת. עם אלו ניתן למנות את הפרדיגמה המקבילית (אשר נלמדת בקורסים אחרים בפקולטה), הפרדיגמה מונחית האספקטים ועוד רבות רבות אחרות. בנוסף, ישנן גם שפות המשתייכות למספר פרדיגמות.

הפרדיגמה הלוגית

הפרדיגמה הלוגית מתאפיינת בכך שאין בה לא פקודות הצבה, לא פקודות תנאי ולא פקודות לולאה. בפרדיגמה הזו אין גם פונקציות ולא פרוצדורות, ואין (כמעט) מילים שמורות. בשפות בפרדיגמה הלוגית יש אמנם "משתנים", אך אלו שונים לחלוטין בטיבם מהמשתנים המוכרים לנו בשפות אימפרטיביות.

במקום כל אלו, תכנית בפרולוג מורכבת משלושה חלקים:

1. עובדות יסוד.
2. כללי היסק.
3. מטרת החישוב.

כל אחד משלושת אלו מנוסח כטענה לוגית. יתירה מכך, עובדות הן למעשה כללי היסק, אשר בסיס ההיסק שלהן ריק. ניתן לכן לקרוא לקבוצת עובדות היסוד וכללי ההיסק אקסיומות, הנכתבות כפרדיקטים לוגיים. המתכנת הכותב בפרולוג, מגדיר אוסף של אקסיומות, ואחר כך, החישוב בפועל מתבצע על ידי הצבת "מטרה", שהיא פרדיקט בלוגיקה.

בפרולוג יש מנוע פנימי המנסה להוכיח את פרדיקט המטרה מתוך האקסיומות. ההוכחה יכולה להיות של הפרדיקט הכללי, או של מקרים פרטיים של המשפט המתקבלים באמצעות קביעת ערך של משתנים עבורו.

טרם תחילת הביצוע בפרולוג, טוען מנוע החישוב מספר רב של פרדיקטים. הנה הדגמה של הרצה בפועל של פרולוג.

```
% prolog # invoke interpreter from command line
Welcome to SWI-Prolog (Multi-threaded, 瞎说
For help, use ?- help(Topic). or ?-
apropos(Word).

?- plus(2,2,4).
true.

?- plus(1,1,3).
false.

?- plus(3,X,17).
X = 14.
```

בדוגמא לעיל:

1. הקלט מהמשתמש צבוע באדום, ואילו הפלט אותו מוציא המחשב, צבוע בכחול.
2. במקום שבו ראינו לקצר ולא לפרט את הפלט המלא' החלפנו אותו בסימנים 瞎说 אשר את פענוח משמעותם בשפה הסינית נותיר לקורא החפשן.
3. הסימן '%' הוא הזרז (prompt) של המפרש bash.
4. הסימן '#' בשפת bash מתחיל הערה אשר נמשכת עד לסוף השורה
5. הסדרית

```
-?
```

היא הזרז של מנוע הפרולוג.

אנו רואים בדוגמא ניצול של המנוע של פרולוג כדי לבדוק ש 4=2+2, כדי לבדוק ש 2≠1+1 וכדי למצוא הצבה ל"משתנה" X אשר תקיים את הפרדיקט

```
plus(3,X,17)
```

ניתן לנצל את שפת פרולוג כדי לבצע חישובים מורכבים מאלו. ניזכר למשל בפתיחה של פרקי שיר השירים לסופר שלום עליכם בתרגומו של י.ד. ברקוביץ':

לִילִי – שֵׁם יַלְדָּה אֲשֶׁר מוֹצָאוֹ מֵאֶסְתֵּר-לֵאָה: לֵאָה – לִילִי. גְּדוֹלָה הִיא מִמֶּנִּי בְּשָׁנָה אַחַת, וְאֶפְשָׁר בִּשְׁתֵּי שָׁנִים. וּמִסְפָּר שְׁנוֹתֵינוּ יַחַד לֹא יַגִּיעַ גַּם עַד עֶשְׂרִים. שַׁעֲרוּ אֵפוֹא בֶּן כַּמָּה שָׁנִים אֲנִי, וּבַת כַּמָּה שָׁנִים לִילִי?

כדי לפתור את החידה אשר מציג כאן שלום עליכם לקורא, נסמן את גילו של הדובר במשתנה I ואת גילה של לילי במשתנה L. עוד נסמן את ההפרש בין גילו של הדובר ובין גילה של "אהובתו" במשתנה D, ואת סכום גיליהם במשתנה S.

כעת, כל שנותר הוא לפרק את הנתונים בחידה למשפטים לוגיים:

- `plus(4, 13, I)` הדובר הוא ילד
- `plus(4, 13, I)` גם לילי היא ילדה
- `plus(I, D, L)` ההפרש בין גילי הילדים
- `between(1, 2, D)` הפרש זה הוא בין שנה או שנתיים
- `plus(I, L, S)` סכום גילי הילדים
- `between(0, 19, S)` אינו עולה עד לעשרים

וכך תיראה ההרצה:

```
?- between(4, 12, I),
|  between(4, 12, L),
|  plus(I, D, L),
|  between(1, 2, D),
|  plus(I, L, S),
|  between(0, 19, S).

I = 4,
L = 5,
D = 1,
S = 9 .
```

במקרה זה, אנו רואים שהמנוע של פרולוג מוצא פתרון אחד למערכת האילוצים של שלום עליכם. ניתן גם להשתמש בפרולוג כדי למצוא את כל הפתרונות של מערכת האילוצים, ועל כך נלמד בהמשך הקורס[3]. באופן דומה, קל להשתמש בפרולוג כדי למצוא פתרון לחידות סודוקו, או גם לחידת הזברה המפורסמת.

התיזה של צ'רץ' וטיורינג

לאחר שנוכחנו כי פרולוג מייצגת צורת חשיבה השונה בתכלית מזו המוכרת לנו בפרדיגמה הציווית, טבעי לשאול האם כוח הביטוישל פרולוג זהה לזה של שפת C למשל. כלומר, האם אפשר לכתוב בפרולוג כל תכנית שאפשר לכתוב בשפת C. מתברר, כי לא זו בלבד שהתשובה לשאלה זו חיובית, מסתבר כי (כמעט) כל שפות התכנות שקולות זו לזו.

עבור שפת תכנות $\mathcal{L}$ נסמן ב $\boldsymbol{S}_{\mathcal{L}}$ את קבוצת הבעיות אותן אפשר באמצעות תכניות הכתובות בשפה $\mathcal{L}$. כך למשל הקבוצה $\boldsymbol{S}_{\text{prolog}}$ היא קבוצת הבעיות אותן אפשר לפתור בעזרת שפת התכנות פרולוג, ואילו $\boldsymbol{S}_{\text{C}}$ היא קבוצת הבעיות אותן אפשר לפתור בעזרת שפת C.

אם מתקיים השיוויון $\boldsymbol{S}_{\mathcal{L}}=\boldsymbol{S}_{\mathcal{L}'}$ אזי נאמר שהשפות $\mathcal{L}$ ו $\mathcal{L}'$ הן **שקולות חישובית**.

נסתכל לרגע על הקבוצה $\boldsymbol{S}_{\text{batch}}$ הלא היא קבוצת הבעיות אותן אפשר לפתור בעזרת שפת האצוות של מערכת ההפעלה DOS, אשר בה הפקודות הבאות בלבד:

1. `IF` *file* `EXISTs`
2. `IF` *string* `equals string` (and negation)
3. `IF ERRORLEVEL` (and negation)
4. `FOR` *variable* `in` *set* `DO`
5. `GOTO`
6. `CALL`
7. `:`*label*

ניכר שהשפה הזו מנוונת במיוחד, ואולי יביא הדבר להשערה שהקבוצה $\boldsymbol{S}_{\text{batch}}$ היא קטנה למדי. באורח מפתיע משהו מתברר כי

$$\boldsymbol{S}_{\text{prolog}} = \boldsymbol{S}_{\text{C}} = \boldsymbol{S}_{\text{batch}}$$

---

[3] למען החיות נפשם של קצרי הרוח, כדי לקבל את **כל** הפתרונות לחידתו של שלום עליכם, יש להוסיף עוד שני איברים למטרה:

```
between(4, 12, I), between(4, 12, L),
 plus(I, D, L), between(1, 2, D), plus(I, L, S), between(0, 19, S),
 printf("I am %d years old, while Lily is %d\n", [I,L]), fail.
```

בהזנת תכנית זו לפרולוג נקבל את הפלט הבא:

```
I am 4 years old, while Lily is 5
I am 4 years old, while Lily is 6
I am 5 years old, while Lily is 6
I am 5 years old, while Lily is 7
I am 6 years old, while Lily is 7
I am 6 years old, while Lily is 8
I am 7 years old, while Lily is 8
I am 7 years old, while Lily is 9
I am 8 years old, while Lily is 9
I am 8 years old, while Lily is 10
I am 9 years old, while Lily is 10
false.
```

כלומר שלושת השפות הללו שקולות. הכוח החישובי של פרולוג זהה לזה של שפת C, וגם לזה של השפה בת שבעת הפקודות שהצגנו מעלה. לא זו בלבד, מתברר כי השיוויון הזה מתקיים בעבור כמעט כל שתי שפות תכנות.

תופעה זו התגלתה על ידי שני מדענים, Alonzo Church ו Alan Turing אשר ייגעו מוחם בתהיה אם השאלה "עם אלו בעיות יכול מחשב להתמודד?" היא שאלה מטאפיזית, פילוסופית, או שאלה מתימטית.

טיורינג הגדיר מודל חישוב הקרוי "מכונת טיורינג", ואילו צ'רץ' הגדיר שיטה מתימטית להגדרת חישוב הקרוייה "תחשיב הלמבדא". התברר כי שני הפורמליזמים הללו זהים, והתזה של טיורינג וצ'רץ' גורסת (והניסוח הלא מתימטי הוא במתכוון) בדלקמן: "כל **המודלים החישוביים זהים**".

בפרט, מתברר כי שפת תכנות אשר מכילה תנאים לוגיים ורקורסיה (או לחילופין, לולאות), שקולה מבחינה חישובית למכונת טיורינג. הדרישות הפעוטות אלו מהשפה, מביאות לתוצאה שרק חלק אפסי משפות התכנות אינות שקולות חשיבות למכונת טיורינג , ועל כן רוב שפות התכנות שקולות חישובית זו לזו.

שפה אשר כוח החישוב שלה שקול לזה של מכונת טיורינג, נקראת שפה **אוניברסלית** או שפה שהיא **טיורינג שלמה** (**Turing Complete**).

הקורס "תורת החישוביות" מעמיק מאוד בשאלה של מודלים חישוביים שונים, היחסים ביניהם, האבחנה בין בעיות הנתונות לחישוב ובין בעיות שאינן ניתנות לחישוב. (כך למשל, בעית שמונה-המלכות היא בעיה הניתנת לחישוב, בעוד שבעיית העצירה[4] אינה ניתנת לחישוב.)

כדאי לשים לב לכך ששקילות חישובית מתייחסת לשאלה של פתירות: האם קיימת תכנית הפותרת בעיה מסויימת, אך היא אינה מתייחסת לשאלה של יעילות. יתכן שפתרון אחד יהיה יעיל יותר מפתרונות אחרים.

הקורסים במבני נתונים ובאלגוריתמים דנים בשאלה של **יעילות החישוב** מבחינת משאבי המכונה, ובעיקר מספר פעולות החישוב וגודל הזיכרון הנדרש לאלו. לעומת שאלת היתכנות החישוב, ושאלת יעילות החישוב בקורס בשפות תכנות דן מתייחסת ל**יעילות ותמציתיות דרכי הביטוי של הפתרון** והתאמתן של דרכים אלו לדרכי המחשבה של המתכנת.

צבת בצבת עשוייה

בהינתן שפת תכנות אוניברסלית ℒ יש טעם לשאול באיזו שפה כתוב המהדר או המפרש של ℒ. אך, כיוון ש ℒ היא אוניברסלית, הרי אם אפשר לכתוב את המהדר (לחילופין, המפרש) של ℒ בשפה 'ℒ הרי גם ניתן לכתוב את המהדר(לחילופין, המפרש) בשפה ℒ עצמה.

מתברר שמקובל מאוד לכתוב את המהדר של השפה בשפה עצמה. כך למשל, המהדר של שפת C כתוב בשפת C. המהדר של שפת Java כתוב בשפת Java, וכו'. כמובן שהדבר מעורר קושי: כי אם המהדר עבור שפה מסויימת כתוב באותה שפה, הרי כיצד הודר המהדר? התשובה הפשוטה היא שהמהדר הידר את עצמו, וכך הם בדרך כלל פני הדברים. אלא, שהדבר מוביל לרקורסיה אינסופית.

חז"ל הבחינו בבעיה דומה של רקורסיה אינסופית מעין זו בסוגיא התלמודית הידועה בשם "צבת בצבת עשוייה". בגמרא, במסכת פסחים דף נ"ד עמוד א' נאמר:

צבתא בצבתא מתעבדא וצבתא קמייתא

מאן עבד הא לאי בריה בידי שמים

ובתרגום לעברית: "הצבת אינה נעשית אלא בצבת אחרת. וראשונה מי עשאה? על כרחך מאליה נעשית בידי שמים.". כלומר, צבת שהיא מכשיר לאחיזת מטילי ברזל לשם ליבונם באש ועיבודם, עשוייה אף היא ברזל, ואף

[4] הקלט לבעית העצירה הוא תכנית מחשב יחד עם הקלט שלה. פותר הבעיה נדרש להחליט האם הרצת התכנית על הקלט תמשך זמן אינסופי.

היא מיוצרת בצבת אחרת שקדמה לה. כיצד אם כן נוצרה הצבת הראשונה?

הפתרון המוצע על ידי הגמרא הוא שהצבת הראשונה נבראה בערב שבת הראשון, בזמן "בין השמשות", שעה שאלוהים סיים לברוא את כל הדברים האחרים, והתכונן לשבות ממלאכתו לקראת ירידת השבת.

פתרון ניסי שכזה אינו בא בחשבון עבור שפות תכנות. מסכת פסחים מציגה גם דרך אחרת שבה יוצרה הצבת הראשונה (על ידי דפוס נחושת). לעומת זאת, בשפות תכנות, ניתן לבצע תהליך של bootstratping שבאמצעותו ניתן לפתח מהדר המסוגל להדר את עצמו.[5]

מדד לאלגנטיות של שפה

אבן בוחן מרתקת לאלגנטיות של שפת תכנות היא אורך המהדר (או המפרש) של השפה, כאשר הוא כתוב בשפה עצמה. שהרי ככל שהשפה מורכבת ומתוחכמת יותר, מחד קל יותר לכתוב את המהדר, אך מאידך המהדר לעסוק בכל המורכבות והעושר הזה.

לחילופין, שפה שהיא פשוטה ביחס (כמו שפת ה batch של DOS) שהוזכרה מעלה, היא קלה אולי להידור, אבל הפשטות של השפה מהווה אבן נגף בבואנו להשתמש בה כדי לכתוב מהדר.

הנה מספר אורכים אופייני של מהדר לשפה הכתוב בשפה עצמה:

1. מהדר לשפת C הכתוב בשפה עצמה, דורש כמה עשרות אלפי שורות. המהדר gcc מפרס על פני כשבעה מיליון שורות קוד.
2. המהדר הראשון לשפת פסקל, אשר נכתב בשפת פסקל דרש כשבעת אלפים ומאתיים שורות.
3. מפרש לשפת ליספ הכתוב בליספ, הידוע גם **כפונקציה האוניברסלית eval** דורש כמאה שורות, או מעט פחות מכך.
4. לעומת זאת, מפרש בסיסי לשפת פרולוג הכתוב בפרולוג יכול להכתב בשורה אחת בלבד, ומפרש מתוחכם, המאפשר למשל מעקב אחרי החישוב, לא ידרוש בדרך כלל יותר מעשר שורות.

[5] לא נתאר כאן את התהליך, אבל בסופו של יום, הוא דומה מאוד למה שהיה עושה נַפָּח עני שברשותו ברזל, אך לא די ממון לרכישת צבת. נפח כזה היה משתמש בכבשנו באופן איטרטיבי, כאשר בכל פעם הוא היה משתמש בגוש הברזל הדומה ביותר לצבת שיש ברשותו, כדי ליצר קירוב טוב יותר לצבת.

כיצד נוצרות שפות תכנות?

לידתן של שפות התכנות הראשונות

המכונות האוטומטיות הראשונות לא נזקקו לשפות תכנות. ישנו מוטיב משותף למכונות אלו:

- מֵהַנוֹלִים לְטֶוַיִת טֶכְסְטִיל של יוסף מארי ג'קארד (שנת1801), ועד מכונת הטאבולציה והמיון של הרמן הולרית' (הפעלה ראשונה בעבור מפקד האוכלוסין של 1890 בארה"ב),
- מתיבת הנגינה (1811) ועד הפיאנולה (הדגמה ראשונה בשנת 1876)
- ממכונת ההפרשים של צ'רלס בבג' (שתוכננה ונבנתה חלקית בשנים 1833-1842 אך נזנחה אחרי כן) ועד המנתח הדיפרנציאלי (1934) אשר נבנה מִמְּכָנוֹ לצורך פתרון משוואת דיפרנציאליות.

המוטיב המשותף למכונות אלו היה שהתכנות לא היה חלק מהמכונה, כלומר, המכונה לא הייתה יכולה לטפל בתכנית שלה כדרך שהיא מטפלת בנתונים אותם היא נוצרה לעבד. במכונות אלו היה אולי ניתן לממש אלגוריתמים רבים, אך לא היה ניתן לממש אלגוריתם לעיבוד תכניות. תכנות של המחשבים העתיקים הללו נעשה על כן באמצעות מניפולציה ישירה של המכונה על ידי המתכנת.

הפרדה זו היא טבעית מאוד בחישוב אנאלוגי (כמו שהיה במנתח הדיפרנציאלי למשל), שכן במחשב האנאלוגי, כל נתון מספרי אינו מקודד (בשיטה העשרונית, הבינארית, או בשיטה קידוד אחרת), אלא גודלו של הנתון מיוצג באמצעות גודל המצוי במחשב, כגון עצמת מתח, זווית הסיבוב של גלגל שיניים וכיוצ"ב, וכמובן מאליו שיטה זו אינה מתאימה לקידוד פקודות.

בכל זאת, גם בחישוב הדיגיטלי בתחילתו, הנתונים היו שמורים באופן נפרד מהתכנית. כך למשל, התכנית הייתה מוזנת באמצעות סרט מנוקב אשר לא היה נוצר באמצעות המחשב עצמו, אלא באמצעות התקנים אחרים. מודל התכנית המאוחסנת של המדען ג'ון פון ניומן הציע ארכיטקטורה בה התכנית עצמה נשמרת בזיכרון המחשב, וניתנת לטיפול על ידי המחשב כדרך שהמחשב מטפל בנתונים אחרים.

תכנות בשפת אסמבלי[6] מתאפשר בזכות מודל התכנית המאוחסנת. בשפה זו, ניתן לכתוב את פקודות המכונה באמצעות מילים המובנות לבני אדם במקום צירופים של 0-ים ו1-ים, או קידוד דיגטלי. האסמבלר היא תכנית מחשב המבצעת תרגום של האסמבלי לקידוד המוכר למכונה. תרגום זה הוא אחד-לאחד: כל פקודת אסמבלי מתרגמת לפקודת מכונה אחת (ישנם חריגים מעטים לכלל זה, אך הם אינם רלוונטיים לענייננו.) באחת המכונות הראשונות, ה ENIAC שפותח בשנת 1946, התרגום של שפת האסמבלי לפקודות מכונה נעשה באורח ידני, על ידי שש מתכנתות, שעבודתן זו הביאה להן תהילה.

גם בימינו, השפה הטבעית של מחשב היא שפת מכונה. אך מתכנתים צריכים שפה גבוהה יותר משום שכתיבת תכניות בשפת מכונה צורכת זמן רב, הן בגלל העובדה שתכנות בשפת מכונה מועד לבאגים, והן בגלל העובדה שקשה מאוד לבטא רעיונות מורכבים בשפת מכונה. גם התחזוקה של תכניות בשפת מכונה היא קשה: קשה לקרוא אותן (על אף הכלל הידוע שיש לכתוב שורת הערה על כל שורה בשפת מכונה), קשה לדבג אותן, וכמעט בלתי אפשרי להעבירן למכונות חדשות יותר.

שפת התכנות העילית הראשונה הייתה FORTRAN אשר פותחה בשנת 1957 על ידי ג'ון באקוס שהודה כי פיתוח השפה נבע מ"עצלנות" מתלתכנת באסמבלי חישובים בליסטיים ובהם נוסחאות מורכבות. השם FORTRAN הוא קיצור לביטוי **For**mula **Tran**slation, ואכן, למצהלותיהם של העצלנים, בשפה זו, ממש כמו בשפות מודרניות היה ניתן לכתוב ביטוי מתימטי מסובך, אשר היה מתורגם על ידי המהדר לשפת מכונה.

זמן קצר לאחר מכן, בשנת 1958, פותחה שפת התכנות LISP. גם השם LISP נוצר מקיצור של ביטוי **List**

[6] בשפת הדיבור מכנים לעיתים הן את השפה, והן את המעבד שלה בשם אסמבלר.

Processing. ההתפתחות של ליספ היא מעניינת במיוחד: המדען ג'ון מקארת'י פיתח את השפה ככתיב פורמלי הדומה לתחשיב הלמבדא (שהוזכר למעלה). להפתעתו של מקארת'י, עובד יבמ בשם סטיב ראסל גילה שניתן לתרגם את הפונקציה האוניברסלית eval של ליספ לשפת מכונה, וכך כתב את המפרש הראשון של השפה.[7]

שיקולים בתכנון שפות תכנות

שפות התכנות הראשונות נוצרו כדרך נוחה יותר ל**תקשורת בין אדם למכונה**. אולם, מתברר כי נודעת להן חשיבות נוספת: שפת התכנות משמשת גם ל**תקשורת בין אדם לאדם**, לצורך העברה של רעיונות אלגוריתמים מורכבים. יתירה מכך, שפת התכנות משמשת גם ל**תקשורת בין אדם לעצמו**, בסייעה לו לנסח רעיונות אלו.

תכנון של שפת תכנות חדשה הוא לעולם פשרה בין המטרות השונות הללו:

- תקשורת בין אדם למכונה עוסקת בשאלה של היעילות של השפה, כלומר, באיזו מידה ניתן לתרגם את השפה לשפת מכונה המתבצעת ביעילות.
- תקשורת בין אדם לאדם עוסקת בשאלה של הקריאות של השפה, כלומר באיזו מידה ניתן לאדם אחר, לבד מן הכותב, להבין את הרעיונות המבוטאים בשפה.
- תקשורת בין אדם לבין עצמו עוסקת בנוחות הביטוי בשפה, כלומר באיזו מידה נוח לכותב להשתמש בשפה כדי לבטא רעיונות מורכבים.

הנה רשימה של מספר אבני בוחן, לעיתים סותרות, לתכנון של שפת תכנות.

1. **אוניברסליות**: מרבית שפות התכנות, אך לא כולן, הן אוניברסליות במובן זה שהן שקולות חישובית למכונת טיורינג. שפות מסויימות, כגון Datalog הן אינן אוניברסליות, ובכוונת מכוון.
2. **טבעיות**: זו המידה שבה שיטת הסימונים של שפת התכנות דומה לזה של שפות אחרות, או של כתיב אחר המוכר לנו. כך למשל, שפת COBOL (השפה שפותחה מיד אחרי פורטרן וליספ), ניסתה להשתמש בדקדוק שיגרום לפקודות השפה להיות דומות למשפטים בשפה האנגלית, לעומה בעוד ששפות אחרות מנסות לחקות את הכתיב המתימטי.
3. **יעילות**: אבן בוחן זו עוסקת במידה שבה אפשר לתרגם תכניות בשפה למימוש יעיל בשפת מכונה.
4. **פשטות, אחידות ועקביות:** על פי אבן בוחן זו, בשפת התכנות יהיו מספר קונצפטים (מושגים, או מבנים לשוניים) קטן, ותהיה רק דרך אחת לבטא כל רעיון תכנותי. לא זו בלבד, הקונצפטים יהיו אחידים ועקביים. כך למשל, העובדה שבפסקל יש גם פונקציות וגם פרוצדורות מהווה ריבוי קונצפטים. האיחוד שלהם לכדי פונקציות בשפת C מהווה על כן פישוט. באופן דומה העובדה שבשפת C ישנם שלושה סוגי לולאות (for, while ו do...while) מהווה ריבוי קונצפטים, המאפשר מספר דרכים לביטוי האיטרציה. בשפות מסויימות יש רק מבנה אחד לביצוע לולאות.
5. **מנגנוני הפשטה (Abstraction):** שפת תכנות נדרשת לתת כלים המאפשרים לבטא בנוחות רעיונות תכנותיים מורכבים, ובכללם אלגוריתמים ומבני נתונים.
6. **מודולריות**: שפה שכדי להרחיב קטע קוד שלה צריך לכתוב אותו מחדש איננה נוחה לשימוש.
7. **בטיחות**: השפה צריכה לאפשר זיהוי שגיאות לפני שהן מתרחשות –אידיאלי בזמן קומפילציה, אך לפחות בזמן ריצה.

כלל התפיסה המגובשת

בדרך כלל, שפות תכנות אינן מתפתחות כניסיון הנדסי לאזן בין הדרישות השונות, אלא מתוך "חזון", כלומר תפיסה מגובשת לגבי הצורך והמטרה של שפת התכנות. איזון הדרישות הללו נעשה לאורה של תפיסה זו. לעיתים

[7] הקורא הרוצה להרחיב אופקיו יתכבד וילמד מעט על העת בה פותחה שפת התכנות קובול, וכיצד היא שונה משתי השפות הקדמוניות האחרות.

החזון הזה נכתב בצורה מפורשת על ידי המתכנן, ולפעמים, אנו יכולים להסיק את החזון הזה מתוך בדיקה של תופעות, או עקרונות שמופיעים בשפה.

בשפת C למשל, נמצא את העקרונות הבאים:

1. **תמציתיות**: אין מילים שמורות המציינות פונקציות, מסגור בלוקים נעשה באמצעות זוג של סוגריים מסולסלים (curly brackets), כלומר { }, אין כמעט מילים שמורות לציון אופרטורים (כך, mod מצויין באמצעות סימן האחוז, כלומר %), ישנם הרבה קיצורי דרך הקרובים לפקודות מכונה (למשל האופרטורים ++ ו- =-), שימוש בקיצורי מילים (למשל **int** במקום **integer**) ועוד.
2. **קירבה למכונה, מבלי הצמדות למכונה מסויימת:** כאן נמצא למשל את האפשרות לסמן משתנה כ **register** מתוך המלצה למהדר לשמור אותו ברגיסטר, את הטיפוסים **short, long** ו **int** כמו גם **float** ו **double** והכללים למיפויים למכונה, אריתמטיקה של מצביעים, אופרטורים הפועלים על היצוג של מספרים כביטים, ועוד.
3. **מודולריות הכלים:** עיבוד של תכנית בשפת C נעשה על ידי שלושה כלים שונים, שהצימוד ביניהם נמוך, כלומר, אף אחד מהם אינו מכיר ממש את האחר. הקדם מעבד (pre processor), המהדר (compiler), והעוקד (linker). הקדם מעבד אינו מכיר את שפת C ויכול לעבד קבצי טכסט בכל שפה שהיא, המהדר אינו מכיר את קדם המעבד, ואינו מודע לפעולתו, ואילו העוקד יכול לשמש, ומשמש בפועל, ככלי לעקידה של שפות אחרות.
4. **העדר בטיחות:** ככלל, שפת C מניחה שהמתכנת הוא "ילד גדול", ואינה מגינה עליו בפני שגיאות. כך ניתן בשפת C להעביר פרמטרים לפונקציה אשר אינם תואמים כלל את הגדרת הפונקציה, לחרוג מגבולות של מערכים, להמיר מספרים שלמים למצביעים ולהיפך, ועוד.
5. **העדר הסתרה:** שפת C נמנעת מלהסתיר פעולות שהן יקרות למימוש, במובן זה שהיא אינה מאפשרת להשתמש במבנים של השפה באופן כזה שהמימוש שלהם ידרוש מספר פעולות רב. מסיבה זו, לא ניתן להעביר בשפת C מערכים ורשומות כפרמטר by value, לא ניתן להציב בצעד אחד מערך או רשומה לתוך משתנה מאותו סוג, ולא ניתן לאתחל מערכים המוגדרים כמשתנים אוטומטיים בפונקציות.

זיהוי העקרונות הללו מסייע לנו לזכור את הדקדוק והמשמעות של השפה, ומאפשר לעיתים קרובות ניחוש מושכל של פרקים בתכנון השפה שאותם איננו מכירים. יתירה מכך, זיהוי העקרונות מאפשר לנו לזהות את התפיסה שעמבדה בבסיס תכנון השפה. במקרה של שפת C, החזון של המתכננים היה לבנות שפה פשוטה, הקרובה לשפת מכונה אך יחד עם זאת יבילה (portable) לשם תכנות מערכת ההפעלה UNIX (ראוי לזכור שטרם פיתוח שפת C , פיתוח מערכת ההפעלה נעשה בדרך כלל בשפת מכונה.)

כלל החריגות

שפות תכנות אשר זוכות לפופולריות אינן שוקטות על שמריהן. הניסיון בשימוש בשפה על ידי קהילת משתמשים גדולה מביא לעיתים תכופות לבקשות לשינויים בשפה. כך למשל, העדר בדיקה של פרמטרים לפונקציות התגלה כענין כאוב בשפת C. המענה הראשון של מפתחי השפה היה פיתוח כלי נוסף (על פי עיקרון מודולריות הכלים) הנודע בשם lint[8]. בכל זאת, מענה זה לא היה מספק, והשפת התפתחה הלאה. בפרט Ansi-C המהדורה הראשונה של השפה שזכתה לסטנדרטיזציה, הרחיבה את הדקדוק של השפה כדי לתמוך בבדיקת פרמטרים לפונקציות, אם כי הותירה את האפשרות לכתוב פונקציות אשר הפרמטרים אליהם לא נבדקים. מהדורות נוספות של השפה הן C99 ו C11.

פיתוח השפה נעשה בדרך כלל תוך התחשבות בחזון התכנון שלה. כך למשל, ב C99 נוסף הטיפוס

```
long long int
```

[8] משמעות המילה lint בשפה האנגלית היא "מוך". שם הכלי מעלה אסוציאציה לנוקדן אשר גוחן לנקר את המוך מבגד חגיגי שלובש חברו.

אשר נועד לאפשר תמיכה במכונות שבהן יש מילים גדולות במיוחד, ואשר לא היו בנמצא בזמן הפיתוח המקורי של השפה. אולם לעיתים, כפי שראינו בדוגמא של בדיקת פרמטרים, המהדורות החדשות מפירות את עקרונות התכנון. כך למשל, במהדורות מודרניות של C, ניתן להעביר רשומות (אך לא מערכים) כפרמטר by value.

במקרים אחרים, הפתחות השפה והמענה לקהילת המשתמשים נעשה באמצעות פיתוח דיאלקטים שונים. זו המצב למשל בשפת פסקל, אשר הפופולריות שלה כשפת הוראה באקדמיה הביאה לפופולריות בתעשיה, ועל כן להתפתחות מעבר לחזונו של מתכנן השפה.

ככלל, אנו יכולים על כן לצפות לחריגות מכללים. העקרונות היסודיים של תכנון השפה, במיוחד בשפות אשר זכו לפופולריות רבה, עלולים להשתנות. מתברר גם כי ככלל, מרבית הכללים שנלמד בקורס זה, זוכים לחריגות משלהם. שפות תכנות הם יציר מוחם הקודח של בני אנוש, אשר נוטים מטבעים לבדוק את הגבולות והכללים, ולהשתעשע בחריגות מהם.

כיצד מגדירים שפות תכנות?

דקדוק ומשמעות

לאחר קביעת החזון ועקרונות התכנון, על מתכנן השפה להגדיר את השפה בפרוטרוט. יש שתי דרכים עיקריות לעשות זאת. ראשית ניתן להגדיר את השפה באמצעות המימוש שלה: כלומר באמצעות כתיבת מהדר לשפה, וההחלטה שמימוש זה מהווה את הגדרת השפה. כזהו המצב למשל בשפות Perl ו PHP. שיטה זו קרוייה Reference Implementation.

הדרך האחרת, המקובלת יותר, והנוחה יותר למשתמשים, היא הגדרת השפה באופן בלתי תלוי במימוש שלה, וגזירת המימוש מתוך ההגדרה הזו. כדי לעשות זאת, יש לבצע שתי הגדרות נפרדות: האחת בעבור הדקדוק של השפה (syntax) והאחרת בעבור המשמעות (semantics) שלה.

הגדרת הדקדוק קובעת אימתי סדרה של תווים היא תכנית חוקית בשפה. בהגדרת הדקדוק נמצא התייחסות לענינים כגון ביטויים, פקודות וכו'. הגדרת המשמעות קובעת את המשמעות, כלומר אופן הביצוע החוקי של כל תכנית חוקית: כיצד יש לחשב ביטוי, כיצד יש לשערך פקודה, וכיוצ"ב.

יתכן שבשפת תכנות יהיו מספר אפשרויות דקדוקיות לבטא את אותו ענין סמנטי. אפשרויות אלו, מקלות על המשתמש בשפה לבטא באורח תמציתי יותר את אותו רעיון. כך למשל, בשפת C הסמנטיקה של שלושת הביטויים הבאים היא זהה:

- `y = (++x)`
- `y = (x += 1)`
- `y = (x = x + 1)`

לאפשרות שנותנת שפה להשתמש בדקדוק קצר יותר קוראים בשם **Syntactic Sugar**

אידיאלית, הגדרות הדקדוק והמשמעות הן נפרדות. אבל, לעיתים הקשר ביניהן הוא בלתי נמנע. למשל, כדי לבדוק אם תכנית בפסקל היא חוקית, יש לבדוק אם כל משתנה שבו נעשה שימוש הוגדר קודם לכן, ואם השימוש בו מתאים להגדרה. בדיקה זו דורשת הבנה מסויימת של המשמעות, ובפרט הכללים על פיהם הגדרת משתנה במקום מסויים מוכרת במקום אחר, כמו גם הכללים הקובעים אימתי שימוש במשתנה מסויים הוא חוקי.

יש מספר כלים נפוצים (פורמליזמים) להגדרת דקדוק, ובהם:

1. **ביטויים רגולריים:** המשמשים להגדרות דקדוקיות פשוטות. כך למשל, ההגדרה של משתנה חוקי בשפת C כביטוי רגולרי נראית כך:

```
(_a-zA-Z)(_a-zA-Z0-9)*
```

2. **דקדוקי BNF:** המשמשים להגדרת הדקדוק המלא של שפת התכנות. כך למשל, הגדרת פקודת תנאי בפסקל בפורמליזם זה נראית כך:

Statement →*IfStatement*

IfStatement → `if` *Expression* `then` *Statement ElsePart*

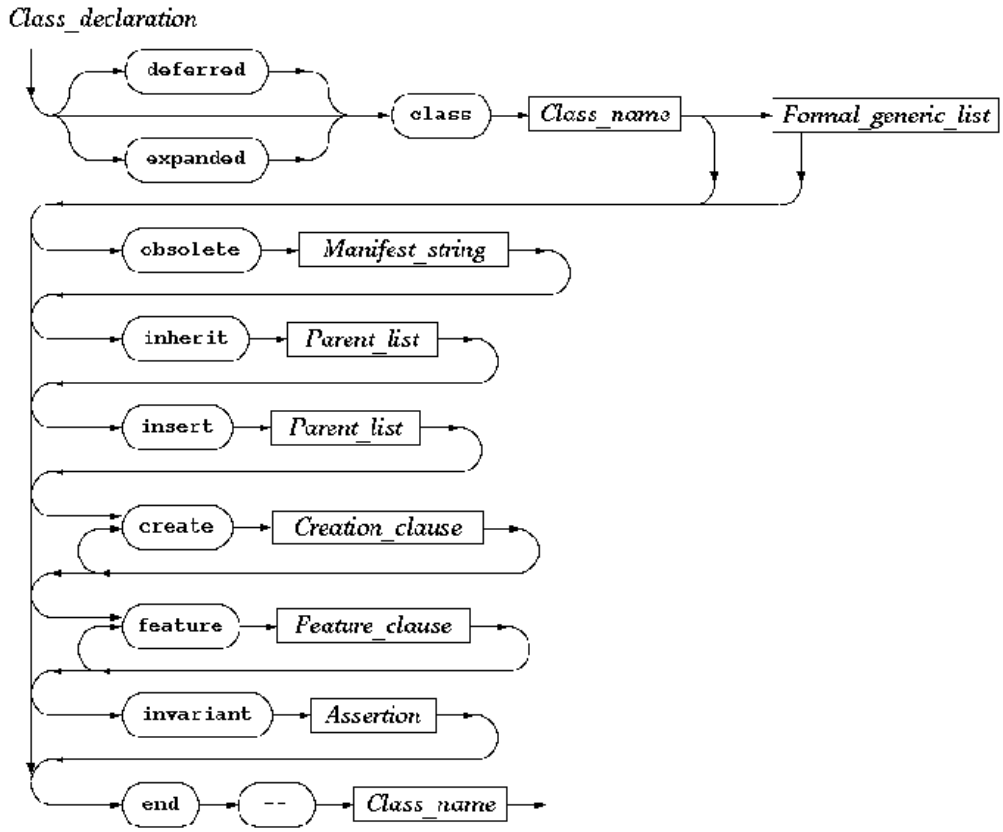
ElsePart → **ε**

ElsePart → `else` *Statement*

3. **דקדוקי EBNF:** אשר מתירים Syntactic Sugar נוסף ביחס לדקדוקי BNF תוך שימוש בסימונים המשמשים בביטויים רגולריים. הגדרת פקודת תנאי בפסקל בפורמליזם זה נראית כך:

IfStatement → `if` *Expression* `then` *Statement* [`else` *Statement*]

4. דיאגרמות דקדוקיות. אנו לא נעסוק בקורס בהגדרות דקדוקיות באופן זה, אך למען השלמות, הנה דוגמא של הגדרה של הדקדוק של מחלקה בשפת Eiffel בכתיב זה:



למרבה הצער, בדיקה של כללי משמעות סוג זה שתואר לעיל, אינה מתאפשרת באמצעות הפורמלליזמים הללו. על כן, נהוג לכלול את בדיקת הטיפוסים (למשל) בתוך בדיקת המשמעות.

הכלים הנפוצים לתיאור משמעות של תכנית הם:

1. **פרוזה**: כלומר, תיאור לא פורמלי בשפה טבעית של המשמעות של השפה. תיאור פרוזאי כזה מצוי למשל ברוב ספרי הלימוד של שפות התכנות.
2. **Programming Languages Legalese**: זהו התיאור של שפת התכנות בסטנדרטים רשמיים, המבוסס על תיאור בשפה "משפטית" שהיא מאוד מדוייקת מצד אחד אך קשה להבנה מן הצד האחר.
3. **סמנטיקה פורמלית:** הסמנטיקה הפורמלית היא פיתוח של לוגיקה כדי לתאר בצורה מדוייקת ופורמלית את המשמעות של השפה.

   השימוש בסמנטיקה פורמלית הוא נפוץ במחקר, אך חשיבותו בפועל היא מועטה, שכן יש בסמנטיקה הפורמלית שימוש כבד ביותר בסימונים מתימטיים, וניסוח מסורבל ולא תמציתי של המשמעות באמצעות סימנים אלו. כתוצאה מכך, השימוש בסמנטיקה פורמלית עבור שפות תכנות הנמצאות בשימוש פעיל, הוא משימה הרקוליאנית שנדיר שיש מי שמבצע אותה.

   בכל זאת, ניתן לקבל תוצאות חלקיות באמצעות כלים פורמליים כאלו. כך למשל, ישנה הוכחה חלקית

לכך ששפת ג'וה היא בטוחה, במובן זה שתוכנית בשפה זו אינה יכולה לעשות שגיאות טיפוס.[9]

הקורס הזה מאמץ את הדרך שבה מרבית משתמשי שפות התכנות לומדים את השפה שלהם, כלומר הגדרות בשטה טבעית ולא משפטית של שפת התכנות.

קבוצות מוגדרות רקורסיבית

סעיף 4ב בחוק השבות, תש"י - 1950 קובע: **לענין חוק זה, "יהודי" - מי שנולד לאם יהודיה או שנתגייר, והוא אינו בן דת אחרת".**[10] אנו רואים שההגדרת מיהו יהודי היא רקורסיבית, שכן הגדרת המונח יהודי בחוק מבוססת על המונח יהודי עצמו.

שיטת הגדרה רקורסיבית שכזו (הקרוייה גם הגדרה אינדוקטיבית), היא מקובלת ומועילה מאוד.

להגדרת קבוצה באופן רקורסיבי, יש שלושה מרכיבים:

1. רשימה של **איברים אטומיים** אמר משתייכים לקבוצה.
2. רשימה של כללי יצירה המאפשרים יצירה של **איברים מורכבים**, מאיברים אטורמיים ואיברים מורכבים שנוצרו כבר. כל **כלל יצירה** נוטל אחד או יותר מאיברי הקבוצה, ובונה מהם איבר אחר.
3. הדרישה, אשר נוהגים להשמיטה למען הקיצור, כי אין בקבוצה איברים אחרים פרט לאטומיים ולמורכבים שנוצרו באמצעות כללי היצירה. .

בדוגמא של חוק השבות, האיברים האטומיים הם אברהם אבינו, וחשוב מכך, שרה אמנו, וכל מי שנתגייר, ואילו היצירה הוא הלידה.

נשתמש בשיטה זו כדי להגדיר את אוסף הפקודות בפסקל. האיברים האטומיים, הלא הם הפקודות האטומיות, הם שלושה:

1. הפקודה הריקה
2. פקודת ההצבה
3. קריאה לפרוצדורה

כללי היצירה העיקריים של הפקודות הם:

1. שרשור של פקודות המופרדות על ידי סימן הנקודה ופסיק (`;`) הוא פקודה.
2. פקודת תנאי, כפי שתוארה לעיל, היא פקודה, אשר מכילה בתוכה פקודה אחת או שתיים.
3. פקודת תנאי רבת ראשים המוגדרת באמצעות המילה השמורה **`case`**
4. לולאות המתארות ביצוע איטרטיבי של פקודה (אטומית או מורכבת), אף הן פקודות. יש בפסקל שלושה סוגים של פקודות לולאה מורכבות:
   - **`for`**
   - **`while`**
   - **`repeat until`**

ניתן גם באופן דומה להגדיר את הביטויים בשפת תכנות פשוטה. הביטויים האטומיים יהיו מספרים ושמות משתנים, ואילו כללי היצירה יהיו מבוססים על פעולות החשבון וסימני הסוגריים.

---

[9] עיינו למשל במאמר Java is Type Safe, Probably

[10] הגדרה רקורסיבית דומה קיימת ביחס לשאלה "מיהו מוסלמי?". על פי ההלכה המוסלמית, מוסלמי הוא מי שנולד לאב מוסלמי, או שהפך למוסלמי באמצעות אמירת העדות, הלא היא השהאדה: لَا إِلَٰهَ إِلَّا اللهُ مُحَمَّدٌ رَسُولُ اللهِ בפומבי. אבל, כיוון שהאמירה חייבת להעשות בפני מי שהם מוסלמים, ההתאסלמות מהווה כלל יצירה של איבר מורכב. אגב, ההמרה לנצרות, לפחות על פי כתבי הקודש דורשת גם היא אמירה, אך פרטית, ולא פומבית, ככתוב באיגרת פולוס השליח אל הרומיים, פרק י"א "כִּי אִם־בְּפִיךָ תוֹדֶה שֶׁיֵּשׁוּעַ הוּא הָאָדוֹן וְתַאֲמִין בִּלְבָבְךָ שֶׁהָאֱלֹהִים הֶעִירוֹ מִן־הַמֵּתִים תִּוָּשֵׁעַ: כִּי בִלְבָבוֹ יַאֲמִין הָאָדָם וְהָיְתָה לוֹ לְצֶדָקָה וּבְפִיהוּ יוֹדֶה וְהָיְתָה־לוֹ לִישׁוּעָה: כִּי הַכָּתוּב אֹמֵר כָּל־הַמַּאֲמִין בּוֹ לֹא יֵבוֹשׁ: וְאֵין הַפְרֵשׁ בֵּין הַיְּהוּדִי לַיָּוָנִי כִּי אָדוֹן אֶחָד לְכֻלָּם".

ביטויים רגולריים

בהינת אלפאבית $\sum$ של סימנים יסודיים, נגדיר כ $\sum^*$ את אוסף כל **הסדריות**(Strings) הסופיות שניתן לכתוב בעזרתו, ובכלל אלו את הסדרית הריקה אשר מסומנת בדרך כלל כ ε. במנוחים אלו **שפה פורמלית** היא פשוט תת-קבוצה של $\sum^*$. ביטויים רגולריים (regular expressions), הם מכשיר להגדרת שפות פורמליות.

קבוצת הביטויים הרגולריים מעל $\sum$ מוגדרת רקורסיבית באופן הבא:

1. ביטויים אטומים:
   - כל תו ב $\sum$ הוא ביטוי רגולרי אטומי המתאר שפה המכילה סדרית אחת בלבד, התו עצמו.
   - הסימן ε הוא ביטוי רגולרי אטומי, המתאר שפה המכילה את הסדרית הריקה בלבד.
2. כללי יצירה
   - אם R_1 ו R_2 הם ביטויים רגולריים, אזי
     - R_1R_2 הוא ביטוי רגולרי שהשפה שלו מורכבת מכל הסדריות שאפשר לחלק אותם לשני חלקים רצופים וזרים, כאשר החלק הראשון, הרישא של הסדרית, שייך לשפה של R_1 ואילו החלק השני, הסיפא, שייך לשפה של R_2.
     - $R_1 \mid R_2$ הוא ביטוי רגולרי שהשפה שלו היא האיחוד של שתי השפות של R_1 ו R_2.
   - אם R הוא ביטוי רגולרי, אזי גם R^+ הוא ביטוי רגולרי, שהשפה שלו היא כל הסדריות שניתן לחלק אותן לסדריות זרות ורצופות אשר כל אחת מהן שייכת לשפה של הביטוי הרגולרי R.

במרבית השימושים בפועל מוסיפים Syntactic Sugar כגון:

- $R?$ הוא קיצור עבור $\varepsilon|R$
- R^* הוא קיצור עבור $\varepsilon|(R+)$
- u-v הוא קיצור עבור כל התווים שמצויים בין התווים u ל v, תוך הנחה שיש סדר מוסכם של התווים באלפאבית $\sum$
- הסימן "." (נקודה) מתאר את הביטוי הרגולרי שמכיל אות אחת בדיוק מהאלפאבית.
- ניתן לקצר את הכתיבה באמצעות מתן שמות לביטויים רגולריים חלקיים.

הנה דוגמא לשימוש ב Syntactic Sugar כדי להגדיר ביטוי רגולרי בעבור מזהה חוקי בשפת C:

```
Digit = 0-9
Lower = a-z
Upper = A-Z
Letter = Lower|Upper
CLetter = Letter|_
IdCharacter = CLetter|Digit
Identifier = CLetter IdCharacter*
```

כדאי לדעת כי ניתן לכתוב ביטוי רגולרי עבור חיתוך של השפות של שני ביטויים רגולריים, וגם עבור השפה של כל הסדריות של ביטוי רגולרי אחד אשר אינן מצויות בשפה של ביטוי רגולרי אחר, וזאת בעבור כל שני ביטויים רגולריים שרירותיים.

השימוש בביטויים רגולריים בתיאור הדקדוק של שפה, אינו חשוב רק למען הדיוק. ישנם כלים אוטומטיים אשר הקלט שלהם הוא ביטוי רגולרי ואשר מייצרים תכניות המסוגלות לזהות מופעים של ביטויים רגולריים בטקסט. כלים אלו מועילים מאוד בביטויים בכתיבת מהדרים עבור שפות תכנות.

דקדוקי BNF

כוח הביטוי של ביטויים רגולריים הוא מוגבל ביותר. כך למשל, לא ניתן לבטא באמצעות ביטוי רגולרי את הדרישה שהסוגריים בתכנית מאוזנים. לפיכך, השימוש בביטויים רגולריים מוגבל להגדרות פשוטות של אבני הבנין של השפה: משתנים, הערות, מספרים וכו'.

להגדרות מורכבות יותר, יש להשתמש במנגנון הידוע בשם דקדוק חסר הקשר (Context Free Grammar), אשר נכתב בדרך כלל בשיטת הסימון הידועה בשם Backus Naur Form או BNF. כתיב אחר לדקדוקים אלו הוא הדיאגרמות שתוארו לעיל.

הגדרת דקדוק בשיטת סימון זו מורכבת מארבעה חלקים:

1. קבוצה של סימנים סופיים, Termials. (ה Terminals קרויים לעיתים גם Tokens או אסימונים). הדקדוק מתאר שפה מעל האלפאבית שיוצרים הסימנים הסופיים.
2. קבוצה של סימנים לא סופיים, Non Terminal Symbols, המשמשים ככלי עזר להגדרת הדקדוק. סימני עזר אלו דומים מעט לשימוש בשמות לביטויים רגולריים חלקיים, אלא, שהגדרתם של סימני העזר הללו יכולה להיות רקורסיבית, וניתן להגדירן יותר מאשר פעם אחת.
3. קביעה של אחד מהסימנים הלא סופיים כסימן התחלה: Start Symbol
4. אוסף של כללי גזירה, כאשר לכלל גזירה יש שני חלקים: ראש הכלל הכתוב בצד שמאל של הכלל,, הוא תמיד סימן לא סופי, ואילו גוף הכלל הכתוב בצידו הימני, הוא סדרית (היכולה להיות ריקה) של סימנים סופיים ולא סופיים. כללי הגזירה נכתבים כך שישנו חץ המוביל מראש הכלל אל גופו.

בפועל, נוהגים להשמיט את המרכיבים 1 עד 3 של הגדרת הדקדוק ולהסתפק בכללי הגזירה לבדם. קל להבחין בין סימנים סופיים ולא סופיים בכללי הגזירה, משום שסימן סופי לא יופיע לעולם בראש כלל. גם סימן ההתחלה ברור בדרך כלל מההקשר.

הנה דוגמא פשוטה ביותר המדגימה זאת:

$S \rightarrow E$
$E \rightarrow$a E b
E $\rightarrow$ ε

בדוגמא זו בדקדוק ישנם שלושה כללי גזירה, אשר מקריאתם מתגלה כי:

1. הסימנים הלא סופיים הם S ו E
2. סימן ההתחלה הוא S
3. הסימנים הלא סופיים הם a ו b

השפה המוגדרת על ידי הדקדוק חסר ההקשר הזה היא פשוטה ביותר, והיא מכילה את כל הסדריות שבראשן יש מספר n (שיכול להיות 0) של a ואחריהם n מופעים של הסימן b.

ניתן להוכיח (ולא נעשה זאת כאן), כי לא ניתן להגדיר שפה זו באמצעות ביטויים רגולריים.

הנה דוגמא המהווה קטע של הדקדוק של שפת פסקל:

pascal-program → **program** *identifier program-heading* **;** *block* **.**
program-heading → **ε**
program-heading → **(** *identifier-list* **)**
identifier-list → *identifier*
identifier-list → *identifier-list* **,** *identifier*
block → *block1*
block → *label-declaration* **;** *block1*
block1 → *block2*
block1 → *constant-declaration* ; *block2*
block2 → *block3*
block2 → *type-declaration* **;** *block3*
block3 → *block4*
block3 → *variable-declaration* **;** *block4*
block4 → *block5*
block4 → *proc-and-func-declaration* **;** *block5*
block5 → **begin** *statement-list* **end**
...
type-declaration → **type** *type-declarator*
type-declaration → *type-declaration* **;** *itype-declarator*
type-declarator → *identifier* **=** *type*
...
type → *identifier*
type → **record** *field-list* **end**
field-list → **ε**

קל לזהות בהגדרת דקדוק זו את סימן ההתחלה. לשם הנוחות סימנו את הסימנים הסופיים כגופן וצבע מיוחדים. מהגדרת הדקוק הזו אנו למדים למשל:

1. תכנית פסקל מתחילה תמיד במילה **program** ומסתיימת בסימן "**.** "
2. לתכנית יש שם שאחריו יכולה להפועים רשימת מזהים העטופה בסוגריים עגולים.
3. בראש התכנית יש ארבעה פרקי הגדרות החייבים להופיע בסדר קבוע: הגדרת תוויות, הגדרת קבועים, הגדרת טיפוסים והגדרת משתנים.
4. כל אחד מארבעת מפרקי ההגדרות הוא אופציונלי.
5. בהגדרת פרק הטיפוסים אם מופיעה המילה **type** אזי אחריה חייבת להופיע הגדרת טיפוס אחת לפחות.
6. הגדרות הטיפוסים חייבות להיות מופרדות בסימן "**;**"
7. כל הגדרת טיפוס בודדת מכילה מזהה, סימן שיוויון, ואחריו גוף הטיפוס, שיכול להיות מזהה או רשומה.

והנה דוגמא לתכנית פשוטה (וסרת טעם) המצייתת לדקדוק לעיל:

```
program p;
type
     shalem = integer;
     student = record
               end;
begin
end.
```

הגדרת הדקדוק של שפת תכנות באמצעות דקדוק חסר הקשר לא נועדה למען הדיוק בלבד. ישנם כלים אוטמטיים המאפשרים תרגום של דקדוק חסר הקשר כזה לתכנית ניתוח, אשר לוקחת טכסט נתון, ובונה בעבורו את אופן גזירתו מהדקדוק.

אופן הגזירה הזה נקרא "עץ גזירה" (Parse Tree) אשר מהווה הוכחה כי הטכסט אמנם נגזר מהדקדוק.

הפורמליזם של דקדוק BNF חזק יותר מהפורמליזם של ביטויים רגולריים שכן הוא מתיר הגדרות רקורסיביות. כך למשל, בהגדרת הדקדוק של פסקל נמצא הגדרות רקורסיביות שבהן הסימן הלא סופי *statement-list* מוגדר באמצעות הסימן הלא סופי *statement* ולהיפך:

statement-list → *statement*
statement-list → *statement-list* ; *statement*
statement → *ε*
statement → *variable* **:=** *expression*
statement → **begin** *statement-list* **end**
statement → **if** *expression* **then** *statement*
statement → **if** *expression* **then** *statement* **else** *statement*
statement → **case** *expression* **of** *case-list* **end**
statement → **while** *expression* **do** *statement*
statement → **repeat** *statement-list* **until** *expression*
statement → **for** *varid* **:=** *for-list* **do** *statement*
statement → *procid*
statement → *procid* **(***expression-list***)**
statement → **goto** *label*
statement → **with** *record-variable-list* **do** *statement*
statement → *label* **:** *statement*

הגדרות רקורסיביות מעין אלו הינן חיוניות בהגדרת שפות תכנות מודרניות, אך הן אינן ניתנות להיעשות בביטויים רגולריים.

דקדוקי EBNF

EBNF הוא קיצור של Extended BNF. פורמליזם זה דומה בעיקרו לפורמליזם של דקדוק BNF, אלא שגופו של כלל הגזירה יכול להיות ביטוי רגולרי מעל אוסף הסימנים, הסופיים והלא סופיים כאחד. שימוש בביטויים רגולריים כאלו הוא בבחינת Syntactic Sugar לדקדוקי BNF.

ההרחבה עצמה אינה מאפשרת הגדרת שפות פורמליות נוספות פרט לאלו הניתנות להגדרה בדקדוק BNF, אך ניתן באמצות הרחבה זו להגדיר שפות פורמליות ביתר תמציתיות.

הנה שכתוב של קטע הדקדוק הראשון של פסקל שהבאנו כאן, תוך שימוש בשיטות הסימון של EBNF.

```
pascal-program → program identifier [ ( identifier {,identifier} ) ] ; block .
block → [label-declaration ;]
        [constant-declaration ;]
        [type-declaration ;]
        [variable-declaration ;]
        begin statement-list end
...
type-declaration → type type-declarator {; type-declaration}
type-declarator → identifier = type
type → identifier | record field-list end
field-list → ε
```

כדאי לשים לב לכך שהכתיב של ביטויים רגולריים בגוף כלל הגזירה של EBNF הוא מעט שונה. למעלה, בדוגמא הזו השתמשנו בכתיב על פיו

- חזרה אפס או יותר פעמים מסומנת על ידי עטיפה הביטוי החוזר בסוגריים מסולסלים, המעוצבים טיפוגרפית בדוגמא כך: {} כך למשל תת הביטוי המופיע בגופו של כלל הגזירה הראשון לעיל

  identifier { , *identifier*}

  מציין רשימה של אחד או יותר מזהים המופרדים בפסיקים.
- ביטוי אופציונלי עטוף בסוגריים מרובעים, המעוצבים טיפוגרפית בדוגמא כך: [] כך למשל תת הביטוי

  [*label-declaration* ;]

  מציין שה *label-declaration* שאחריו יש סימן ; הוא אופציונלי.
- עוד נשים לכך שהדוגמא מניחה כללי קדימות של האופרטורים היוצרים את הביטוי הרגולרי, בפרט

  identifier | **record** *field-list* **end**

  מתפרש כך:

  identifier | (**record** *field-list* **end**)

  ולא כך:

  (*identifier* | **record**) *field-list* **end**

דקדוק ה EBNF של שפת תכנות מסויימת עשוי להשתמש בשיטת סימון מעט אחרת ואולי אף כללי קדימות אחרים. בדרך כלל, יכול הקורא הנבון להסיק את שיטת הסימון מתוך הקריאה, ואילו הקורא הסכל, אך הסבלן, יוכל לעיתים לאתר את מובנם של הסימונים במקום אחר.

שלבים קונצפטואליים בעיבודן של שפות תכנות

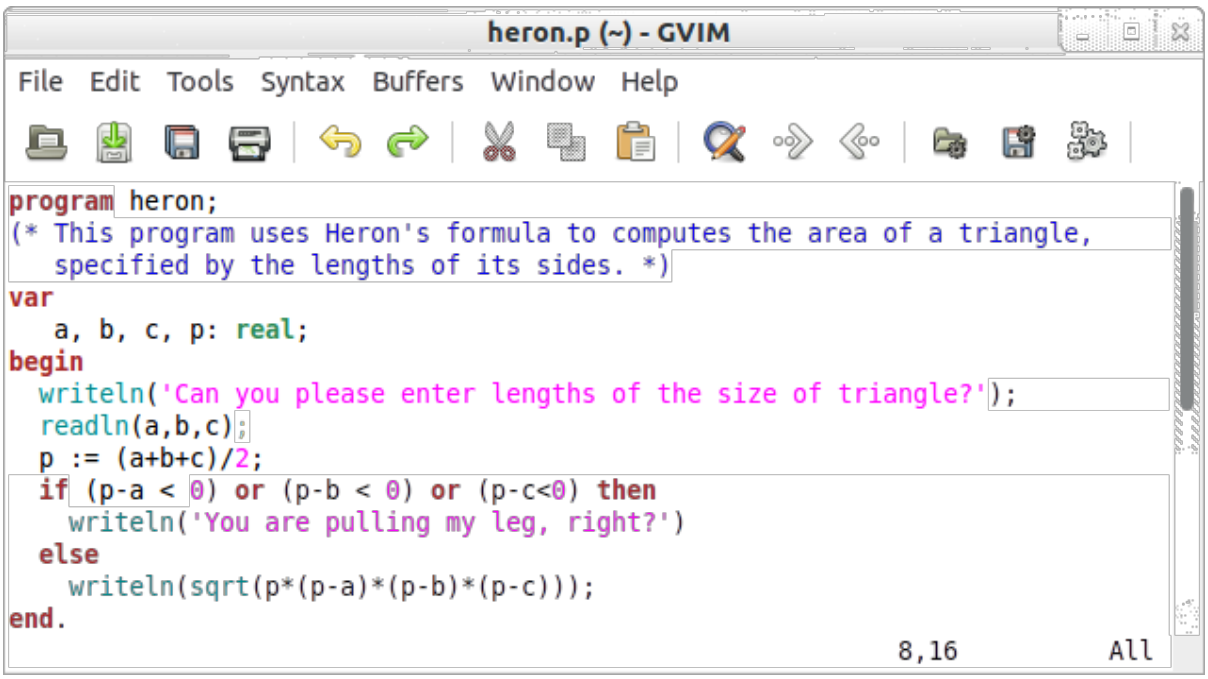
ברגיל, הידורן של שפות תכנות נעשה במספר שלבים (קונצפטואליים):

1. עיבוד מקדים, הכולל הסרת ההערות.
2. ניתוח לקסיקלי באמצעות כלי אוטומטי לעיבוד ביטויים רגולריים, אשר מחלק את הטכסט לאסימונים (tokens).
3. בניית עץ גזירה עבור השפה באמצעות כלי אוטומטי לעיבוד דקדוקים.
4. גיזום של הנתונים הבלתי רלבנטיים מעץ הגזירה, והותרת עץ גזירה אבסטרקטי. כך למשל המילים השמורות **`if`** ו **`then`** בפסקל הן נתון טפל, אשר מקודד ממילא במבנה העץ.
5. בדיקה סמנטית המשלימה את הבדיקה הדקדוקית אשר עשה המנתח. בדיקה זו מוודאת למשל כי ישנה הגדרה לכל פונקציה אשר נקראת מהתוכנית, וכי הגדרת הפונקציה תואמת את השימוש בה.
6. תרגום לשפת מכונה ואופטימיזציה.

אבני הבנין של שפות התכנות

פרק זה הינו סקירה חלקית של "צעדים ראשונים", אשר על הסטודנטים ללמוד באופן עצמאי.

נתבונן בקוד בשפת תכנות כלשהי הנכתב בעורך vim. נשים לב כי העורך צובע אלמנטים שונים בצבעים שונים לפי סוג האלמנט. העובדה שהעורך מכיר מספר רב של שפות, ומסוגל לצבוע בצבעים אחידים את האלמנטים שלהם, מעידה על כך שישנם אבני בנין יסודיות המשותפות לכל שפות התכנות.



סוגי האלמנטים הנפוצים הם:

1. **מזהים** (identifiers) שם המציינישות אחרת. כך למשל "נדב" הוא שמו של אדם העונה לשם זה והשם אינו האדם עצמו. מזהים הם שמות המציינים ישות כלשהי כגון: טיפוס, template, משתנה, ערך, פונקציה, פרודצורה ושאר ירקות. סוגי הישויות תלויים כמובן בשפת התכנות.
מתן שם לישות בשפת תכנות מאפשר שימוש חוזר באותה הישות, מבלי ליצור אותה מחדש, ובכך תומך בתכנות מודולרי.

2. **מילולונים** (literals) : המילולונים מהווים מנגנון לציון ישויות שאין להן שם, לבד מעצמן. כך למשל אין שם למספר 13 לבד מהמספר עצמו. המילולון מתפרש באופן מילולי, והוא מציין את עצמו.[11] דוגמאות לכך הינן סדריות ומספרים.
3. **הערות**: אלו לא מהוות חלק תכנותי בקוד עצמו ומשמשות לנוחיות המתכנת והקורא.
4. **מילים שמורות** אלו הן מילים שמבחינה דקדוקית הם מזהים, אך השימוש בהם נאסר על המתכנת, שכן השפה שומרת אותן למטרה אחרת.
5. **אופרטורים**: אלו הם סימני פעולה למיניהם. למשל *, - , /, +....
6. **סימני פיסוק:** לדוגמא **;**

נעיין בתכנית הדוגמא הקצרה שהוצגה למעלה:

```
program p;
type
     shalem = integer;
     student = record
               end;
begin
end.
```

בתכנית זו, המזהים הם: p, shalem, integer ו student, המילים השמורות הן **program, type, record, end** ו **begin,** וסימני הפיסוק הם ";", "=" ו ".". בתכנית אין הערות, אופרטורים או מילולונים.[12]

המזהים והמילולונים מתוארים בדרך כלל על ידי ביטויים רגולריים. גם כל אחד מן האופרטורים, מן המילים השמורות, ומן סימני הפיסוק ניתן לתיאור באמצעות ביטוי רגולרי, אלא שהביטוי הזה הוא פשוט במיוחד, שכן הוא מציין שפה פורמלית בה יש איבר אחד ויחיד. אוסף כל הביטויים הרגולריים הללו הוא זה שיוצר את הניתוח הלקסיקלי של התכנית. אשר בה מתולק הטקסט לאסימונים.

כדאי לשים לב לכך שהניתוח לאסימונים אינו תמיד פשוט ומיידי, אף אם מדובר באופרטורים וסימני פיסוק פשוטים, שכן יתכן כי אופרטור יהיה מורכב מצירוף של מספר תווים. לדוגמא, זיהוי האסימונים בביטוי החוקי הזה בשפת C דורש תשומת לב, לא רק מבן אנוש, אלא גם מתכנית מחשב.

ההערות אף הן יכולות להיות מתוארות על ידי ביטוי רגולרי, אלא שהן מוסרות בדרך כלל באמצעות עיבוד מקדים על ידי קדם מעבד שאינו נצרך להפעלת הארטילריה של מנתח ביטויים רגולריים.

[11] וזאת בדומה לילד קטן האומר "זה" בכדי לציין דברים עליהם הוא מצביע.
[12] נסה לזהות את האלמנטים הללו בתמונת המסך של העורך GVIM שהופיע למעלה.

מזהים לעומת מילולונים

בכדי להעמין בהבנת ההבדל בין מזהה לבין מילולון, נעיין בחידה הבאה מאת ר' אברהם אבן עזרא (הראב"ע, 1089 –1164):

תחילת שמי כמו אמצע שמי, אמצע שמי כמו תחילת שמי, סוף שמי כמו מחצית סוף שמי. מהו שמי?

בקריאה ראשונה, חידה זו נראית כפרדוקסלית. ראשית, תחילת השם המבוקש זהה לאמצעו, ומכאן, נובע שאמצע השם זהה לתחילתו, ויש לתמוה על כן מדוע ראה בעל החידה להוסיף פרט מיותר זה).אך קשה מכך היא הקושיה כיצד סוף השם יכול להיות שווה למחצית סוף השם? הלא הדבר יתכן רק אם סוף השם הוא המספר 0?

המיסתורין יוסר בכתיבה של החידה מחדש תוך אבחנה בין מילולון ומזהה.

תחילת שמי כמו אמצע "שמי", אמצע שמי כמו תחילת "שמי", סוף שמי כמו מחצית סוף "שמי". מהו שמי?

האבחנה הזו מאפשרת לנו להבין כי המילה שמי מציינת את שמו של מי אשר חד את החידה, ואילו "שמי" מציין את עצמו, כלומר את הסדרית "שמי" עצמה. מכאן ברור כי התשובה לשאלה היא השם "משה".[13]

ומי אשר הראב"ע מרובע מדי לטעמו, ידרש לדבר מנחם המשעמם, לאמור:

בתאוריה יש ארבע אותיות אהו"י, אבל במציאות יש רק שלוש

כדי להבין זאת, נוסיף מרכאות, המציינות מילולונים. כלומר נכתוב "תאוריה" כדי לציין את הסדרית בת שש התוים, ונכתוב - תאוריה - כאשר הכוונה למזהה של מושג מופשט.

ב"תאוריה" יש ארבע אותיות 'א' 'ה' 'ו' 'י', אבל ב"מציאות" יש רק שלוש[14]

שם לעומת ישות

כדאי לחדד את ההבדל בין שם ובין ישות. השם מציין את הישות והוא אינו זהה לה. לעיתים יתכן שלאותה ישות יהיה יותר מאשר שם אחד, ויתכן גם כי לישות מסויימת יהיו מספר שמות.

---

[13] הניסוח המקורי של הראב"ע קל יותר לפתרון: כי תרצה לדעת שמי // קח אמצע שמי והוא ראש שמי // קח ראש שמי והוא אמצע שמי // קח סוף שמי וחלקהו לשניים // ומצאת את שמי. ולמתקדמים הנה חידה נוספת לנעוץ בו את שיניהם: אמור מה שם צבי נכבד // אשר חציו חצי חציו // ועוד בשמו חצי נוסף // ורואיו יאמרו איו?

[14] הסברו המקורי של מנחם המשעמם היה: בתאוריה יש א' ו' י' ה', אבל במציאות יש רק י' א' ו', רק שלוש מתוך הארבע. ניכור שמנחם המשעמם לא היה בין המקפידים על סימון מרכאות.

נעיין למשל בקושיה הבאה, אשר התשובה עליה עשויה לעורר קושי אצל יהודים מאמינים שאין להגות את (מה שהם סבורים כי הוא) ה"תשובה הנכונה", ויתירה מכך, כי יש להתייחס ביראת כבוד אל טכסטים אשר התשובה הנכונה כביכול מופיעה בהם:

מהו השם המפורש?

מתברר כי התשובה לשאלה זהו היא קלה ואינה מעוררת כל קושי, שכן השם המפורש אינו אלוהים, והוא אינו גם שמו של אלוהים. השם המפורש הוא שמו של שמו של אלוהים. ליתר דיוק, השם המפורש הוא שם קיבוצי לשלושה "שמות מפורשים" של אלוהים:

1. Tetragrammaton, הטטראגאמאטון, הלא הוא השם המפורש בן ארבע אותיות.
2. השם המפורש בן 42 אותיות.
3. השם המפורש בן ה 216 אותיות המחולקות ל- 72 מילים שכל אחת מהן היא בת שלוש אותיות.[15]

בשפות תכנות יש ישויות למכביר אשר אין להן שם. הנה כמה דוגמאות:

- משתנה אשר הוקצה באמצעות `malloc` בשפת C הוא משתנה אשר אין לו שם.
- שפת התכנות ML כמו גם הרחבות של שפת C++ מאפשרת הגדרת פונקציות ללא שם. כך למשל, ההגדרה

```
fn x => x * x
```

מציינת פונקציה אנונימית אשר מחזירה את הארגומנט שלה כשהוא מועלה בריבוע.

- בשפת C אין שם (למשל) לטיפוס מצביע למצביע למספר שלם, כמו גם לשורה ארוכה של טיפוסים אחרים.
- בשפת C ניתן להגדיר טיפוס של רשומה, מבלי לתת שם לטיפוס, על העובדה שהגדרת טיפוס רשומה מלווה בדרך כלל במתן שם לטיפוס.

כך למשל, קטע הקוד הבא יוצר טיפוס חדש שהוא אנונימי, מגדיר משתנה מהטיפוס הזה אשר יש לו שם, ומאתחל משתנה זה.

```
struct {
        char *first;
        char *last
} boy = { "Danny", "Dean"};
```

המצב בו לאותה ישות יש יותר מאשר שם אחד, אף הוא אפשרי בשפות תכנות. כדי לראות דוגמא פשוטה לכך, נשווה רגע לנגד עינינו פרוצדורה בפסקל המעבירה משתנה by reference לפונקציה המוגדרת בתוכה. בתוך הפונקציה יהיו למשתנה הזה שני שמות: הראשון הוא עקב העובדה שהפונקציה המוכלת מכירה את כל המשתנים של הפרוצדורה המכילה, והשני, הוא שם הפרמטר המתאים. הנה תכנית המדגימה זאת:

```
program n;
```

[15] ידיעת השם המפורש הארוך הזה, והגייתו היא זו אשר מאפשרת לגיבורו של "פרקי שיר השירים" (אשר פתיחתו הובאה לעיל), לרחף באוויר ולעשות עוד מעשי קסם וגבורה (כך על כל פנים הוא מתרברב בפני לילי אהובתו-אחייניתו).

```
var a: integer;
  function t(var b:integer): boolean;
  (* This function seems to always return true, right? *)
  begin
    a := 1; b := succ(a); t := a <> b
  end;
begin
   writeln(t(a))
end.
```

בדוגמא זו, הסתכלות על הפונקציה t לבדה, עלולה להביא אותנו לחשוב שהפונקציה תחזיר תמיד את הערך true, שכן לא יתכן שמספר שלם יהיה שווה לעוקב לו. עיון מדוקדק יותר יגלה את הטעות. הקריאה לפונקציה תיצור בה מצב שבו לאותו משתנה יש שני שמות, ועל כן תוצאת הפונקציה תהיה false וזה גם יהיה הפלט של התכנית כולה.

מתן שם לשם אף הוא אפשרי בשפות תכנות. כך למשל בשפת הסקריפטים bash ניתן לפרש סדרית נתונה כשם של משתנה, לקרוא את תוכן המשתנה הזה, לפרשו גם כשם של משתנה אחר, וכן הלאה עד בלי די.

מילים שמורות

מקצת מהמילים השמורות הן מזהה שמור. כך, המילה **int** בשפת C היא **מזהה שמור** בכך שהיא מהווה את שמו של הטיפוס האטומי של מספר שלם בשפת C, אך בניגוד למזהים אחרים, אין אפשרות למתכנת לקשור את המזהה לישות אחרת.

באורח דומה, ניתן לחשוב על המילים **mod, div** בפסקל כמזהים שמורים, באשר הם שמותיהם של אופרטורים.

אחת הסיבות העמוקות יותר לקיומן של מילים שמורות, היא העובדה ששפות תכנות מסתמכות על מושג הקבוצה המוגדרת רקורסיבית כדי להגדיר את קבוצות הביטויים החוקיים, קבוצת הפקודות החוקיות, קבוצת הטיפוסים החוקיים ועוד.

למילים שמורות יש תפקיד חשוב בהגדרות אלו.

- ראשית, מילים שמורות מציינות לעיתים את בסיס הרקורסיה. כך למשל, הטיפוסים האטומיים בשפת C מצויינים על ידי מילים שמורות, או צירופים של מילים שמורות. למשל, **int** או **signed char**. למעשה, המילים השמורות הללו הן **מזהים** של הטיפוסים האטומיים. גם כמה מהפקודות האטומיות בשפת C מצויינות על ידי מילים שמורות, ובכלל אלו, **goto, break** ו **return**.
- לעיתים יש למילים השמורות תפקיד כסימני הפיסוק המגדירים כיצד מופעל כלל יצירה. כך למשל, הפעלת כלל יצירת פקודות התנאי המורכבות בשפת פסקל משתמש בפועל במילים השמורות **if, then** ו **else**.

יש מילים שמורות אשר תפקידן הדקדוקי הוא כזה של מילולון. בשפות מסויימות, המילים **true** ו **false** מציינות את הערכים הבוליאניים היסודיים, ולכן הן נקראות לעיתים מילולונים בתיאור השפה. לעיתים יש שימוש במילה השמורה **nil** או **null** כדי לציין את הערך של מצביע שאינו מצביע לדבר, וגם מילים אלו נקראות לעיתים מילולון בתיאור השפה.[16]

לעיתים המילים השמורות יש תפקיד בפיסוק של התכנית (וזאת מבלי שהן משתתפות בכלל יצירה של קבוצה המוגדרת רקורסיבית). כך למשל, כל תכנית בפסקל חייבת להתחיל במילה השמורה **program,** וארבעת הפרקים שבכותרת כל בלוק מצויינים באמצעות ארבעת המילים השמורות: **label**, **const, type** ו **var.**

[16] ניתן לכאורה להבחין בין המילה **true** ובין ערך האמת אותו היא מייצגת, ולטעון שהמילה היא **רק** שמו של ערך האמת הזה. באותו האופן ניתן גם להבחין בין היצוג הגרפי של המספר אחת, 1 ובין המספר עצמו. האבחנה הזו אינה מועילה לענייננו ואנו נדוש אותה בעקבינו.

כדאי לשים לכך שלעיתים שפת תכנות משתמשת באותה מילה שמורה לכמה תפקידים. תופעה הידועה בשם **העמסה**:

- המילה השמורה **var** בשפת פסקל משמשת הן לציון פרק המשתנים בבלוק, והן לציון העברת משתנים by reference.
- המילה השמורה **end** בשפה משמשת למספר תפקידים שונים.

למידת שפה באמצעות המילים השמורות שבה

דרך טובה להכרת שפת תכנות חדשה היא לימוד של המילים השמורות שבה, וניסיון לסווגן לקבוצות בהתאם לאמור לעיל. נעיין לשם דוגמא ברשימת המילים השמורת של שפת Lua:

and break do else elseif end false for function if in local nil not or repeat return then true until while

קל לזהות ברשימה זו את המילים השמורות המשמשות כסימני פיסוק המהווים בנאים של פקודות:

if else elseif then for do repeat until while

(הפקחים שבקוראים יציינו לעצמם לברר את ההבדל בין **else** ובין **elseif**, ובמיוחד מדוע לא ניתן להשתמש בצירוף **else if** במקום המילה השמורה הבודדת **elseif**.)

גם שתי המילים שמורות אשר משמשות לציון פקודות אטומיות קלות לזיהוי:

return break

יש ברשימה גם מספר מילים שמורות אשר הם שמות של אופרטורים:

and not or in

(עם זיהוי המילה in אנו נרשום לעצמנו לברר האם השפה נותנת תמיכה ישירה לקבוצות. בבואנו לברר שאלה זו, יהיה עלינו גם לזכור שאין בשפה מילה שמורה לציון קבוצה.)

אנו מוצאים בשפת Lua גם שלוש מילים שמורות שהן מילולונים:

true false nil

שתי המילים שנותרו, הן ככל הנראה סימני פיסוק, המאפשרים יצירת פונקציות והגדרת משתנים מקומית:

function local

מאתגר יותר לזהות ברשימת המילים השמורות את מה שאין בה. מהרשימה אנו למדים שאין בשפה פרוצדרות, ואין בשפה אין מילים שמורות עבור הטיפוסים האטומיים, ועל כן נרצה לבדוק גם נקודות אלו.

התחלת התכנית והגדרת גבולותיה

נוסחת הירון לחישוב השטח S של משולש שגדלי צלעותיו הן a, b ו c היא אלגנטית במיוחד:

$$S = \sqrt{p(p-a)(p-b)(p-c)}$$

כאשר

$$p = (a+b+c)/2$$

הבה נכתוב בזריזות תכנית פסקל בעבור הנוסחה הזו

```
Program h;
VAR a,b,c,p: real;
Begin
  readln(a,b,c);
  p := (a+b+c)/2;
  writeln(sqrt(p*(p-a)*(p-b)*(p-c)));
end.
```

מי שכתב אי פעם תכנית בשפת מכונה כלשהי, יכול לשוות בנפשו כיצד יתרגם המהדר את התכנית לעיל לשפת המכונה שעליה היא תרוץ, למעט ענין פעוט אחד: ידוע לכל כי ישנן פקודות מכונה לחישוב הסכום הסכום, המכפלה, ההפרש והמנה של שני מספרים ממשיים. אבל, כיצד יחושב השורש הריבועי? אפילו אם נניח כי יש המכונה מכילה פקודה לחישוב השורש, עדיין ישנה השאלה של התרגום של פקודות הקלט והפלט שבתכנית, שהרי אלו בוודאות אינן מצויות באוסף פקודות המכונה.

על כרחך אתה נאלץ לומר כי התרגום של תכנית בשפת תכנות מסויימת, פשוטה ככל שתהייה, לשפת מכונה, חייב להשתמש בפרודצרות ופונקציות אשר אינם מצויים בתכנית עצמה. השאלה בה עוסק פרק זה היא, בניסוח לא פורמלי, היא: "כיצד נקבעים גבולות התכנית?"[17], או בלשון אחרת, בהינתן שפת תכנות מסויימת, ובהינתן כל קטעי הקוד הקיימים בעולם הכתובים בשפה זו:

- אלו קטעים יוצרים תכנית מסויימת, ואלו לתכנית אחרת?
- האם יתכן שקטע קוד מסויים יהיה שייך ליותר מאשר תכנית אחת? ואם כן, מתי יקרה הדבר?
- היאך מוגדר בכלל "קטע קוד"?
- בהינתן כל קטעי הקוד מהם מורכבת התכנית, כיצד נקבע מאיזה מהם יתחיל הביצוע?

כדי לדעת את התשובה לכל אלו, בדרך כלל אין זה מספיק להכיר את הדקדוק והמשמעות של השפה, אלא נדרשת הבנה של הסביבה בה פועלת התכנית, והאינטרקציה של התכנית עם סביבתה.

הגישה הנקוטה בשפת פסקל, היא **הגישה האוטרקית**, על פיה יש רק קטע קוד אחד היותר תכנית, וקטע זה מצוי בקובץ אחד. כל הפונקציות והפרוצדורות אשר יכולות להיות מופעלות על ידי התכנית מצוייות בקטע זה, או מוגדרות על ידי שפת התכנות.

בפסקל ישנה גם מילה שמורה **program** המגדירה את התחלת התכנית, באשר הביצוע יתחיל במילה השמורה

[17] הקורא הכמה בשלב זה לארמז ספרותי או תיאולוגי, יכול ללמוד מעט על התהליכים היהודיים והנוצריים של קנוניזציה של כתבי הקודש, אשר בהם נקבע אלו מבין הספרים שייכים לתנ"ך. פלגים שונים בנצרות הגיעו לתוצאות שונות ביחס לשאלה זו. מעניין לציין כי בתנ"ך היהודי ישנם מה שהיינו קוראים שגיאות עקידה (Linking Error), ככתוב "כִּי אַרְנוֹן גְּבוּל מוֹאָב בֵּין מוֹאָב וּבֵין הָאֱמֹרִי. עַל-כֵּן יֵאָמַר בְּסֵפֶר מִלְחֲמֹת יְהוָה: אֶת וָהֵב בְּסוּפָה וְאֶת הַנְּחָלִים אַרְנוֹן..." אלא שספר מלחמות ה' אינו מצוי בידינו. (אגב, בברית החדשה הנוצרית, בכל הגדרותיה השונות, אין הפניה לספרים חיצוניים או לספרים שאבדו.)

begin המתאימה לה. בשפות אוטרקיות אחרות, כמו גירסאות ישנות של בייסיק למשל, תחילת הביצוע יכולה להקבע בדרך אחרת, למשל מהפקודה הראשונה בתכנית.

הגישה האוטרקית בפסקל נובעת מייעוד השפה ללומדים. לשם קיומו של חזון זה, בחר מתכנן השפה גם להימנע משימוש בספריות. כל הפונקציות והפרוצדורות אשר בהן יכול המתכנת בשפה להשתמש הן אלו שהוא כתב בעצמו, או אלו שמוגדרות כחלק מהשפה. בפרט, המפרט של הפונקציות `readln` ו `writeln` מצוי בהגדרת השפה עצמה. יוצר השפה בחר שלא להשתמש במילים שמורות בכדי לציין את הפונקציות והפרוצדורות המוגדרות על ידי השפה, אלא לקבוע שאלו הם מזהים מוגדרים מראש. הבחירה הזו מבטיחה ששירותים אלו יהיו זמינים למתכנת בכל עת, זאת, מבלי לסרבל את השפה בהגדרה של מספר גדול של מילים שמורות, אשר ימנעו מהמתכנת להשתמש במילים אלו לצרכיו.

רעיון זה מתפרס בפסקל לא רק על הפונקציות והפרוצדרות, אלא גם על הטיפוסים היסודיים, ואף על שמות הקבועים. המילים `integer`, `true` ו `false` אינן מילים שמורות בשפה, והמתכנת יכול להגדירן מחדש. בפרט, התכנית הבאה המדגימה את סיסמת המפלגה בספר 1984 לג'ורג' אורוול היא תכנית חוקית בפסקל:

```
program NineteenEightyFour;
const
  integer=true;
  true=false;
  false=integer;
begin
  writeln("Truth is ", true);
  writeln("False is ", false);
end.
```

מרבית הסיכויים הם שאם תזין תכנית זו במהדר הפסקל החביב עליך, הוא יפלוט בתגובה שגיאות הידור לרוב. התפתחות השפה הביאה לכך שהמילים `true` ו `false` הפכו בחלוף השנים למילים שמורות, וההבדל בין המילה `true` ובין הישות שהיא מציינת, הלא היא ערך האמת המופשט נמוג.

מנגד, האבחנה בין הסימון ובין הישות שהוא מייצג מתחזקת בשפות מודרניות, אשר בהן הסימן '+' למשל, נבדל מפעולת החיבור אותה הוא מייצג. כך ב C++ למשל, המתכנת יכול להשתמש בסימן זה גם לפעולות אחרות באמצעות תהליך הקרוי העמסת אופרטורים.

מנגד לגישה האוטרקית, קיימת **הגישה המטאפיזית** הנקוטה על ידי שפת C. על פי גישה זו, תיחום התכנית נעשה מבחוץ לה, באמצעות כלים אשר אינם מוגדרים בשפה. בגישה זו, אוסף של קטעי קוד המצויים (למשל) בקבצים שונים נאסף על ידי העוקד[18] לכדי תכנית אחת, אך השפה עצמה אינה מתייחסת להגדרת העוקד. העוקד גם מחבר לתכנית ספריה של שירותים. הספריה היא סטנדרטית, במובן זה, שלהגדרת השפה נלווית גם הגדרה של הספריה הסטנדרטית, אך שתי הגדרות אלו הן בלתי תלויות. ניתן לשנות ולפתח את הספריה הסטנדרטית מבלי לשנות את השפה ואת המהדר שלה. המתכנת אף הוא יכול לבחור להחליף חלקים מהספריה הסטנדרטית, או את התכנית כולה.

נעיין כעת בתכנית שלום עולם בשפת C:

[18] הלא הוא ה Linker

```
main() {
     printf("Hello, World!\n");
}
```

טרם שנמשיך, נדרשות כמה מילים כדי להפיס את דעתו של מי שיזעם על התכנית כפי שהיא כתובה כאן, משום שלכאורה חסרה בה ההוראה:

```
#include <stdio.h>
```

אשר כביכול גורמת לקישור של התכנית לספריה הסטנדרטית, אשר בה מוגדרת הפונקציה `printf`.

ובכן, לא מיניה ולא מקצתיה: התכנית תהודר ותקושר לספריה הסטנדרטית בין אם נכלול הוראה זו ובין אם לאו. ההוראה אמנם גורמת ליבוא קובץ הגדרות, אשר מסייע למהדר לבדוק את התכנית. אך יבוא זה נעשה טרם ההידור, והמהדר אינו מודע לו כלל וכלל, ועל אחת כמה וכמה העוקד.

העקידה לספריה הסטנדרטית נעשה באמצעות הפקודה cc אשר אינה מפעילה את המהדר בלבד, אלא גם את הקדם מעבד, כמו גם את העוקד. הפעלת העוקד מעבירה לו בקשה לעקוד את הקובץ המהודר עם הספריה הסטנדרטית. בהפעלת פקודה זו ניתן בקלות להשתמש בדגלים אשר יכתיבו עקידה עם ספריה אחרת, או אי ביצוע עקידה כלל.

לאחר דברי פיוס אלו נציין כי התכנית הזו משתמשת בשירותי הספריה הסטנדרטית בשתי דרכים.

- ראשית, ישנו השימוש הברור בפונקציה `printf` אשר לקוחה מהספריה. פונקציה זו כתובה בשפת C, והיא נגישה לכל דיכפין אשר יכול אף לשנות אותה אם יחפוץ בכך.
- התלות השניה בשירותי הספריה היא בקביעה כי נקודת תחילת הביצוע היא בפונקציה main. החלטה זו מוכתבת על ידי העוקד, אשר אוסף את כל קטעי התכנית, ובכללם שלושת השורות מעלה, הגדרת הפונקציה `printf` מהספריה, כמו גם את כל הפונקציות והמשתנים בהם משתמשת `printf`, ויצירת תכנית אחת מהם. אחרי האיסוף הזה, יש גם לקבוע את נקודת ההתחלה.

  העובדה שהעוקד בוחר להתחיל את הביצוע בפונקציה `main` היא מוסכמה, אולם, ישנם מקרים בהם היא אינה מקויימת. כך למשל תכניות הכתובות עבור מערכת ההפעלה חלונות יתחילו ב `WinMain`.

הסיבה העיקרית שבה לימוד תכנות בשפה חדשה לעולם יתחיל בכתיבת תכנית "שלום, עולם!" היא שתכנית זו כופה על המתכנת להשתמש (אם כי לא בהכרח מתוך הבנה מלא) במנגנונים לתחימת גבולות התכנית, ולקביעת תחילת הביצוע. וכפי שהדוגמאות כאן מורות, המנגנון הזה אינו תמיד מוסבר באופן מפורש, או שהביאור של אופן פעולתו הוא מייגע ובלתי נגיש למתכנת המתחיל.

השוואה בין השיטה האוטרקית לשיטת ספריה

	אוטרקית	מטאפיזית/ספריה
נקודת התחלה	שורה ראשונה או ממילה שמורה	מוגדרת מחוץ לתוכנית. בדרך כלל ישנם נהגים מקובלים, קונבנציות, (אולם הנוהג אינו בהכרח דין).
קלט/פלט **שגרות עזר וכיוצא בזה**	אחד משלושה: הווים חלק מהגדרת השפה אמצעות מזהים מוגדרים מראש לים שמורות	בקבצי ספריה למיניהם שעל המתכנת לחבר לתוכנית.
טיפוסים יסודיים	שפה לים שמורות לים מוגדרות מראש	בדרך כלל מילים שמורות
היקף התוכנית	תמיד יהיה בקובץ בודד	מוגדר מחוץ לשפה באמצעות הלינקר או כלי אחר. כלומר מוגדר גם מחוץ לתוכנית עצמה.