

**שפות תכנות**

**Scheme**: שפה אינטראקטיבית. צורת הכתיב היא prefix: קודם האופרטור, אח"כ האופרנדים. הסיבה: שמירה על אחידות:  $f(x, y, z)$  – קודם שם הפונקציה, ולכן גם  $(x,y,z)$  וכיו"ב.

יש אופרטורים שמקבלים מס' משתנה של אופרנדים, למשל:  $(+ 2 3)$  או  $(+ 2 3 4)$  [לעומת זאת, לא הגיוני לכתוב:  $(/ 4 3 6)$  שכן הפעולה אינה אסוצ'].

פונקציה יכולה להשתמש בפונקציה שאינה מוגדרת עוד, אולם בעת ההרצה (השאליתה) יש לדאוג שהפונקציה תהיה מוגדרת.

car – return the 1<sup>st</sup> item on the list (context of register a),

cdr – returns the list (context of register d), but the 1<sup>st</sup> item (in Prolog:  $[X | Xs] - x = \text{car}, Xs = \text{cdr}, | = \text{cons}$ ):

Every element and a pointer are in a register.

$(\text{cdr } '(3)) = ()$ . |  $(\text{cdr } 3) - \text{undefined}$ .

$(\text{cons } 1 '(2 3)) = (1 2 3)$  [the 2<sup>nd</sup> parameter of cons must be a list].

$\text{cons}(a b) = (a.b)$ .

$(\text{map } \text{cadr } '((a b) (d e) (g h))) = (b e h)$  [הפעלת האופרטור על הארגומנטים איבר-איבר, ובסוף הכנסת התוצאות לרשימה]

$(\text{cons } '(1 2) '(3 4 5)) = ((1 2) 3 4 5)$ , |  $(\text{cons } 1 '(2 3 4)) = (1 2 3 4)$ ,

$(\text{list } '(1 2) '(3 4 5)) = ((1 2) (3 4 5))$ , |  $(\text{list } 1 '(2 3 4)) = (1 (2 3 4))$

$(\text{append } '(1 2) '(3 4 5)) = (1 2 3 4 5)$ . |  $(\text{append } 1 '(2 3 4)) = \text{undefined}$ .

```
(define (factorial n)
  (if (= n 1) 1
      (* n (factorial (- n 1)))))
```

**לא רקורסיית זנב:**

בסוף מבצעים כפל, ולא את הקריאה הרקורסיבית.

הפתרון: שימוש באובר.

כינון פונקציות מאפשר לנו לא להעביר פרמטרים לכל הפונקציות – הפרמטרים של הפונקציה הראשית מוכרים בפונקציות המכוננות:

1. הכינון חוסך כתיבה של פרמטרים ← מונע בלבול.
2. הכינון מאפשר להתייחס לפונקציות, מבחינת הקורא, כפונקציות פנימיות, פונקציות עזר.

**למה ב-Scheme וב-Prolog הרקורסיה מרכזית?** ב-2 השפות הנ"ל אין מבני בקרה של for, while וכו'. הדרך היחידה לבצע לולאות היא ע"י רקורסיה.

פונקציה של רקורסיית זנב יעילה יותר מזו של פונקציה רקורסיבית אחרת.

**ב-Prolog וב-Scheme אין השמה.**

nil ב-Lisp הוא מצביע ריק. כשהומצאה השפה התייחסו ל-nil גם בתור רשימה ריקה, גם בתור מחרוזת ריקה וגם בתור #f (שקר). לכן, nil יכול להיות גם תוצאה של משהו שהיה צריך לעוף.

<pre>(cond (p1 e1)       (p2 e2)       ...       (pn en))</pre>	<p>אם p1 אמת התוצאה המוחזרת של ה-cond היא e1. אחרת עוברים ל-p2 וכן הלאה. בסוף, אם הכל שקר, מגיעים ל-pn שצריך להיות אמת. צריך לדאוג לכך ש-pn יהיה אמת, למשל ע"י כתיבת #t במקום pn. ניתן גם לכתוב else (המשמעות - אמת).</p>
---	---

**דוגמאות:**

<pre>(define (pi-sum a b) ; 1/1*3 + 1/5*7 + 1/9*11 + ...   (if (&gt; a b) 0       (+ (/ 1 (* a (+ a 2)))           (pi-sum (+ a 4) b))))  (define (sum term a next b)   (if (&gt; a b) 0       (+ (term a)           (sum term (next a) next b))))  (define (pi-sum a b)   (define (pi-term x) (/ 1 (* x (+ x 2))))   (define (pi-next x) (+ x 4))   (sum pi-term a pi-next b))</pre>	<pre>(define (pi-sum a b)   (sum (lambda(x) (/ 1 (* x (+ x 2))))        a        (lambda(x) (+ x 4))        b))  (define (expt x n)   (if (= n 0) 1       (* x (expt x (- n 1)))))  (define (make-expt n)   (lambda(x) (expt x n)))  &gt; (make-expt 2) = (CLOSURE (x) (expt x n)) = x<sup>2</sup> &gt; ((make-expt 3) 2) = 8</pre>
---	---

בד"כ, אם קוראים לפונקציה רק פעם אחת אזי משתמשים בפונקציה אנונימית –  $\lambda(x)$ .

ב-Scheme יש שימוש בפונקציות כפרמטרים.

### Prolog לעומת Scheme:

ב-Scheme הפונקציות מחזירות ערכים. ב-Prolog מוחזר רק אמת/שקר.  
ב-Prolog זה negation by failure – או אמת או שקר בלבד.

## 1. מבוא

### 1.1 סקירה היסטורית:

**Fortran** – השפה העילית הראשונה שהמציאו. המטרה: שהמהדר לשפה יהיה יעיל. Fortran יעילה לחישובים נומריים כבדים ומשתמשים בה הרבה למחשבים מקביליים כבדים.

**Legacy Code**: כשמוסיפים שינויים ותיקונים לשפה שיש לה מיליוני שורות קוד אי-אפשר להיפטר מהקוד ולכן פקודות ישנות לא נמחקות אלא נשארות בשפה.

**PL/1** (Programming Language – Ver. 1) – ניסיון לבנות שפה אחת גדולה ומקיפה, מתוך תקווה שכולם יכתבו בשפה זו.

**ברירת מחדל**: המושג החל ב-Fortran: לא צריך להצהיר על המשתנים והייתה ברירת מחדל: משתנה שהתחיל באותיות מסוימות היה integer, כל אות אחרת – float.  
ב-PL/1 הרחיבו את מושג ברירת המחדל מתוך מטרה שיהיה ניתן ללמוד את השפה מהר, כאשר המתכנת יודע דברים מסוימים והמהדר יוסיף דברים לפי המוסכמות.

**ALGOL** (Algorithmic Language): אין קלט/פלט. שפת ALGOL היא השפה הראשונה שזכתה להגדרה **תחבירית** פורמלית חד-משמעית (BNF).  
מדוע פותחה גם ALGOL '68?  
1. הגדרת שפה שבה ניתן לתת הגדרה **סמנטית** פורמלית.  
2. רצון לבנות שפה אחת גדולה שתכלול את כל מה שצריך.

**PASCAL**: המטרות בפיתוח השפה: 1. לבנות שפה קטנה וקומפקטית.  
2. שתהיה טובה למטרות חינוכיות – תכנות מבני.  
3. בניית single-pass compiler עבור השפה.  
**הבעיה**: כיוון שהשפה נהייתה פופולרית, הרבה חברות כתבו מהדרים לשפה ולא הוקפד על סטנדרט, וכך השפה נהפכה **ללא נידת**.

**SIMULA**: השפה ה-OO הראשונה. הייתה מיועדת לסימולציות.

**MODULA**: שונה מפסקל ומוסיפה מושגים חדשים, כמו: מודולים (יחידות הדומות למחלקה רק ללא הורשה).

**ADA**: הוצא copy-right על השם ADA ובכך כל מי שרוצה לעשות שינוי בשפה אינו יכול לבנות מהדר חדש לשפה ולקרוא לו מהדר ל-ADA בלי לקבל רשות. השפה הזו היא "פיל לבן". השפה היא אמינה מאוד.

**LISP** (List Processing): שפה שבעיקרה נועדה לעיבוד רשימות.

**SmallTalk**: שפה שנבנתה מלכתחילה להיות OO.

**C**: פותחה על-מנת לפתח מע' הפעלה בשפה עילית. פיתחו את Unix שהיא מע' הפעלה טובה ← כדאי לתכנת מע' הפעלה ב-C, אולם אנשים הסיקו: הואיל ו-Unix מע' הפעלה טובה שפותחה ב-C ← שפת C היא טובה לכל מטרה.

**JAVA**: רצו לפתח שפה אמינה למע' משובצות ולמיקרו מעבדים.

**1.2 הנדסת תוכנה**: פיתוח ותחזוקה של מע' תוכנה גדולות.

**2: הגדרה**: הנדסת תוכנה היא הגישה השיטתית לפיתוח, הפעלה, תחזוקה ופרישה של מערכות תוכנה (גדולות).

**תחזוקה**: 1. תיקון באגים. 2. התאמת התוכנה לשינויים בעולם.

הקריטריון החשוב ביותר בהערכת שפות תכנות הוא האם יהיה קל לקורא הקוד להבין אותו: התוכניות נכתבות עבור בנ"א ולא עבור מחשבים.

### 1.3 מחשבים וזמן קישור:

#### הגדרה 1:

1. מחשב הוא אוסף של כללים ומבני נתונים המאפשרים אחסון וביצוע תוכניות בשפה מסוימת.
2. זמן קישור של מושג בשפת תכנות הוא הזמן, תוך כדי תכנון או מימוש השפה, או תוך כדי ניסוח או עיבוד תכנית בשפה, בו נקבעים (נקשרים) המאפיינים והתכונות של המושג הנדון.

מחשב בחומרה שייך להגדרת המחשב. האם מחשב בשפת C יכול להיות מחשב בחומרה? כן, ניתן לבנות מחשב לשפת C בחומרה, אולם לא בונים מחשבים בחומרה לשפה X שכן יש הרבה שפות ומשתמשים בכמה שפות במקביל. בעצם קונים מחשב בחומרה שעליו מלבישים מחשב ב-C, מחשב בפרולוג וכו', שהם מחשבים בתוכנה: מהדר ומפענח.

1. התוכנית עוברת תרגום לשפת המכונה.
2. ביצוע שפת המכונה.

מפענח: מקבל את התוכנית, לוקח את הפקודה הראשונה ומתרגם אותה. אח"כ עובר לפקודה ה-2, מתרגם וישר מבצע אותה וכן הלאה.

דוגמאות:

1. `for (int i = 0; i < 1,000; ++i) { S }`  
- המהדר יתרגם את S פעם אחת ויבצע את S 1,000 פעם.  
- המפענח יתרגם את S כל איטרציה ויבצע, כלומר יהיה תרגום וביצוע 1,000 פעם.
2. השמה של מחרוזת ל-S1: `S1 = ' I = 0; I = ' ; //`  
`S2 = ' I + 5; ' ;`  
`S3 = S1 + S2; //` שרשר  
`Execute(S3); //` ביצוע מחרוזת S - ביצוע הפקודות

- מהדר: S1, S2 יקבלו ערך רק בזמן הביצוע, לכן לא ידוע מה יש ב-S3. בזמן התרגום המהדר לא יודע איך לתרגם את `Execute(S3)`. בזמן הביצוע, אחרי שהתרגום נגמר, ב-S1 וב-S2 יש מחרוזות, ועכשיו ניתן לדעת מה יש ב-S3, אולם לא ניתן לבצע עכשיו `Execute(S3)` שכן צריך יהיה לתרגם את המחרוזת לשפת מכונה, אולם שלב התרגום נגמר.
  - מפענח: יתרגם את הפקודה ה-1 ויבצע; יתרגם את הפקודה ה-2 ויבצע; משרשר את S3 (תרגום + ביצוע) ל-S3 יש תוכן. ע"י שימוש במפענח ניתן לבצע את הקוד הנ"ל.
- מסקנה: בעת המצאת שפה צריך לדעת האם מהדר יכול לתרגם את השפה (שפת קומפילציה) או שהשפה תהיה שפת פענוח.
- אם קומפילטר יכול לעבוד על תוכנית ← מפענח גם יכול, אולם הצד השני אינו נכון (זה שמפענח יכול לא גורר שמהדר יכול).

Prolog ו-Scheme שפות פענוח (ב-Scheme ניתן לשלוח פונקציה בתור ארגומנט, בפרולוג יש את `univ`). כיצד נהפוך אותן לשפות קומפילציה?

1. ניקח את הגדרת השפה ונמחק את שדורש רק מפענח. כמו כן צריך לדרוש שתהיה הצהרה על סוגי המשתנים.
2. נעשה 2 שלבים: א. הידור – עד כמה שאפשר (מצב ביניים). ב. ממצב הביניים נעשה פענוח וביצוע.

Java היא שפת קומפילציה – ניתן לתרגם אותה מההתחלה ועד הסוף. עושים לפי השיטה ה-2 (תרגום למהדר ביניים ופענוח). הסיבות לכך:

1. משתמשים ב-Java הרבה לאינטרנט. נעשית קומפילציה של התוכנית ומביאים רק את הפונקציות הנחוצות ולא את כל ה-`packages` ← יעילות בזמן מאשר לו היו עושים קומפילציה ישירה והיינו מעבירים את כל המידע למחשב.
2. בטיחות: לשפת Java יש אמצעי בטיחות `built-in`. מפענח יכול לבצע במהלך הפענוח בדיקות בטיחות.

Debugger חייב להיות מפענח, שכן ב-`debugging` מתרגמים ומבצעים ויש אפשרות לעצור אותו בפקודה מסוימת ולבחון את הערכים הנתונים באותו שלב ← ניתן לעצור בכל מקום ולעבור לפקודה אחרת וממנה להמשיך הלאה.

**זמן קישור:**

4 זמני קישור:

1. **זמן הגדרת השפה.**

2. **זמן מימוש השפה.**

3. **זמן תרגום.**

4. **זמן ביצוע:** 1. בכניסה לבלוק; 2 בתוך הבלוק.

I.  $y = x + 5$ ; מתי  $x$  מקבל ערך (מתי נקשר ערך ל- $x$ )?

1. אם בשפה הגדרנו קבועים מראש, למשל PI, אזי הערך עבור PI נקבע בזמן הגדרת השפה.

2. אם מי שמממש את השפה, כלומר כותב את הקומפיילר לשפה, נותן את הערך למשתנה PI – למשל, מגדיר אותו להיות 3.14  $\leftarrow$  **זמן מימוש השפה.**

3. אם היה כתוב: `const int x = 4;` או `static int x = 4;`  $\leftarrow$  מתבצע בזמן התרגום.

4. אם לפני הפקודה, באמצע התוכנית, היה `x = 4;` אזי **בזמן הביצוע** נכנס ל- $x$ .

אם היה `{ int x = 4; ... }` הדבר מתבצע בכניסה לבלוק,  $x$  כאן הוא **משתנה אוטומטי**. רגע לפני הכניסה לבלוק אין משתנה  $x$ .

**ההבחנה בין זמן תרגום לזמן ביצוע נכונה רק לשפות קומפילציה. בשפות פענוח הכל נעשה בזמן הביצוע!**

**II. מתי נקשר טיפוס ל- $x$ ?**

1. **בזמן הגדרת השפה** מגדירים את טיפוס המשתנים (למשל: int).

2. **בזמן מימוש השפה** כותב המהדר מחליט מה זה בעצם int (למשל: int ש-ישוב ב-4 בתים).

3. **בזמן התרגום:** `const int x = 4;`

4. אם הטיפוס נקשר **בזמן הביצוע** ניתן שתהיה שפה שתעשה:  $a \leftarrow 3.9 \dots a \leftarrow 2$ : כאשר בהתחלה בתא ה-descriptor של  $a$  רשום int ואח"כ זה משתנה ל-float, כלומר: בכל רגע נתון ניתן לקשור טיפוס למשתנים. ניתן לעשות את זה ב-APL אבל לא ב-C, שכן APL היא **שפת פענוח** והכל נעשה בזמן הביצוע  $\leftarrow$  גמישות (החסרון: התוכנית לא קריאה), ואילו C היא שפת קומפילציה, ולכן בזמן התרגום נקבע סוג הטיפוס.

**ככל שקושרים את הדברים יותר מוקדם  $\leftarrow$  יותר אמין ויעיל:**

**יעיל:** כיוון שזמן הקומפילציה לא נחשב, ועל מה שכבר נקשר לא מבזבזים זמן.

**אמין:** יש גילוי שגיאות בזמן הקומפילציה.

**III. מתי נקשרות התכונות של +?**

1. **בזמן הגדרת השפה** מגדירים שבמסגרת השפה יהיו פעולות אריתמטיות ומגדירים את הסימון שלהן.

2. **בזמן מימוש השפה** מחליטים מה זה +: התרגום לשפת מכונה (add).

3. **בזמן התרגום:**

`output = 2 + 3 * 4 // output = 14`

`priority + = 5, * = 4;`

`output = 2 + 3 * 4 // output = 20`

ב-ALGOL, שהיא שפת קומפילציה, המהדר מתרגם את הפקודה לשפת מכונה. כשהוא מגיע ל-priority הוא **משנה** את סדר העדיפויות של הפעולות  $\leftarrow$  שינוי הפעולות של חיבור וכפל **בזמן התרגום** (קשירת תכונות אחרות לחיבור ולכפל).

**overloading – תוספת**, זה לא מבטל הגדרה קודמת (מה שנקשר בזמן ההגדרה והמימוש נשמר). overloading זה דוגמה לקשירת תכונות "+" בזמן התרגום.

4. **בזמן הביצוע:** ננסה להסתכל על שפת פענוח, שכן בשפת פענוח הכל נקשר בזמן הביצוע:

ב-Snobol, שהיא שפת פענוח, ניתן לשנות את ההגדרות ה-built-in (שזה לא כמו overloading שזה תוספת להגדרות ה-built-in):

`output = 2 + 3 * 4 // 14`

`opsyn('+', '*', 2); // every location where there is + we do *`

`output = 2 + 3 * 4 // 24`

יש כאן שינוי של תכונת + בעת הביצוע.

**1.4 פרדיגמות של תכנות:**

1. **שפות ציוניות** (C, C++, Java, Ada, Cobol, Pascal, Fortran)

2. **שפות הצהרתיות:** א. שפות לוגיות (Prolog). ב. שפות פונקציונליות (Scheme).

לשפות הצהרתיות יש 2 אספקטים:

1. אספקט הצהרתי טהור (Scheme אינה שפה הצהרתית טהורה).

2. אספקט פרוצדורלי שדומה לציווי, למשל: בקשת שאילתה בפרולוג.

בשפות הצהרתיות קל להוסיף ולעדכן נתונים.

שפות OO ניצבות ל-2 הפרדיגמות הנ"ל:  
 C - ציווית ולא OO.  
 C++ - ציווית ו-OO.  
 Prolog, Scheme – הצהרתיות ואינן OO.  
 Prolog++, OO-LISP, Haskell - הצהרתיות ו-OO.

**הגדרה 3:**

1. פונקציה נקראת שקופה התייחסותית אם היא תמיד מתנהגת באותה צורה כאשר קוראים לה עם אותם ארגומנטים.
2. מע' נקראת שקופה התייחסותית אם כל אחד מחלקיה שקוף התייחסותית.
3. משפט: מע' שקופה התייחסותית אם"ם המובן של המע' כולה נגזר מהמובן של כל אחד מחלקיה בנפרד.

דוגמה:

```
int swich = 0;
int f(...) { ... swich = 1; ... }
int g(int n) {
    if (swich) return 4*n;
    else return 3 * n; }
main() {
    g(2); f(); g(2); }
```

הפונקציה g אינה שקופה התייחסותית: הערך של swich גורם לקריאה של g(2) לתת ערכים שונים.

מדוע חשוב שהפונקציה תהיה שקופה התייחסותית? לתחזוקה: אם מתחזק יודע שהדברים שקופים התייחסותית והוא הבין את הפונקציה, אזי הוא לא צריך להבין עוד פעם את הפונקציה עבור ערכים שונים. שקיפות התייחסותית תורמת להבנת התוכנית.

**הגדרה 11:** עקרון המבניות: התמונה הדינמית של החישובים צריכה להתאים בצורה פשוטה עד כמה שאפשר למבנה הסטטי של התוכנית המתאימה.

**התמונה הסטטית של התוכנית = הקוד.**

**התמונה הדינמית של התוכנית = איך שהתוכנית רצה.**

**עקרון המבניות:** הבנה של מה שקורה בתוכנית זו הבנה של התמונה הדינמית. עמ"נ להבין את התמונה הדינמית יש את התמונה הסטטית וככל שהם יותר דומים קל יותר להבין.

**שקיפות התייחסותית תורמת לעקרון המבניות!**

בשפה מסוימת יכולים להיות תכונות שתורמות לעקרון המבניות יחד עם תכונות שלא תורמות. למשל, ב-C, ובכל שפה בה יש משתנים גלובליים, אין שקיפות התייחסותית. בשפות הצהרתיות אין משתנים גלובליים ← יש שקיפות התייחסותית ומהאספקט הזה הן מקיימות את עקרון המבניות. בפונקציה שמקבלת קלט מהמשתמש אין שקיפות התייחסותית.

בשפות ציוויות הפקודה הנפוצה היא =. בשפות הצהרתיות אין השמה כלל.

בפרולוג: X is X + 2 זו לא השמה ל-X אלא instantiation ל-X.

מה רע בהשמה? עקרון המבניות: בשביל להבין את הפקודה  $a = b + c$ ; צריך להבין מה הערך של b ושל c. כדי לדעת מה הערך של b צריך לחזור להיסטוריה של b. אם אין השמה ← אין היסטוריה. ולכן, אם אין השמה זה תורם לעקרון המבניות.

אם מע' תוכנה מחולקת לרכיבים, כאשר כל רכיב הוא שקוף התייחסותי, אזי אין בעיה להריץ כל רכיב ב-CPU אחר ובכך ליצור מקביליות ← הרבה יותר קל לנסח דברים מבוזרים או מקביליים בשפות הצהרתיות מאשר בשפות ציוויות (למרות שניתן גם בשפות ציוויות, כמו למשל Threads ב-Java).

טענה: התפוקה של כל מתכנת היא קבועה, לא משנה באיזה שפה הוא כותב.

אף אחד לא הוכיח את הטענה. אם מקבלים את הטענה, אזי כדאי להמציא שפה מאוד מתומצתת ואז אותו מס' שורות קוד יתרום בסופו של דבר יותר קוד. כיוון שהשפות ההצהרתיות יותר תמציתיות מהשפות הציוויות, אזי, לפי הטענה, כדי לתכנת בהן.

למה בתעשייה לא משתמשים בשפות הצהרתיות?

1. מבחינה היסטורית, השפות הראשונות היו ציוויות והמשיכו איתן.
2. הגישה ההצהרתית ברורה יותר מהגישה הציווית. אולם, בחלק מהשפות ההצהרתיות, הבעייתיות היא איך שהן עובדות – האספקט הפרוצדורלי.

## 2. צורת התוכנית:

### 2.1 פורמט כללי:

התוכנית הראשונה היו סדרה ארוכה של פקודות. לאחר מכן התווספו פונקציות ופרוצדורות. הסיבה העיקרית להוספתן הייתה: **חסכון בזיכרון**. הסיבות היום לשימוש בפונקציות ובפרוצדורות:

1. הפשטה: פונקציות כלליות (למשל: פונקציה למיון וקטור מכל סוג).
2. כותבים תוכניות בשביל בנ"א: אף בן אדם לא כותב תוכנית גדולה לבד. אם התוכנית נכתבת כמקשה אחת, אזי קשה לחלק את העבודה לצוות. אם התוכנית מחולקת לפונקציות ← באופן טבעי יש חלוקה של המשימות.

יש אריזה של כמה פונקציות יחד ליחידה גדולה יותר – מודולים. האריזה למודולים היא לפי פונקציות שיש ביניהן קשר. בנוסף, המודולים מאפשרים את עקרון הסתרת המידע. ניתן לקבץ כמה מודולים למודול ראשי ולאפשר ירושה (classes).

החלוקה של פונקציות, מודולים וירושה מתאימה הן לפרדיגימה הציווית והן לפרדיגימה ההצהרתית.

### 2.2 פורמט של פקודה:

בהתחלה הפקודה הייתה קשורה למצב הפיסי, לפי כרטיס הניקוב. אח"כ שונה מבנה כתיבת הפקודה כך שיהיה ברור לבנ"א.

### 2.3 הערות ומלים שמורות:

צירוף תווים התחלתי וצירוף תווים סופי מהווה הערה. אח"כ הוסיפו סימון מיוחד להערה שנמשכת עד סוף השורה. ברוב השפות אסור לעשות כינון של הערות: הקומפיילר לא מאפשר את זה עקב הרצון שהקומפיילר יהיה יעיל (חסכון בזמן).

**מלים שמורות:** טוב עבור בנ"א – מאפשר להתמצא בתוכנית. PL/1 אין מלים שמורות – נוח ללמוד את השפה מהר.

צריך לדעת לבחור את המלים השמורות.

דוגמאות לא טובות לבחירה:

1. virtual ב-C++: קשורה ל-2 מושגים: פונקציות (פולימורפיזם), ירושה.
2. static:

- א. משתנה סטטי בתוך פונקציה – זמן הקישור של המקום בזיכרון של משתנה כזה הוא זמן הקומפילציה והוא נשאר חי וקיים גם ברגע שאתה יוצא מהפונקציה.
  - ב. משתנה סטטי בתוך class – משתנה מחלקה ולא משתנה מופע (ערך אחד עבור כל המופעים).
  - ג. משתנה סטטי גלובלי – המשתנה מוכר רק באותו קובץ.
  - ד. פונקציה סטטית.
3. &: א. אופרטור and לוגי; ב. כתובת בזיכרון.

### 2.4 הגדרה לעומת הצהרה:

INTEGER : X – גם הגדרה וגם הצהרה.

ב-C יש הבחנה בין הצהרה ובין הגדרה: extern x; - הצהרה של x. int x - הגדרה של x.

### 2.5 שפה מכוונת פקודות ושפה מכוונת ביטויים:

C היא שפה מכוונת ביטויים: לכל פקודה יש ערך: ניתן להסתכל על פקודה כעל פקודה או כעל ביטוי. דוגמה:

cin - כפקודה: קריאת קלט; כביטוי: בעל ערך T/F.  
i++ - כפקודה: הוספת ערך אחד ל-i; כביטוי: הערך של i.  
while (cin >> s[i++]);

Ada היא שפה מכוונת פקודות.

**עקרון המבניות** נתמך טוב יותר בשפה מכוונת פקודות מאשר בשפה מכוונת ביטויים.

## 3. אופרטורים ופעולות חשבון:

### 3.1 סדר ביצוע הפעולות:

APL: יש הרבה אופרטורים בשפה. כאשר כותבים פקודה המפענח עובר על הפקודה **מימין לשמאל** ומה שהוא רואה הוא ישר מבצע. דוגמה:  $a = b - (c - d) // a <- b - c - d$ . עקרון המבניות לא מתקיים כאן, שכן התמונה הסטטית היא  $a = b - c - d$ , ואילו החישוב הוא  $a = b - c + d$ .

Lisp כתוב ב-prefix. האם *prefix* נוגד את *עקרון המבניות?* לא, שכן מתרגלים לכתוב ולראות כתיב prefix. הסוגריים בביטוי ב-Lisp נחוצים כדי שהמפענח ידע מה מדובר, שכן יש אופרטורים שיכולים לפעול על כמה אופרנדים.

יש שפות המשתמשות בטבלה של סדר עדיפויות של אופרטורים ב-Pascal האופרטורים and, or קודמים ל-, <, >. ואילו ב-Ada זה הפוך. הפקודה  $x > y$  and  $z < 4$  לתרגם ב-Ada ל- $(x > y)$  and  $(z < 4)$  ואילו ב-Pascal היא תתרגם ל- $x > (y \text{ and } z) < 4$  והפקודה תיכשל ב-Pascal. אין בעיה ב-Pascal שכן המטרה היא שישימו סוגריים, וכך ניתן לשלוט בסדר העדיפויות, כלומר ב-Pascal נכתוב ישר:  $(x > y)$  and  $(z < 4)$  ← תמונה דינמית תהיה ברורה לבנ"א מעצם הקריאה של התמונה הסטטית.

**3.2 סדר חישוב האופרנדים:**

$(exp1) \text{ op } (exp2)$  – מי מחושב קודם:  $exp1$  או  $exp2$ ?  
למשל:  $n = 1; m = n / ++n;$  האם  $m$  יהיה שווה ל-1 או ל-1/2 בסוף?  
**זה לא מוגדר בשפה.** כל מי שכותב מהדר מגדיר כיצד זה יהיה. הבעיה נובעת מזה ששפת C היא שפה מכוונת ביטויים, שכן ניתן לכתוב בביטוי פקודות. בשפות מכוונות פקודות לא ניתן לכתוב פקודה בתוך פקודה נוספת ולכן בהן אין בעיה כזו.  
 $x = y = z = -1; u = ++x \ \&\& \ ++y \ \&\& \ ++z;$  ביטויים לוגיים מחושבים משמאל לימין ורק עד איפה שידועים מה התוצאה, לכן הערכים הסופיים יהיו:  $x = 0, y = z = -1$ . כאן יש הגדרה הגיונית וחד-משמעית שלגבי אופרטורים לוגיים החישוב מתבצע משמאל לימין ונפסיק ברגע שידועים את התוצאה.  
ב-Ada: אם רוצים שהתמונה הסטטית תישאר דומה לתמונה הדינמית משתמשים ב-and/or then. אם לא משנה סדר החישוב וניתן לאפשר למהדר לשנות את סדר החישוב (למשל, עבור אופטימיזציה) משתמשים רק ב-and/or-ב.

**3.3 שינוי הגדרה – Overloading:**

```
10  output = 4 * 3; (1)
    opsyn('%', '*', 2); // % is equivalent to multiplication.
    opsyn('*', F, 2); // * is equivalent to F which is subtraction.
    F = x - y;
    output = 4 * 3; (2)
    goto 10
```

עבור **שפת פענוח** (כמו Snobol):  
(1) הפלט יהיה בפעם הראשונה 12 ומהפעם השנייה ואילך 1.  
(2) הפלט יהיה 1 (4 - 3).

לעומת זאת **בשפת קומפילציה** (כמו ALGOL'68)

```
10  output = 2 * 3 + 4; (1)
    priority * = 13, + = 14;
    output = 2 * 3 + 4; (2)
    goto 10
```

(1) תמיד יודפס 10.  
(2) יודפס 14.

שכן בשלב הקומפילציה יש תרגום של הפקודה, והתרגום לא משתנה בעת הביצוע (תמיד התרגום של שורה 10 יהיה לפני שינוי העדיפות).

**overloading לא משנה את ההגדרה, הוא מרחיב את ההגדרה שהיא built-in.**

האם **overloading** נוגד לעקרון המבניות?  $a + b$ : התמונה הסטטית לכאורה אומרת שמדובר בחיבור, אולם הכוונה היא להשתמש בהגדרה המורחבת, כלומר: התמונה הדינמית כן שונה.  
הפתרון: תלוי איך מרחיבים את ההגדרה. למשל, חיבור מטריצות, שרשור מחרוזות, חיבור מס' מרוכבים – כאן ה-overloading מקל על עקרון המבניות. לעומת זאת, 4 פונקציות הדפסה ל-4 טיפוסים שונים – כאן לא ניתן להבין את התמונה הדינמית מהתמונה הסטטית.

**3.4 השמה מרובה:**

כל הביטויים בשורה השנייה צריכים לקבל את אותו ערך  $(I + 1)$ .  
ההשמה הזו מוגדרת באופן חוד-משמעי, אבל התמונה הדינמית לא מובנת מהתמונה הסטטית.  
PL/1:  $I = 1;$   
 $a(I), I, a(I + 1) = I + 1$

ב-PL/1 ההשמה ואופרטור השוואה הם אותו אופרטור:  $A = B = C = 5$ . בעצם רשום:  $A = [B = C = 5]$ . כיוון ש-PL/1 היא שפה מכוונת פקודות אזי רק ה- $=$  הראשון הוא השמה וכל השאר זה השוואה. בסופו של דבר A יהיה אמת/שקר.

**4. מבני בקרה:**

**4.1 פקודות התניה:**

**בעיית ה-dangling else:** לאן ה-else מצטרף – צריך להגדיר את זה בשפה. C- הכלל אומר שה-else מצטרף ל-if הפתוח הקרוב ביותר.

switch (expr) – expr יכול להיות ב-C מכל סוג דיסקרטי (בדיד), שכן ה-switch בא להחליט בין מס' דיסקרטי של פקודות ולכן סוג הטיפוס הוא דיסקרטי.  
 case e1 – תוכן e1 תלוי בשפה: ב-C, e1 יכול לקבל רק ערך אחד.  
 default: ב-C אם לא כתובים default והערך של ה-case שונה משאר הערכים, אזי התוכנית ממשיכה הלאה ולא עפה. ברגע שיש default תמיד הערך של ה-case נתפס.

נניח שה-switch חייב לעבור על כל הערכים האפשריים ללא default. האם הקומפיילר יכול לבדוק את זה?  
 - אם זה טיפוס דיסקרטי אזי יש מס' סופי של אפשרויות.  
 - אם הערכים ..., e1 נקבעים בזמן הביצוע המהדר לא יכול לבדוק את זה.  
 ב-C הערכים של ei הם קבועים ולכן המהדר יכול לבדוק.  
 ב-C המהדר אינו בודק שכל הערכים כוסו, כי מבחינתו זה שקוף – ממשיכים הלאה. ב-Ada המהדר כן בודק.

**הגדרה 4:** אורתוגונליות בשפת תכנות היא היכולת להרכיב מושג או תכונה בסיסית אחת של השפה עם מושג או תכונה בסיסית אחרת של השפה.

למשל: ALGOL'68 היא שפה אורתוגונלית בצורה מלאה. למשל, ניתן לבנות מערך של פונקציות ופונקציות של מערכים.

**4.2 לולאות ורקורסיה:**

fact(0, 1).  
 fact(N, F) :- M is N - 1, fact(M, G), F is N \* G.

**ברקורסיה רגילה:**  
 פעולות O(n)

**מקום:** הקצאה: O(n).  
 זמן להקצאה: O(n).

fact(N, F) :- fact(N, 1, F).  
 fact(0, G, G).  
 fact(N, M, F) :- N1 is N - 1, M1 is N \* M, fact(N1, M1, F).

**רקורסית זנב:**  
 פעולות O(n)  
**מקום:** הקצאה: O(1).  
 זמן להקצאה: O(1).

כל רקורסיה ניתנת לכתיבה כרקורסית זנב!

בשפות ציוויות הרקורסיות **לא יעילות** (גם לא רקורסיות זנב – המהדר לא מתייחס בכלל לסוג הרקורסיה). אין הרבה רקורסיות בשפות ציוויות כי יש מבני בקרה אחרים (לולאות for וכו'). בשפות ההצהרתיות אין לולאות ולכן צריך רקורסיות.

**טרנספורמציות של תוכניות:**

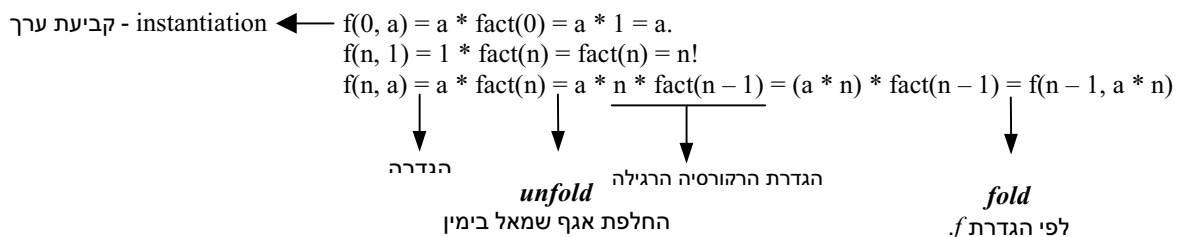
**fold / unfold:**

**הגדרה 5:**

- פעולת פריסה (unfold) על פונקציה מחליפה קריאה לפונקציה בגוף הפונקציה.
- פעולת כיפול (fold) על פונקציה מחליפה גוף של הפונקציה בקריאה לפונקציה.

**דוגמה 1:** הפיכת fact רקורסיבי רגיל לרקורסית זנב:

ההגדרה הרקורסיבית הרגילה: fact(n) = n \* fact(n - 1) = n!  
 נגדיר f(n, a) = a \* fact(n). כעת:





וקיבלנו הגדרה אחרת לפונקציה f:

$$f(n, a) = f(n - 1, a * n)$$

$$f(0, a) = a$$

$$\underline{f(n, 1) = fact(n)}$$

a הוא בעצם צובר.

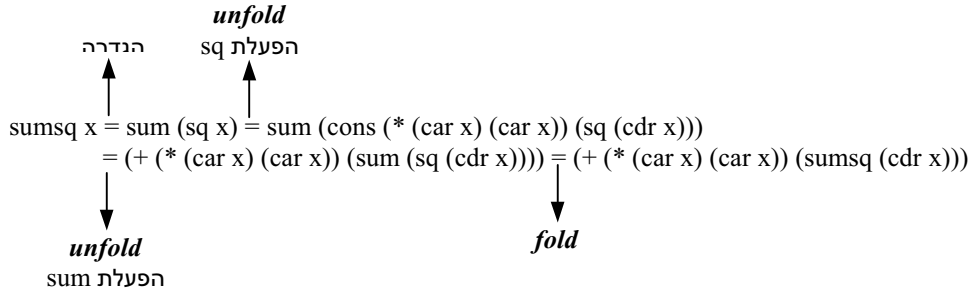
דוגמה 2:

<pre>(define (sum x)   (cond ((null? x) 0)         (#t (+ (car x) (sum (cdr x))))))</pre>	<pre>(define (sq x)   (cond ((null? x) ())         (#t (cons (* (car x) (car x)) (sq (cdr x))))))</pre>
---	---

```
(define (sumsq x) (sum (sq x)))
```

הפונקציה מחשבת סכום ריבועי איברי הרשימה. אולם ההגדרה אינה יעילה: הפונקציה sumsq עוברת פעמיים על הרשימה: פעם ראשונה בשביל להעלות את כל האיברים בריבוע, פעם שנייה בשביל לחבר את כל האיברים. **טרנספורמציה fold/unfold:**

$$(sumsq ()) = (sum (sq ())) = (sum ()) = 0 // instantiation$$



קיבלנו:

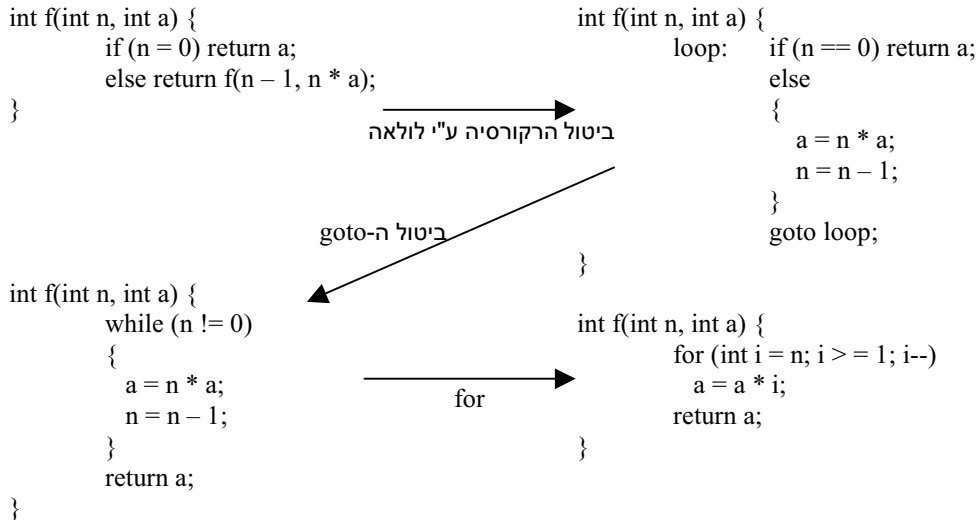
$$(sumsq ()) = 0$$

$$(sumsq x) = (+ (* (car x) (car x)) (sumsq (cdr x)))$$

הרקורסיה עדיין רגילה, אבל יעילה יותר: עוברים על הרשימה רק פעם אחת.

הפיכת רקורסית זנב ללולאת for:

דוגמה:



האם משתנה הלולאה מוגדר לאחר הלולאה?

ב-Fortran זה לא מוגדר ← מי שכותב את המהדר מחליט, ולכן כדאי להגדיר משתנה הלולאה מוגדר רק בבוק של הלולאה.

ב-Pascal משתנה הלולאה לא מוגדר לאחר הלולאה.

ב-Pascal ניתן לשנות את משתנה הלולאה רק בערך אחד כל איטרציה (למעלה או למטה). הסיבה: תיעוד עצמי: כדי שהקורא יידע בדיוק כמה פעמים הלולאה מתבצעת.

ב-C ניתן לשנות את משתנה הלולאה בגוף הלולאה. ב-Pascal זה אסור.

**הגדרה 6:**

1. יחידת תוכנית היא חלק של תוכנית המאפשר הצהרות.
2. בלוק הוא יחידת תוכנית שמופעלת ע"י זרימה לתוכה.
3. פרוצדורה היא יחידת תוכנית שמופעלת ע"י קריאה מפורשת. היחידה נקראת פונקציה אם היא מחזירה ערך.

ב-C++ בלוק, קובץ, פונקציה ומחלקה הם יחידות תוכנית.

**הגדרה 7:**

1. מודול הוא יחידת תוכנית שמאגדת נתונים, מבני נתונים, פרוצדורות ופונקציות.
2. אריזת נתונים (data encapsulation) היא ריכוז מבנה נתונים ורוטינות הגישה שלו במודול אחד.
3. עקרון הסתרת המידע: יש לתכנן מודולים כך ש:
  - א. למשתמש יש כל המידע הדרוש כדי להשתמש במודול בצורה נכונה – ולא יותר (ז"א אין למשתמש מידע על אופן מימוש המודול).
  - ב. לממש של המודול יש כל המידע הדרוש כדי לממש את המודול – ולא יותר (ז"א אין הוא מנצל מידע על שימושים ספציפיים של המודול).

רוטינות גישה הן רוטינות שמאפשרות לשים/להוציא ממבני הנתונים.

DO X = 0.0 to 1.0 by 0.01 – התמונה הסטטית מטעה שכן עקב שגיאות עיגול, יכול להיות שהלולאה תתבצע פעם אחת יותר/פחות. מה שהיה צריך לעשות:

```
DO I = 0 TO 100
  X = A + H * I
  H = (B - A) / 100 // where: B = 1.0, A = 0.0
```

כאן יש שגיאות עיגול, אבל התמונה הסטטית מראה בדיוק את התמונה הדינמית בלולאה. שגיאת העיגול אינה מצטברת.

```
I = 5;
DO I = I + 1 TO I + 7 BY I - 3
  I = I + 1 // values of I in the loop: 6, 6+1; 9, 9+1; 12, 12+1
END;
```

המהדר מכיל תא עבור הגבול התחתון, הגבול העליון והקפיצה של הלולאה ובעת הביצוע מכניסים להם ערכים והם **לא משתנים**, כלומר הגבול העליון בדוגמה הנ"ל הוא תמיד 12, הגבול התחתון: 6 והקפיצה היא ביחידות של 2.

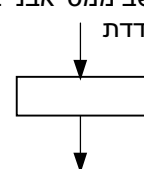
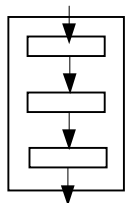
```
for I:= 3, 7, 11 step 1 until 16, i/2 while I>=1, 2 step I until 32 do print(I)
for I = 3 print (I)          for I = 7 print (I)          for (I = 11; I < 16; I++) print (I)
while (i/2 >= 1) print(I) // I starts from 16 (the previous loop)
for (I = 2; I < 32; I = I + I) print(I)
```

לא ניתן להבין דבר מהתמונה הסטטית!

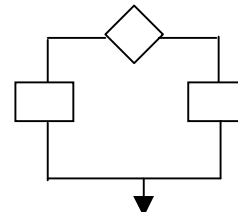
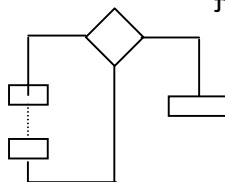
**4.3 מחלוקת ה-goto:**

לפי דיקסטר, בתוכניות שיש בהן הרבה goto קשות יותר ל-debugging. לכן הוא פיתח **תכנות מבני**: יש לבנות תוכניות מחשב ממס' אבני בנייה בסיסיות:

1. פקודה בודדת
2. סדרה של פקודות בודדות (המתבצעות באופן סדרתי): יש נק' כניסה אחת ונק' יציאה אחת.



3. פקודת if: נק' כניסה אחת ונק' יציאה אחת
4. לולאות



לפי דיקסטר, מי שמשמש רק באבני הבנייה הללו הוא מתכנת בצורה מבנית. מה שהורס את התכנות המבני הוא goto. ה-goto הורס את עקרון המבניות.

**טענה:** יותר קל להבין תוכנית הבנויה מ-4 אבני הבנייה הנ"ל.

בשביל מה צריך goto?

ב-Fortran היה ניתן לבצע פקודה בודדת אחרי if שמתקיים. אם רוצים לבצע הרבה פקודות, אז נכתוב בתוך הפקודה הבודדת goto ושם נבצע את שאר הפקודות.

**break** אינו כמו goto. הוא מאפשר לצאת רק מהמבנה בו אתה נמצא ולא לשום מקום אחר – ניתן להסתכל עליו כמעין goto מבוקר.

**exception** זה בעצם goto, אולם בניגוד ל-goto שמוגבל לאותה יחידת תוכנית, ה-throw מוציא אותך מהפונקציה ולא ברור לאן אתה מגיע – צריך למצוא את ה-catch המתאים לו. מתוך התמונה ה-הסטטית לא ניתן לעקוב אחר ה-throw. מתוך התמונה ה-דינמית כן ניתן לעקוב אחריו, ולכן ה-**exeception נוגד לעקרון המבניות**.

[exception לא נועד לטפל רק בבאגים, הוא נועד גם לטפל במקרים נדירים של קלט]

ב-Java אין goto.

**הגדרה 12:**

1. סכימת תוכנית היא סדרה סופית של פקודות, המתחילה עם  $START(x, y, \dots)$  ומסתיימת עם  $HALT(z, u, \dots)$ . כאן  $x, y, \dots$  הם ערכי קלט, ו- $z, u, \dots$  הם ערכי פלט. הפקודות האחרון הן:

- א. פקודה ריקה: null.
  - ב. פקודת השמה:  $x = term$ .
  - ג. פקודת התנייה: if P then S else T, כאשר S, T הן פקודות.
  - ד. פקודה מורכבת: {S, T, ...}
  - ה. פקודת הסתעפות: goto L, כאשר L היא תווית של פקודה.
2. סכימת While היא סכימת תוכנית שמשמשת בפקודות מהסוגים א'-ד' (אך לא ה') וכן בפקודות מהסוג:
- ו. פקודת while: while P do S, שמשמעותה: L: if P then {S; goto L} else null.

**הגדרה 13:**

- 1. אינטרפרטציה I של סכימת תוכנית מורכבת מתחום  $D_I$  ממנו משתני הסכימה יכולים לקבל ערכים, ומפרט הפונקציות והפרדיקטים מעל  $D_I$  המתאימים לפונקציות ולפרדיקטים של הסכימה. האינטרפרטציה גם מספקת ערכי התחלה מתוך התחום  $D_I$ . לכל אינטרפרטציה I, הסכימה S יוצרת תוכנית  $program(S, I)$ .
  - 2. עבור ערכי התחלה נתונים x יש לתוכנית חישוב סופי או לא-סופי. אם הוא סופי, אזי  $val(S, I, x)$  יסמן את ערכי הפלט. אם החישוב לא-סופי, הביטוי  $val(S, I, x)$  אינו מוגדר.
- שתי סכימות תוכנית  $S_1, S_2$  נקראות שקולות אם עבור כל האינטרפרטציות I וכל ערכי ההתחלה x מתקיים:
- $$val(S_1, I, x) = val(S_2, I, x)$$
- או ששני הביטויים גם יחד אינם מוגדרים.

**הגדרה 14:**

- 1. משפט: כל סכימת תוכנית שקולה לסכימת while, שמתקבלת ע"י טרנספורמציה של הסכימה המקורית ותוספת משתנים בוליאניים (Boehm-Jacopini).
- 2. משפט: פקודת while אחת בלבד מספיקה לתוספת לטרנספורמציות במשפט הנ"ל (אולם מה שנקבל זה לגמרי בלתי יעיל).

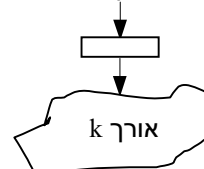
**דוגמה:** העברת תוכנית עם goto (עמ' 36) לתוכנית בלי goto (עמ' 36):

**אלג' 1:** משתמשים ב-2 טבלאות (עמ' 38) ולפיהן בונים תוכנית חדשה (עמ' 36). השיטה אינה יעילה, אבל אנו מחפשים הוכחת קיום (שניתן להיפטר מה-goto).

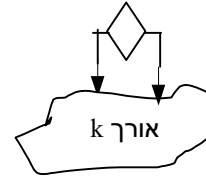
**אלג' 2:** הוכחה יותר יעילה – באינדוקציה.

נניח כי כל תוכנית באורך k מקיימת את משפט Boehm-Jacopini. נוכיח עבור  $k + 1$ :

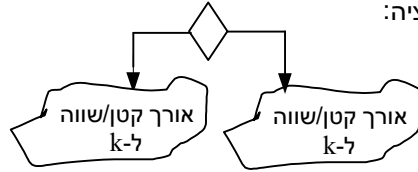
- 1. א. נניח שהפקודה הראשונה היא פקודת השמה: המקרה הזה טריוויאלי: יש זרימה סדרתית ולפי הנחת האינדוקציה ב-k אזי ניתן לבטל את ה-goto.



ב. ניה שיש פקודת if:



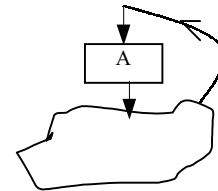
נעשה טרנספורמציה:



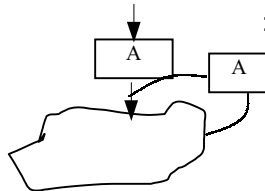
ולפי האינדוקציה ניתן לוותר על ה-goto.

2. מה אם יש goto ישיר בהתחלה, או שיש goto לפקודה הראשונה?

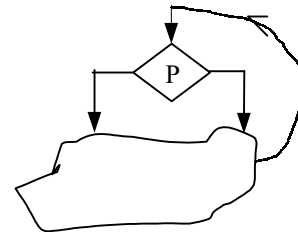
א.



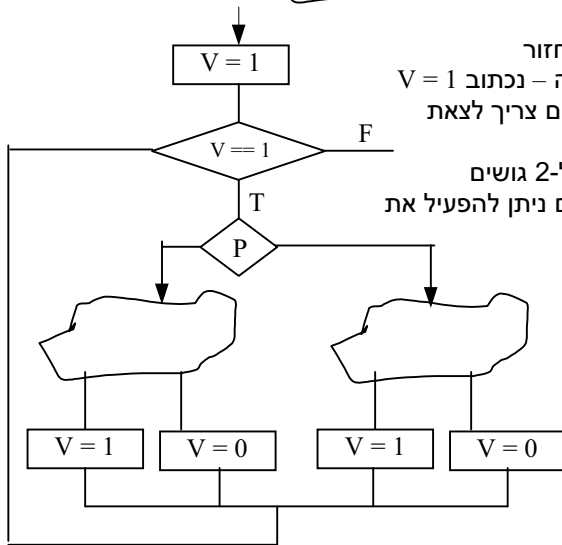
נוסיף את הפקודה שוב:



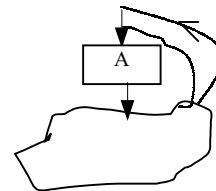
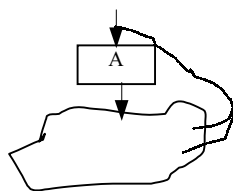
ב.



בכל מקום שצריך לחזור עם ה-goto להתחלה - נכתוב  $V = 1$  בשאר המקומות בהם צריך לצאת נדאג ש- $V = 0$ . בפנים נפרק את P ל-2 גושים שעבור כל אחד מהם ניתן להפעיל את הנחת האינדוקציה.



ג. אם במקומות שונים יש הסתעפות לפקודה הראשונה אז נאחד אותם:



ד. אם הפקודה הראשונה היא goto אזי ניתן לבטל אותה ואז להמשיך לפי הנחת האינדוקציה.

### 5. טיפוסים:

**הגדרה 8:** אובייקט פשוט הוא מושג המתואר ע"י המאפיינים הבאים: (שם, טווח, מיקום, זמן חיים, ערך, טיפוס).

שם: מזהה המסמן את האובייקט.

טווח: אוסף הפקודות בתוכנית בהן האובייקט נגיש.

מיקום: כתובת בזיכרון (מצביע) של שטח אחסון לאובייקט.

זמן חיים: פרק זמן בו יש שטח אחסון בזיכרון הקשור לאובייקט.

ערך: תוכן שטח האחסון.

טיפוס: זוג סדור  $(V, O)$ , כאשר  $V$  קבוצה של ערכים ו- $O$  אוסף של אופרטורים שניתן להפעיל על אברי הקבוצה  $V$ .

**הגדרה 9:**

1. אובייקט בגישה מכוונת עצמים הוא אוסף של קבועים ומשתנים שניתן להפעיל עליהם אוסף של מתודות המוגדרות במיוחד למטרה זו. מתודה מופעלת עי"כ שהודעה נשלחת לאובייקט כדי שיפעיל את המתודה.
2. מחלקה היא תיאור של אובייקטים דומים הנקראים מופעים של המחלקה. משתנה מחלקה (ב- ++C מגדירים אותו כ-static) משותף גם בשמו וגם בערכו לכל המופעים של המחלקה. משתנה מופע משותף רק בשם למופעים השונים של המחלקה.
3. ירושה היא שיתוף מבוקר של מידע (קבועים, משתנים, פונקציות ועוד) בין מחלקה אחת (העל-מחלקה) לבין מחלקה שנייה (התת-מחלקה). אם לכל מחלקה יש לכל היותר על-מחלקה אחת, יחס הירושה נקרא ירושה פשוטה. אם למחלקה אחת יכולות להיות מספר על-מחלקות, יחס הירושה נקרא ירושה מרובה. ההצגה הגרפית של יחס הירושה נקראת גרף הירושה של התוכנית.
4. רב-צורתיות במובן של הגישה מכוונת העצמים (Polymorphism) פירושה כי למחלקות שונות יש מתודות בעלות אותו שם. המתודה המופעלת ע"י אובייקט מסוים נבחרת **בזמן הביצוע** מתוך המחלקה לה שייך האובייקט.
5. הגישה מכוונת העצמים היא גישה המשתמשת ב: אובייקטים + מחלקות + ירושה (+ פולימורפיזם). מושג המחלקה הוא למעשה מודול עם אפשרות להסתרת מידע ואפשרות להגדרת יחס ירושה (פשוט או מרובה).

**הגדרה 10:**

משפט: יחס הירושה הוא טרנזיטיבי. גרף הירושה הוא גרף מכוון ללא מעגלים. לגרף בד"כ יש שורש יחיד. אם הירושה פשוטה, הגרף הוא עץ. במקרה זה הסגור הטרנזיטיבי של יחס הירושה מהווה יחס סדר שלם. אם הירושה היא ירושה מרובה, הסגור הטרנזיטיבי של יחס הירושה מהווה יחס סדר חלקי. לינאריות היא הגדרה של יחס סדר שלם על המחלקות כדי לפתור סתירות היכולות להתעורר עקב ירושה מרובה.

למשל: C יורשת מ-A ומ-B ולשניהם יש פונקציה f. אם אני אקרא ל-f(a) מ-C לאיזו f אני פונה? אם מגדירים לינאריות שאומרת סדרי עדיפות בין העל-מחלקות אז כן ניתן יהיה לקרוא ל-f(a) מ-C.

**5.1 שקילות של טיפוסים:**

1. **שקילות לפי מבנה:** 2 טיפוסים שקולים אם יש להם אותו מבנה. למשל, הטיפוסים הבאים שקולים:

<pre>type Complex is record   RE: integer;   IM: integer; end record;</pre>	<pre>type Another_Complex is record   RE: integer;   IM: integer; end record;</pre>
---	---

האם אם היינו משנים את סדרי השדות ואת שמותיהם הם גם היו שקולים? (ב-ALGOL'68 כן).

2. **שקילות לפי שם:** אם יש 2 טיפוסים, ברור שיש להם שמות שונים והם **לא שקולים** (ככה זה ב-Ada, C).

ניתן לבצע פעולות בין אובייקטים של type לבין אובייקטים של subtype, אבל לא ניתן לבצע פעולות בין אובייקטים מ-type שונה.

**הגדרה 15:**

1. אם הטיפוסים של כל האובייקטים בשפה ניתנים לקביעה (נקשרים) בזמן ההידור, אזי השפה היא בעלת טיפוס סטטי. אם קישור כזה בלתי אפשרי, אזי השפה היא בעלת טיפוס דינמי. בשפה בעלת טיפוס דינמי יש לכל אובייקט מתאר המתאר את הטיפוס ותכונות דינמיות אחרות הקשורות בכל רגע לאובייקט. הסבה של טיפוס יכולה להיות מפורשת או בלתי מפורשת.
2. מערכת כללי טיפוס עבור שפה היא קבוצת כללים המגדירים בצורה מדויקת את אפשרויות ההסבה מטיפוס לטיפוס, הן בצורה מפורשת והן בצורה בלתי-מפורשת. על סמך כללים אלו ניתן תמיד להחליט האם קיימת או לא קיימת התאמה של השמה או התאמה של פעולה בין שני טיפוסים שונים.
3. שקילות לפי מבנה פירושה שאפשר לבצע הסבה בלתי מפורשת בין אובייקטים משני טיפוסים בעלי אותו מבנה.
4. שקילות לפי שם פירושה שלא קיימת הסבה בלתי מפורשת בין שני טיפוסים (ז"א טיפוס שקול רק לעצמו).

למשל, שפת C היא בעלת טיפוס סטטי, ואילו APL היא בעלת טיפוס דינמי.

**Strong Typing 5.2:**

**הגדרה עמומה:** יש המגדירים שפה בתור Strong Typing אם כללי הטיפוס שלה מחמירים.

union dangerous

```
{ float x;
  int y; } z;
```

z.y = 3; cout << z.x // output: float representation of 3 (because z is a union – x and y resides on the same location)

**הגדרה 16:** שפה היא בעלת טיפוס חזק (strong typing) אם מתקיימים בה התנאים הבאים:

- א. השפה בעלת טיפוס סטטי, והמהדר כופה את מע' כללי הטיפוס על כל חלק של התוכנית.
- ב. המהדר מוודא את התאמת הטיפוס בין פרמטרים של פונקציות והארגומנטים המתאימים.
- ג. אין אפשרות לפרש את ההצגה הפנימית של אובייקט מטיפוס אחד כערך לפי טיפוס אחר אלא (אולי) ע"י קריאה מיוחדת לרוטינות של מע' ההפעלה.

יש המגדירים ששפה היא בעלת strong typing אם היא מקיימת את תנאי א', או את תנאי א' + ב', או את כל התנאים.

- ב-C המהדר אינה בודק התאמה בין הקריאה לפונקציה לפרמטרים שלה.
- C מקיימת רק את תנאי א'
- C++ ו-Pascal מקיימות את א' + ב'.
- Ada מקיימת את א' + ב' + ג'.
- ב-Scheme אין אפילו static typing.

**הגדרה 17:**

1. טיפוס הוא אזרח מסוג ראשון אם ניתן להשתמש באובייקטים מהטיפוס:
  - א. כארגומנטים לפונקציות.
  - ב. בפקודות השמה.
  - ג. כערכים מוחזרים ע"י פונקציה.
2. טיפוס הוא אזרח מסוג שני אם ניתן להשתמש באובייקטים מהטיפוס:
  - א. כארגומנטים לפונקציות, אך
  - ב. לא ניתן להשתמש באובייקטים בפקודות השמה.
  - ג. ולא ניתן להשתמש באובייקטים כערכים מוחזרים ע"י פונקציה.
3. טיפוס הוא אזרח מסוג שלישי אם לא ניתן להשתמש באובייקטים מהטיפוס לאף אחד מהשימוש המוזכרים בסעיף 1.

למשל: int הוא אזרח מסוג ראשון. תווית (label) לא מקיימת את ההגדרה הזו ב-C. ב-PL/1 כן ניתן לעשות השמה של תוויות.

**האם פונקציה היא אזרח מסוג ראשון? ב-Scheme כן.**

שם של קובץ ב-Pascal הוא אזרח מסוג שני, שכן פונקציה ב-Pascal יכולה רק לקבל שם של קובץ וזהו.

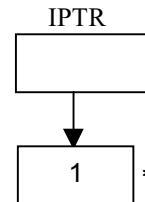
**5.4 Pointer Types:**

- ניתן לחלק את שפות התכנות לרמות לפי מצביעים:
1. אין בכלל מצביעים (Lisp, Prolog, שפות פונקציונליות).
  2. Java, Ada, Pascal.
  3. C, C++.

ב-Java אין אפשרות למצוא כתובות של תא בזיכרון.

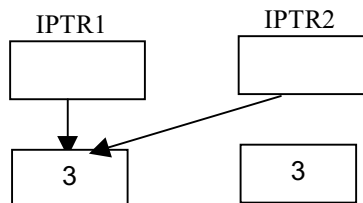
```
type PTR = access integer; // PTR is a pointer to int
var IPTR : PTR; // שיכול להכיל כתובת
begin
```

```
  new IPTR; // הקצאת תא בזיכרון
  IPTR.all := 1;
  FREE IPTR;
```



FREE IPTR גורם ל-\* לעבור לרשימת התאים הפנויים. אולם ההצבעה עליו עדיין נשאר והתוכן שלו נשאר ← **dangling pointer**.

```
IPTR1, IPTR2: PTR;
new IPTR1; new IPTR2;
IPTR1.all := 3; IPTR2.all := IPTR1.all;
IPTR2 := IPTR1;
```



אילו היה garbage collection אזי התא היה מתווסף לרשימת הפנויים. אולם בשפות שאין את זה  $\leftarrow$  *lost object*.

**5.6 פולימורפיזם כללי:**

**הגדרה 18:**

1. מושג נקרא מונומורפי אם יש לו טיפוס אחד ויחיד, אחרת הוא נקרא פולימורפי.
2. פונקציה נקראת פולימורפית אם יש לה פרמטרים (ואולי תוצאה) פולימורפיים.
3. פונקציה פולימורפית אוניברסלית מבצעת את אותן הפקודות עבור ארגומנטים מכל טיפוס מותר.
  - א. בפולימורפיזם פרמטרי יש לפונקציה פרמטר טיפוס מפורש או בלתי מפורש הקובע את הטיפוס של הארגומנטים עבור כל הפעלה (בדומה ל-template).
  - ב. בפולימורפיזם של הכלה ניתן לראות את הפונקציה כשייכת לתת-מחלקות שונות (כמו virtual).
4. פונקציה פולימורפית אד-הוק מבצעת פקודות שונות עבור ארגומנטים מהטיפוסים המותרים השונים.
  - א. בחפיפה/העמסה (overloading) משמש אותו שם לפונקציות שונות, והפונקציה שיש להפעיל נבחרת על-סמך ההקשר של התוכנית בנקודת הקריאה.
  - ב. בכפייה מופעל כלל טיפוס מתאים כדי להסב ארגומנט לטיפוס הנדרש ע"י הפונקציה.

למשל, (4 ב' )  $a + b$  : זה הרבה פונקציות (בין int, בין float וכו') שהקוד שלהם שונה (מעגלים אלקטרוניים שונים).

**5.5 הגדרות והצהרות:**

```
int x; // הגדרה
extern x; // הצהרה - x מוגדר בקובץ אחר
```

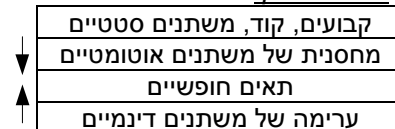
**6. יחידות תוכנית**

**6.1 סביבות**

קיימים 3 סוגי משתנים מבחינת הקצאת הזיכרון: סטטי, אוטומטי ודינמי.

static int a = 1; - בזמן הקומפילציה a מקבל מקום בזיכרון (המהדר עושה הקצאה למשתנים סטטיים). אורך החיים של משתנה סטטי הוא לאורך כל ביצוע התוכנית.

**מודל הזיכרון:**



<pre>void main() {     P(); P(); }</pre>	<pre>void P() {     static int a = 1;     cout &lt;&lt; a;     Q();     cout &lt;&lt; a;     a++; } 1, 2, 3, 2, 1 1, 2, 3, 2, 1</pre>	<pre>void Q() {     static int a = 2;     cout &lt;&lt; a;     R();     cout &lt;&lt; a;     a++; } אם לא היינו מצהירים על a בתור static</pre>	<pre>void R() {     static int a = 3;     cout &lt;&lt; a;     a++; } פלט: 1, 2, 3, 2, 1 2. 3. 4. 3. 2</pre>
--	---	--	--

<pre>static int y = 3; M(); void M() {     int y = 2;     cout &lt;&lt; G();     cout &lt;&lt; H(); }</pre>	<pre>int H() {     return F(); } int F() {     return 2 * y; }</pre>	<pre>int G() {     int y = 1;     return F(); }</pre>
---	--	---

בזמן הביצוע מתבצעת הקצאה של המשתנים האוטומטיים של כל פונקציה והם יושבים ברשומת ההפעלה של הפונקציה. המשתנה y שב-F או לא מקומי ביחס ל-F. מאיפה הוא נלקח? **חוק טווח סטטי:** לפי הכלה, כלומר: ה-y ייבחר לפי ה-y הגלובלי. **חוק טווח דינמי:** לפי הקריאה לפונקציה: מחפשים את y במחסנית. כשפוגשים את ה-y הראשון לוקחים אותו.

- 2. \* 1 = 2 חוק טווח דינמי: חוק טווח דינמי: 2. \* 3 = 6
- 2. \* 2 = 4 חוק טווח דינמי: חוק טווח דינמי: 2. \* 3 = 6

**עקרון המבניות גורר שיש להגדיר לפי חוק טווח סטטי** (שכן ההגדרה היא לפי התמונה הסטטית).  
חוק טווח דינמי עובד לפי התמונה הדינמית.

כל השפות כיום, כולל Scheme, הולכות לפי חוק טווח סטטי.  
Lisp הולכת לפי חוק טווח דינמי.

חוק טווח דינמי יותר מסובך בשפות בהן פונקציה יכולה להחזיר פונקציה ובשפות בהן פונקציה היא אזרח מסוג ראשון.

**הגדרה 19:**

1. משתנה נקרא מקומי או קשור ליחידת תוכנית אם הוא מוצהר ביחידה הנידונה עצמה.
2. משתנה נקרא לא-מקומי או חופשי ביחס ליחידת תוכנית אם הוא מוצהר ביחידה חיצונית מכילה.
3. משתנה נקרא גלובלי אם הוא מוצהר ביחידת התוכנית החיצונית ביותר הקיימת בשפה.

**הגדרה 20: (?)**

1. ההתאמה בין שמות אובייקטים למקומות אחסון בזיכרון נקראת סביבה.
2. הסגור של יחידת תוכנית הוא זוג סדור (P, E) כאשר P הפקודות של יחידת התוכנית ו-E הסביבה הלא מקומית הנוכחית.

**הגדרה 21:**

1. סביבת ההגדרה/הסביבה הלקסיקלית של יחידת תוכנית היא הסביבה בה שוכנת יחידת התוכנית.
2. סביבת הקריאה/הסביבה הדינמית של פונקציה היא הסביבה הקיימת בנקודת הקריאה לפונקציה.

**הגדרה 22:**

1. חוק טווח סטטי/לקסיקלי: הערך של משתנה לא-מקומי ביחידת תוכנית מסוימת נלקח מסביבת ההגדרה של היחידה.
2. חוק טווח דינמי: הערך של משתנה לא-מקומי בפונקציה מסוימת נלקח מסביבת הקריאה לפונקציה.

**6.2 מנגנוני העברה של ארגומנטים ופרמטרים:**

f(X) { ... } // הצהרה

f(S); // הקריאה לפונקציה

- X הוא פרמטר פורמלי, S הוא ארגומנט (אמיתי).  
נגדיר סדרה של פעולות בסיסיות שבאמצעותן נגדיר את המנגנונים השונים:  
A: הקצאת מקום בזיכרון לפרמטר X.  
B: חישוב כתובת של ארגומנט S.  
C: אחסון ערך מ-S לתוך X.  
D: חישוב כתובת של ארגומנט S.  
E: אחסון ערך מ-X לתוך S.  
F: שחרור השטח של X.

**המנגנונים השונים: לפי הקוד הבא:**

p1(x : integer) { x := x + 1; }	p2(x, y : integer) { x := x + 1; y := y + 1; }	p3(x, y : integer) { x := 2; y := y + 2; }	p4(x, y : integer) { x := 2; y := 8; }
s := 1; p1(s); put(s);	I := 1; s(1) := 7; s(2) := 100; p2(I, s(I)); put(I, s(1), s(2));	s := 1; p3(s, s); put(s);	I := 1; s(1) := 7; s(2) := 100; p4(I, s(I)); put(I, s(1), s(2));

0. האחדה.

**1. A, B, C, ..., F: call by copy-in**

כלומר: מבצעים A, אח"כ B, אח"כ C, אח"כ את הפונקציה עצמה (זה ה-...) ולאחריה את F.  
בהרבה מקרים פעולה B היא פעולת-דמה. למשל: כאשר מבצעים f(S) ו-S הוא סקלר, אולם אם כתוב f(S(I)) אזי צריך לחשב את הכתובת.  
במנגנון הזה מעבירים מידע פנימה לפונקציה, אבל בשום שלב לא מוציאים מידע מתוך הפונקציה.



ניתן להגיד ש-call by copy-in הוא: call by constant (בתוך הפונקציה אסור לשנות את הערך של x) ו-call by value.

p1: 1                      p2: 1, 7, 100                      p3: 1                      p4: 1, 7, 100

2. **call by copy-out**: A, B, ..., E, F.

אין העברה של נתונים לפונקציה. המידע מהפונקציה מועתק החוצה מ-X ל-S. עבור פונקציות שדורשות אתחול למשתנים המנגנון הזה יהיה "לא רלוונטי" או "כישלון".

הפונקציות p1, p2, p3 דורשות אתחול של X:                      p1: -                      p2: -                      p3: -                      p4: 2, 8, 100

I	s(1)	s(2)	x	y
1	7	100		
p4(I, s(I)): I = 1 → p4(1, s(1))				
			2	8
2	8			

הגדרה שונה: A, ..., D, E, F. ההבדל בין 2 ההגדרות (ששתיהן call by copy-out) הוא מתי מחשבים s(I).

p4: 2, 8, 100

I	s(1)	s(2)	x	y
1	7	100		
אם מחשבים את שניהם ביחד, לפני החזרת הפרמטר, אזי זה שקול להגדרה הקודמת: p4(I, s(I)): I = 1 → p4(1, s(1))				
			2	8
2	8			

p4: L-R 2, 7, 8

I	s(1)	s(2)	x	y
1	7	100		
אם מבצעים D, E, F ביחד עבור הפרמטר הראשון קודם, ואח"כ עבור הפרמטר השני, כאשר עוברים משמאל לימין, אזי: p4(I, s(I)): I = 1 → I = 2 → S(2)				
			2	
2				8
		8		

p4: R-L 2, 8, 100

I	s(1)	s(2)	x	y
1	7	100		
אם מבצעים D, E, F ביחד עבור הפרמטר הראשון קודם, ואח"כ עבור הפרמטר השני, כאשר עוברים מימין לשמאל, אזי: p4(I, s(I)): I = 1 → S(1) → I = 1				
				8
	8			
			2	
2				

3. *call by copy*: A, B, C, ..., E, F  
 זה קומבינציה של call by copy-in ו-call by copy-out.

p1: 2

p2: 2, 8, 100

I	s(1)	s(2)	x	y
1	7	100		
p2(I, s(I)): I = 1 → p2(1, s(1))				
			1 + 1 = 2	7 + 1 = 8
2	8			

הגדרה שונה: A, B, C, ..., D, E, F  
 כאשר שוב יש 3 אפשרות לביצוע D, E, F: ביחד, R-L, L-R.

p3:

s	x	y
1		
יש B ולכן הקריאה היא עם p3(s, s) → p3(1, 1)		
	1 + 1 = 2	1 + 2 = 3
L-R: 3 R-L: 2		

4. *call by reference*: B והעברת הכתובת וביצוע הפונקציה על S (אין X)

p1: 2

p2:

x	y	s(2)
I	s(1)	
1	7	100
יש B ולכן הכתובת מחושבת עכשיו: p2(1, s(1))		
1 + 1 = 2	7 + 1 = 8	

p3:

y
x
s
1
2
4

ברוב המקרים אין הבדל בתוצאות של call by copy ו-call by reference.

5. *call by name*: שכתוב הפונקציה ע"י החלפת X ב-S (תוך הימנעות מחפיפה מקרית עם משתנים מקומיים) וביצוע על הקוד שמתקבל.

p1: 2

s
1
שכתוב: s := s + 1
2

p2: 2, 7, 101

I	s(1)	s(2)
1	7	100
שכתוב: I := I + 1; s(I) := s(I) + 1;		
2		101

p3: 4

<b>s</b>
1
שכתוב: s := s + 1 s := s + 2
2
4

p4: 2, 7, 8

I	s(1)	s(2)
1	7	100
שכתוב: I := 2; s(I) := 8;		
2		8

ב-C יש מנגנון אחד: call by value והמתכנת יכול לשנות אותו ל-call by constant (ע"י הוספת const) או ל-call by reference (ע"י הוספת &).  
ב-Scheme זה call by value.  
ב-Ada יש 3 מנגנונים:

המתכנת כותב: IN: העברת מידע פנימה (ממומש ע"י call by constant – copy in).

OUT: העברת מידע החוצה (ממומש ע"י copy out).

IN OUT: שניהם. לא מוגדר אם ע"י call by copy או call by reference. מממש

הקומפיילר צריך להביא בחשבון את שניהם ומחליט לבד כך שזה יצא יעיל. המתכנת לא בוחר. מקרים פתולוגיים בהם 2 המנגנונים נותנים תוצאות שנות אסורים.

מה זה יעיל?

1. למשל, נניח ש-S היא מטריצה 100 x 100 ואנו קוראים ל-f(s):

ב-copy call צריך להעתיק את האיברים ובסוף הפונקציה צריך להעתיק חזרה למטריצה המקורית.

ב-reference call מעבירים מצביע ל-S ואין העתקה ל-X ומ-X יותר מהר.

2. S(32) וקטור בוליאני: כל מקום תופס בית אחד (8 ביטים).

אם יש בעיות בזיכרון אזי מצופים ושמים כל משתנה בוליאני רק כביט, אולם כעת לגשת לביט באמצע מילה לוקח הרבה זמן. ע"י call by copy ניתן להעביר את הווקטור המצופף כווקטור רגיל, מסודר בביתים וקעת ניתן לעשות את הפעולות על בתים – מהיר.