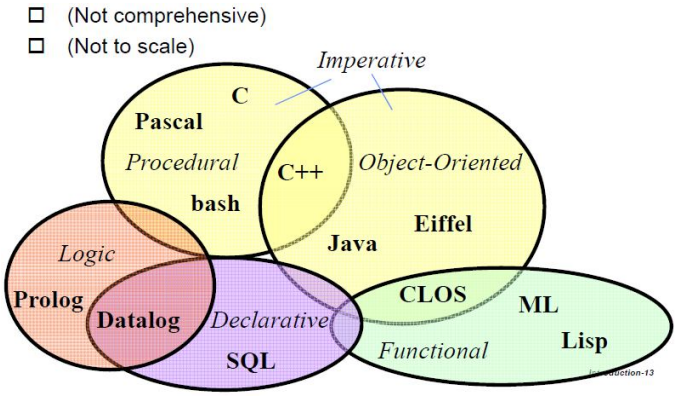
הקדמה

מדוע זקוקים לשפות תכנות?

- למתכנתים קשה לכתוב קוד בשפת מכונה או להבין קוד שכתוב בשפת מכונה.
- לוקח הרבה מאוד זמן לכתוב קוד בשפת מכונה.
- נוח להעביר רעיונות בין אנשים.

פרדיגמות של שפות תכנות

פרדיגמה: משפחה של שפות שיש להם בסיס משותף ואופן ביצוע דומה מבחינת החשיבה.



הפרדיגמה האימפרטיבית:

שפות לדוגמא: Fortran, Go ,Cobol ,Algol ,PL/I ,C ,Pascal ,Ada ,C++ ,Basic.

מצב התוכנית תלוי במצבה מבחינת הזיכרון והמשתנים של התוכנית.

היא מכילה פקודות אשר משנות את מצב התוכנית (השמות, sequencers וכו') אשר יכולים להתארגן במבנים של פרוצדורות ופונקציות.

שפות פרוצדוראליות (C, Pascal):

שורות התוכנית בזו אחר זו מגדירות בצורה מדויקת את הפקודות לביצוע.

שפות מונחות עצמים :

שפות לדוגמא: Smalltalk, C++, Objective C, Object Pascal, Beta, Eiffel.

מכילות אובייקטים, מבני נתונים בעלי מתודות ייחודיות. מתאפשרת ירושה בין אובייקטים שונים. כל אובייקט מכיל מידע לגבי עצמו ומסוגל לבצע פעולות על המידע הזה.

הפרדיגמה הפונקציונאלית:

שפות לדוגמא: Lisp, Scheme, Miranda, ML, Haskell.

שפה ללא מצבים. כל דבר בשפה הוא פונקציה שמחשבת חישוב ומחזירה תוצאה.

הפונקציות בעצמן הם עוד סוג של ערך אשר ניתן לחשב ולהעביר לפונקציות אחרות.

שימוש בפעולות רקורסיביות במקום בפעולות איטרטיביות.

שפות מאוד אלגנטיות ופשוטות.

הפרדיגמה הלוגית:

שפות לדוגמא: Prolog, Icon.

שפות בהן מתארים מצב נתון בצורה לוגית וחוקים לגביו ואז שואלים שאלות לגבי הנתונים הללו.
שפות בהן רושמים מה מחפשים ולא כיצד יש למצוא זאת.

סינטקס: איך ביטויים ופקודות נכתבים בשפה, איך הצהרות יוצרות תוכנית.
סמנטיקה: המשמעות של התוכנית.
אותה משמעות סמנטית יכולה להיכתב בסינטקס שונה בשפות שונות וגם ישנם ביטויים עם סינטקס זהה שיש להם משמעות סמנטית שונה בשפות שונות.

דרישות משפות תכנות:

- Universal: כל שפה צריכה להיות מסוגלת לבצע כל חישוב.
- Natural: שפה צריכה להיות קלה וטבעית לשימוש.
- Implementable: שפה שניתן לממש איתה דברים.
- Efficient: לשפה צריכים להיות ביצועים טובים (למרות שזה נתון לוויכוח כי לפעמים חשוב יותר הזמן שלוקח למתכנת לכתוב תוכנית בשפה מאשר זמן הביצוע שלה).
- Expressiveness: לשפה צריכה להיות יכולת הבעה טובה- שלמות של מכונת טיורינג אך שיהיה קל לממש בה מושגים בסיסיים.
- Efficiency: יעילות.
- Simplicity: פשטות השפה, מתקיים טריידאוף בין מינימליזם במספר הפקודות לקלות השימוש בשפה.
- Uniformity: אחידות, בין פקודות דומות באותה השפה.
- Abstraction: השפה צריכה לאפשר אבסטרקציה לתבניות החוזרות על עצמן.
- Clarity: קלות קריאה.
- Safety: מאפשרת מציאת שגיאות בזמן קומפילציה.

EBNF- Extended Backus-Naur Form:

דרך פורמאלית לתיאור דקדוק של שפת תכנות.
סימנים: | בחירה בין כמה אפשרויות, [ ] אופציונאלי, { } 0 או יותר חזרות.
דוגמא:

- <expression> = <term> {<add-op> <term>}
- <term> = <factor> {<mult-op> <factor>}
- <factor> = <variable-name> | <number>
- <add-op> = + | -
- <mult-op> = * | /

הביטוי: a + 2 / b – c * 7 הוא חוקי כי יש "עץ יצירה" מ-<expression> שמוביל אליו.

היסטוריה קצרה...

1950-1970:

FORTRUN: נובע מ- FORmula TRANslator, שפה שנועדה לחישובים נומריים.
COBOL: שפה לעיבוד של מידע.
Algol 60: נוספו בלוקים ומבני בקרה.
Lisp: נובע מ-List processing, שפה לעיבוד רשימות. השפה הפונקציונאלית הראשונה.
PL/I: שילוב רעיונות מחישוב נומרי (FORTRUN, Algol60) ומעיבוד מידע (COBOL).
Simula: שפה מונחית עצמים.

1970-1990:

C++, Smalltalk, Eiffel: עוד שפות מונחות עצמים.

Prolog: שפה לוגית.

ML, Miranda, Haskell: עוד שפות פונקציונאליות.

Ada: ניסיון יותר מוצלח מאשר PL/I, שפה הכוללת גם concurrency.

ועוד שפות לשימושים ספציפיים:

- SNOBOL, Icon, Awk, REXX, Perl: עבודה עם מחרוזות ועם סקריפטים.
- SQL: שפה לשאילתות ב-DB.
- Mathematica, Matlab: שפות ליישומים מתמטיים.

1990-היום:

Java, C#: שפות מונחות עצמים, ולתכנות ברשת.

Perl, Python, PHP, Ruby: שפות סקריפטים עם תכנות עצמים וגם לתכנות ברשת.

ערכים וטיפוסים

ערך:

ערך: ישות המתקיימת במהלך ריצת התוכנית, משהו שיכול להשתנות במהלך ריצה של תוכנית.
הגדרה אלטרנטיבית (פסקל): כל דבר שניתן להעביר לשגרה.

דרכים לסיווג ערכים:

1. לפי מבנה (structure): הדרך שבה הערך מוגדר.
כל ערך שמור בזמן ריצה כרצף כלשהו של ביטים.
טיפוסים שונים תופסים כמות שונה של מקום בזיכרון.
שפות תכנות נבדלות באופן שבו ערך מיוצג במכונה, ובאופן רמת הפירוט לגבי הייצוג במכונה.
(בפסקל: לא ניתן למשתמש מידע על הייצוג, ב-Java הייצוג הינו חלק מהגדרת השפה לכל ערך).
ערך פרימיטיבי: לא ניתן לפרקו לערכים פשוטים יותר. ערכים פרימיטיביים הם שונים משפה לשפה.
ערך מורכב: מורכב מערכים אחרים. דוגמאות: רשומה, מערך, סט, קובץ.
הדרך ליצירת ערכים מורכבים בשפה בדרך-כלל לא תלויה באופן המימוש שלו.
לעיתים לתת-הערך יש "שם" (בתוך רשומה) ולעיתים יש לו "מסלול" (בתוך מערך או קובץ).
2. לפי הפונקציונאליות: האופן שבו משתמשים בערך והפעולות המותרות עליו.
לא בכל שפה ניתן לעשות את כל הפעולות על כל ערך.
סוגי פעולות על ערכים:

- העברה כארגומנטים לשגרה.
- החזרה דרך ארגומנט של פרוצדורה.
- החזרה כתוצאה של פונקציה.
- השמה לתוך משתנה.
- שימוש ליצירת ערך מורכב.
- יצירה/חישוב הערך ע"י הערכת ערכו של ביטוי.

First class value: ערך שניתן לבצע עליו את כל סוגי הפעולות.
באופן תיאורטי היינו רוצים לאפשר ביצוע של כל פעולה על כל ערך אבל ישנן בעיות בכך:

- מימוש מסובך מידי בשפת התכנות.
- הגדרת הסמנטיקה בצורה מדויקת היא בעייתית.

פסקל:

First class values: רק הערכים הפרימטיביים truth values, characters, enumerands, integers, reals, pointers.

Lower class values: יכולים להיות מועברים כפרמטרים לשגרה אבל לא יכולים להיות מוחזרים ממנה או לשמש כתת-ערכים בתוך ערך מורכב אחר- ערכים מורכבים (רשומה, מערך, סט וקובץ), פרוצדורות ופונקציות, רפרנסים למשתנים.

דוגמא: פרוצדורה A מחזירה כערך את הפרוצדורה B, אבל כאשר A הסתיימה, B כבר לא קיימת! ולכן לא נרצה לאפשר החזרת פרוצדורה מפרוצדורה.

ML:

כל ערך הוא first class value.

ערכים פרימיטיביים: truth values, integers, reals, strings.

ערכים מורכבים: רשומות, טאפלים, רשימות, מערכים.

ביטוי: הגדרה בשפת התכנות ל-איך לקחת ערך אחד או יותר ולייצר מהם ערך חדש.

בחלק משפות התכנות ניתן כבר בזמן כתיבת התוכנית לדעת מה יהיה הטיפוס של הערך גם אם ערכו עדיין לא ידוע- מאפשר למצוא באגים.

טיפוסים-סוגי ערכים:

בשפת מכונה: כל הערכים הם רצפים של ביטים- אין סוגים שונים של ערכים.
בשפת אסמבלר: קיימים שני סוגי ערכים- מידע וכתובות. המתכנת לא יכול להגדיר סוגי ערכים חדשים.
בשפות עיליות: סוגים שונים של ערכים.
אבחנה: טיפוס הוא גם תכונה של:

- תא בזיכרון
- ביטוי, שממנו מחשבים את הערך.

טיפוס כקבוצה של ערכים:

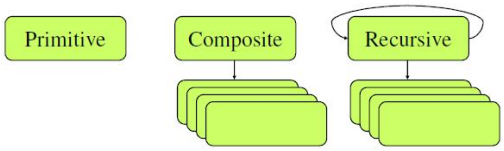
טיפוס זה קבוצה של ערכים. אבל לא כל קבוצת ערכים מייצגת טיפוס.
קבוצה לא יכולה להכיל איברים מאותה רמת היררכיה שלה עצמה.
זו קבוצת ערכים עם פעולות שניתן להפעיל באופן אחיד על כל ערך בקבוצה.
דוגמא: {false, true} מתאים לטיפוס בשפות תכנות רבות.

סוגי טיפוסים:

Primitive type: טיפוס המכיל רק ערכים פרימיטיביים. נחלקים ל:

- Rudimentary: טיפוסים מובנים בשפת התכנות (integer, real, char).
- Non-Rudimentary: טיפוסים שהוגדרו ע"י המשתמש. נחלקים ל:
  - Enumerated types: TYPE Month = (January, February… .
  - Subranges: רצף של ערכים עוקבים. TYPE DayOfMonth = 1..31 (זהו למעשה תת-תחום של המספרים השלמים).

Composite type: טיפוס המכיל ערכים מורכבים.
Recursive type: טיפוס המכיל ערכים מורכבים שיש בהם ערכים מאותו טיפוס (מערך, רשימה).



מערכת טיפוסים: חלוקה של המרחב של הערכים שתוכנית יכולה לפעול עליהם.
החלוקה קובעת אילו פעולות מותרות על איזה ערכים וקובעת את משמעותם.

Primitive types:

נקודות מבלבלות:

- שפות תכנות שונות נותנות שמות שונים לאותם טיפוסים פרימיטיביים:
  דוגמא: Pascal :Boolean, Integer ,ML :bool, int.
- שפות תכנות שונות מגדירות סט אחר של ערכים עבור אותו טיפוס פרימיטיבי.
- ישנן שפות המגדירות כמה טיפוסים שונים ל-integers ול-reals.
  דוגמא: C,C++: float, double עבור real.
  דוגמא: Java: byte, short, int, long עבור טווחים שונים של Integer.
- ב-C ו-C++ הערכים של טיפוס בוליאני מציינים 0 וערך מספרי אחר.

אבחנה: יש להבחין בין הערך לשם של הערך. לפעמים גם לשם הערך יש שם בפני עצמו.
קרדינליות של טיפוס: מספר הערכים בקבוצה שמתאימה לטיפוס.
דוגמאות: #Boolean=2, #char=256, #integer (pascal) = 2*(maxint+1).

טיפוסים שיש בהם סדר:

Ordered type: טיפוס שמוגדר עליו יחס סדר מלא. טוב לשימוש בלולאות במשפטי תנאי.

Unordered type: טיפוס שלא מוגדר עבורו יחס סדר מלא.

דוגמא: המספרים המרוכבים.

Discrete type: טיפוס בדיד עם יחס סדר מלא, שלקבוצת הערכים שלו יש סדר יחיד ביניהם. יכול לשמש כאינדקס של מערך.

דוגמאות: boolean, char, integer.

Ordered indiscrete type: טיפוס לא בדיד.

דוגמאות: string, real.

פסקל: רק discrete types יכולים לשמש כאינדקסים בלולאות/מערכים וב-case statements.

C++: רק discrete types יכולים לשמש כפרמטר פורמאלי לטמפלייט.

Semi-primitive type:

סוג-1: Atomic: ערכים שלא ניתנים לפירוק לערכים מטיפוסים קטנים יותר.

Non-rudimentary: טיפוסים שהוגדרו ע"י המתכנת.

סוג-2: Non-atomic: ערכים הניתנים לפירוק לערכים מטיפוסים קטנים יותר.

Rudimentary: מובנה מראש בשפה.

מחרוזות:

רצף של תווים. אין קונצנזוס לגבי:

- האם זהו טיפוס פרימיטיבי או מורכב?
- האם באורך קבוע או משתנה?
- איזו פעולות על מחרוזות נתמכות ע"י השפה?
- מהם הליטרלים, delimiters או quotes?

שפה	סוג	פעולות מותרות
ML	primitive	שוויון, שרשור, פירוק לרכיבים מוגדרות כחלק מהשפה.
ADA, Pascal	מערך של תווים	ניתן להשתמש בפעולות על מערכים. חסרון: כל הפעולות מוגדרות על מערכים בגודל קבוע.
Algol-68	מערך של תווים שגודלו יכול להשתנות בזמן ריצה	ניתן להשתמש בפעולות על מערכים.
C	בעזרת מצביעים ניתן לממש מערכים שמתנהגים כמערכים דינמיים	ניתן להשתמש בפעולות על מערכים.
Prolog	רשימה של תווים. ניתן לגשת ישירות רק לתו הראשון	אין פעולות מוגדרות בשפה מלבד בחירת התא הראשון.

הערה: ב-C++ זה לא מובנה בשפה. בדרך-כלל רצוי שזה כן יוגדר בשפה.

Composite types:

Type operators: יוצרים טיפוסים מורכבים חדשים בעזרת טיפוסים פרימיטיביים וטיפוסים מורכבים.

דוגמאות: טאפלים, רשומות, unions, מערכים, סטים, מחרוזות, רשימות, עצים, יחסים וכו'.

טיפוסים המוגדרים לפי אופרטורים בתורת הקבוצות:

- מכפלה קרטזית- טאפל ורשומה.
- איחוד- unions ו-variants.
- מיפוי- מערכים ופונקציות.
- קבוצות חזקה- סט.

מכפלה קרטזית: הדרך הפשוטה ביותר להרכיב טיפוס חדש בהינתן שני טיפוסים.
בהינתן שני טיפוסים S ו- T, המכפלה הקרטזית שלהם היא: $S\times T=\{(x,y)|x\in S,y\in T\}$.

קרדינליות: $\#(S\times T)=\#S\times\#T$.

Homogenous tuples: סוג מיוחד של מכפלה קרטזית ובו כל האיברים הם מאותו טיפוס.

סימון: $S^n=S\times\cdots\times S$.

קרדינליות: $\#(S^n)=(\#S)^n$.

S^0 - בעל ערך אחד: מכפלה קרטזית עם 0 איברים.

הטיפוס Unit: טיפוס שיש לו ערך אחד. קרדינליות-1. הוא לא צריך להיות ערך בשפה (לא דרוש עבורו מקום אחסון בזיכרון). מתאים ל-unit ב-ML ול-void ב-C.

הטיפוס void ב-C: לא ניתן להגדיר משתנה מטיפוס void ב-C.

דרכים אלטרנטיביות להגדרת unit:

1. enum עם ערך אחד

```
typedef enum Unit {
      unit
} Unit;
```

The value of this type is written {} in C, however, this value can only be used in initialization statements

Or...

2. רשומה שאין בה אף שדה

```
typedef struct {
} Unit;
```

Ditto!

Or...

3. מערך בגודל 0

```
typedef int Unit[0];
```

Disjoint unions: אופרטור האיחוד על קבוצות: $S+T=\{left\,x|x\in S\}\cup\{right\,y|y\in T\}$.

דוגמא: פויינטר הוא כתובת בזיכרון או null וזה למעשה איחוד הכתובות בזיכרון מסוג מצביע + 1 (איחוד עם unit).

Variant record בפסקל: disjoint union של integer ו-real. הם יושבים באותו מקום פיזית בזיכרון והערך הוא או זה או זה.

```
type Accuracy = (exact, approx);
     Number = record case tag: Accuracy of
         exact: (ival: Integer);
         approx:(rval: Real)
     end
```

```
VAR n: Number;
function round_num(n: Number): Integer;
  case n.tag of
   exact:  round_num := n.ival;
   approx: round_num := round(n.rval)
  end;
end;
```

גישה בטוחה נעשית ע"י בדיקת ערך ה-tag, כך ניתן לדעת איזה מן הערכים זמין.

Variant record ב-ML: שפה שהיא יותר type-safe- ניתן טכנית לפנות רק לשדות שהטאג הוא שלהם.

```
datatype number = exact of int | approx of real;
case n of exact i => i | approx r => round(r);
```

Tagging: מנגנון שמאפשר לדעת איזה מן השדות פעיל במבנה.

ML: tagging נעשה אוטומאטית- כשמכניסים ערך, מוצמד אליו טאג שהוא implicit, לא ניתן לגשת אליו ישירות. השפה היא type-safe: לא ניתן טכנית לקרוא ערך מטאג שלא הוגדר.

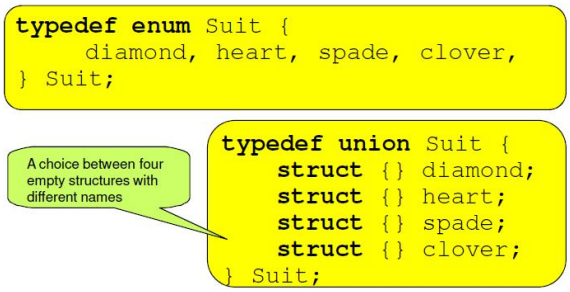
פסקל: הקומפיילר כופה הגדרה של שמות לשדות ה-tag. אבל הוא לא אוכף גישה לרשומה רק לפי הטאגים.

טורבו פסקל: מאפשר למתכנת גם לא להגדיר טאג בכלל.

C: אחריות המתכנת לזכור מהו הטאג, להגדיר אותו ולהשתמש בו בהתאם.

שימוש נוסף ל-tagging: הבדלה בין שני ערכי int שונים

```
typedef union
height {
   int cm;
   int in;
};
```

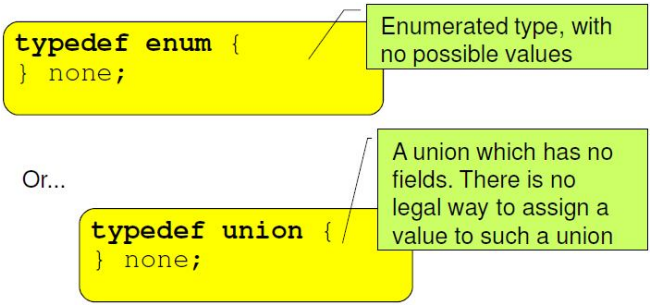


דימוי enum בעזרת union: זו למעשה בחירה בין units.

הטיפוס none: טיפוס שאין בתוכו אף ערך חוקי. מעין "איבר יחידה" של החיבור. קרדינליות: 0.

דוגמא: פונקציה שמחזירה none לא יכולה לחזור כי היא לא יכולה להחזיר ערך חוקי. למשל-exit פונקציה שלעולם לא חוזרת, היינו רוצים שתהיה מוגדרת כמחזירה none.

הגדרת הטיפוס none ב-C:



None לא קיים ב-C.

הערה: הקבוצה הריקה שונה מהקבוצה עם ערך אחד (unit).

Mappings: אופרטור המיפוי.

מערך: T of array $[S]$ מתאים למיפוי: $S \to T$.

T of array $[S_1,\ldots,S_n]$ מתאים למיפוי: $(S_1 \times \cdots \times S_n) \to T$, וגם ע"י curring: $(S_1 \to (\cdots \to S_n)) \to T$.

פונקציה: (n:Integer):Boolean function מתאימה למיפוי: Integer → Boolean.

קרדינליות: $\#(S \to T) = \#T^{\#S}$

דוגמא: $(boolean \times chat) \to month,\ |month|^{|boolean|\times|char|}$.

Power sets: תת-קבוצה, אין משמעות לסדר ויש אחד מכל איבר.

שווה-ערך ל: $T \to Boolean$ (ממפה כל ערך בסט ל-true וכל ערך אחר ל-false).

Recursive types:

בהגדרת הטיפוס נעשה שימוש בטיפוס עצמו.

ML: הגדרה ישירה-

```
datatype intlist = nil | cons of int * intlist;
```

כל רשימה היא סופית אבל אין הגבלה על גודל הרשימה.

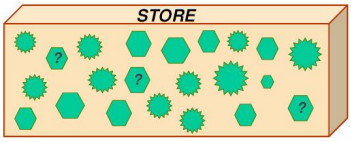
הערה: ניתן לקבל ראש וזנב של רשימה. אין פונקציה שמחזירה ספציפית את האיבר ה-n ברשימה.

Ada/C/Pascal: טיפוסים רקורסיביים צריכים להיות מוגדרים עם מצביעים.

בדיקת תקינות: הקומפיילר צריך לוודא תקינות של התוכנית על-סמך מעבר אחד. בעבר, לעיתים מעבר שני לא היה אפשרי מבחינה טכנולוגית.

Storage

מודל למרחב אחסון המידע של תוכנית:



Store: אוסף של תאים. תא יכול להיות מוקצה או לא מוקצה. לתא מוקצה יש תוכן: ערך כלשהו או undefined.

```
Example:
{
  int n;   // Some unallocated cell changes status to allocated
  n = 1;   // Content changes from undefined to 1
  n++;     // Content changes from 1 to 2
}          // Cell changes status to unallocated
```

הערה: ב-C כשתוכנית מתחילה, 1000 בתים ראשונים מאופסים.

Composite Variables:

Composite value: יש לו תת-רכיבים שהם ערכים אשר ניתן לגשת אל כל אחד מהם בנפרד.

Composite variable: יש לו תת-רכיבים שהם משתנים. ניתן לגשת ולעדכן כל אחד מהם בנפרד.

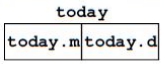
מבנה: בדרך-כלל המשתנה בנוי כמו הערך מאותו הטיפוס.

דוגמא למקרה חריג: "מערך ארוז" בפסקל- ייצוג של מערך ע"י גודל של ביטים. לא ניתן לגשת למערך לפני שעושים לו- unpacked.

משתנה רשומה: טאפל של משתנים.

משתנה מערך: מיפוי מאינדקס לערך.

```
• type Date = record m: Month; d: 1..31 end;
• var today: Date;
```



Static array: גודל המערך נקבע בזמן הקומפילציה.

Dynamic array: גודל המערך נקבע בזמן ריצה ברגע יצירת המערך.

Flexible array: ניתן לשנות את גודל המערך בזמן ריצה גם לאחר הגדרתו (קורה למשל כאשר ניגשים לאינדקס גדול יותר).

מערכים אסוציאטיביים: מיפוי מטיפוסים שהם לא בעלי תחום רציף.

ניתן גם להגדיר מפתח שאין עבורו ערך.

ובד"כ ישנה אפשרות להחזיר את כל ערכי המפתחות.

```
$food["dog"] = "bonzo";
$food["mouse"] = "cheese";
$food["shark"] = "fish";
echo $food[$pet];
```

```
var myArray = []
myArray["key_1"] = "some value"
myArray["key_2"] = undefined      // no value
document.write(myArray["key_2"]) // output: undefined
document.write(myArray["no_such_key"]) // again, output: undefined
for (var key in myArray)
   document.write(key)                    // output: key_1, key_2
```

Reference Vs. Value Semantics:

Value semantics: הערכים האקטואליים הם אלה שמועתקים או משווים.

- Deep value semantics: כל ההיררכיה של האיברים מועתקת.
- Shallow value semantics: רק הרמה הראשונה בהיררכיה מועתקת, כל שאר הרמות מועתקות כ-reference.

תכונות: איטי, דורש מידע בזמן ריצה לגבי הטיפוסים המאוחסנים בכל רמת היררכיה.

שפות: Lisp, ML, Prolog.

Reference semantics: רק הרפרנס- משתנה המכיל מצביע אל הערך, מועתק.

תכונות: מהיר, ניתן לשתף אובייקטים, אין צורך לאחסן טיפוסים במהלך זמן ריצה, לבדיקת שוויון ניתן פשוט להשוות את כתובות המצביעים.

שפות: Java, Smalltalk, C#.

הגישה כיום: value semantics עבור משתנים פרימיטיביים, Reference semantics עבור משתנים מורכבים.

Lazy copying: העתקת כמות גדולה של מידע נעשית תחילה ע"י reference ופעולת ההעתקה בפועל נעשית רק כאשר נעשה שינוי באחד משני המקומות (אשר לא אמורים לשתף את המידע).

Copy Vs. Reference Semantics:

מה קורה כאשר נעשית השמה למשתנה מורכב?

Copy/value semantics: כל הרכיבים של המשתנה המורכב מועתקים למקום היעד.

Reference semantics: למשתנה היעד יהיה מצביע או רפרנס למשתנה המקור.

דוגמאות ל-workarounds:

- אמנם ב-Java יש reference semantics אבל overriding למתודה מאפשר למתכנת לשלוט בעומק ההעתקה:

```
Date dateT = new Date(2006, 1, 1);
dateT = dateR.clone();
```

- ב-C++ בה בדרך-כלל יש value semantics משתמשים במצביעים בשביל reference.

Life Time:

הזמן בין הקצאה של משתנה לבין שחרור שלו.

ניהול אורח החיים של משתנים חשוב לניהול הזיכרון של התוכנית.

Block activation: משתנים לוקאליים, חיים רק בבלוק.

Programmer's decision: משתנים על ה-heap שהוקצו באופן דינאמי ע"י המתכנת. מתים כתלות בגישה של שפת התכנות.

Program activation: משתנים גלובאליים, חיים לאורך כל התוכנית.

Permanent: ממשיכים יותר מאורך חיי התוכנית, למשל: קובץ.

ככל שמשתנה חי יותר, כך יותר קשה לנהל אותו.

בלוק: **פסקל:** פונקציות ופרוצדורות.

ML: let expression.

C++: כל מה שבתוך { }.

משתני בלוק: לוקאלי: מוגדר בתוך הבלוק וניתן לשימוש בתוכו בלבד.

גלובלי: משתנה לוקאלי המוגדר בבלוק הכי חיצוני של התוכנית.

Block activation: פרק הזמן בו בלוק מסוים רץ.

משתנה בעל אותו שם בבלוק יכול להיות בפועל מספר משתנים שונים: בריצות שונות של הבלוק המשתנה נוצר כמה פעמים, בקריאה רקורסיבית ישנן מספר יצירות בו-זמנית של אותו שם משתנה.

משתנה סטטי: משתנה שמשך החיים שלו הוא כל חיי התכנית, ללא קשר למקום בו הוא מוגדר בפועל. המטרה היא לאפשר אחסון ערכים הדרושים להפעלות שונות של אותו המודול.

הערה: המשתנה גלובאלי מבחינת משך החיים שלו אבל לוקאלי לפונקציה- ניתן להשתמש בו להתייחס אליו רק בתוך הפונקציה שבה הוא מוגדר.

משתני heap: יכולים להיווצר ולהימחק ע"י המתכנת, ע"י: allocate, deallocate. למשתנה ניגשים ע"י מצביע או רפרנסים.

הערה: מאפשר גם מבני נתונים רקורסיביים (למשל: עצים).

Dangling References: מצביעים לזיכרון שכבר שוחרר.

דליפת זיכרון: כאשר שוכחים לשחרר משתנה- heap, וכל המצביעים אליו כבר לא קיימים. אם התוכנית ממשיכה לרוץ זמן רב, בסופו של דבר ייגמר לה המקום בזיכרון.

דוגמא: global מצביע על משתנה לוקלי שמת כאשר יוצאים מה-scope של הפונקציה. המקום בזיכרון כבר לא שייך למרחב הזיכרון של התוכנית.

```
int *global;

void foo() {
  int local = 3;
  global = &local;
}

int main() {
  foo();
  *global = 17;
}
```

הערה: מניעת דליפות זיכרון זו אחת הסיבות שב-C אין פונקציות מקוננות. ובפסקל אין משתנים שמצביעים על פונקציה.

Garbage Collection:

מנגנון אוטומאטי לניהול זיכרון. המתכנת לא משחרר זיכרון, וכאשר כבר לא נותר הרבה זיכרון, המנגנון עובר ומפנה אזורי זיכרון שכבר אינם בשימוש התוכנית

Mark: סימון כל התאים שאינם בשימוש. **Sweep:** הורדת סימון של מה שהוא כן בשימוש (מחסנית, משתנים גלובליים) ותאים שניתן לגשת אליהם. **Release:** שחרור כל מה שנשאר מסומן.

הערות:

- המנגנון מסתמך על כך שהמתכנת ישים Null במצביעים שכבר אינם בשימוש.
- במערכות עם garbage collection אין משתנים שמוקצים על המחסנית.
  - Escape analysis: ב-Java הקומפיילר בודק האם משתנה שמוקצה עם רפרנס אי-פעם "חורג" מגבולות הפונקציה, ואם אינו חורג, הקומפיילר מאפשר הקצאה שלו ופינוי שלו מהמחסנית בכל פעם שהפונקציה נקראת.

Closures: הגדרת פונקציה השומרת איתה את כל מה שקיים ב-scope בזמן שהיא נוצרת.

דוגמא: זה משמש בעיקר לפונקציות שמחזירות פונקציות ואז פונקציה פנימית קיימת גם כאשר הפונקציה שהחזירה אותה חדלה מלהתקיים. אבל הפונקציה הפנימית מתייחסת גם למשתנים פנימיים שהיו בפונקציה שהסתיימה ולכן זה טוב שהפונקציה יכולה להמשיך להשתמש בהם.

```
function makeAdder(i) {
  var incr = i;
  function adder(n) {
    return n + incr;
  }
  return adder;
}
```

משתנה לוקלי של makeAdder

משמעות השם: יצירת סגור על ערך אחר של המשתנה המקומי (במקרה הזה- incr).

ה-scope של המשתנה המקומי: ה-scope של הערך שסוגר אותו.

המשך דוגמא:

ניתן להמשיך להשתמש ב-add3 עם המשתנה שהוכנס ל-makeAdder.

```
var add3 = makeAdder(3);
document.write(add3(7));    // Output: 10
document.write(add3(12));   // Output: 15
var add8 = makeAdder(8)
document.write(add8(12));   // Output: 20
```

מערכות טיפוסים

Typing: מנגנון שנועד לסווג את הטיפוסים בשפת התכנות כדי לתאר את המידע ולמנוע פעולות לא הגיוניות עליו.

מרכיבי המנגנון:

- ישנו טיפוס לכל ערך. ישנו לפחות אחד ולעיתים יותר.
- מנגנון בדיקה: בהינתן ביטוי, קובע את הטיפוסים השייכים לו ונותן משמעות לפעולות (בהתאם לטיפוס). בודק האם הפעולה מותרת על הערך.
- מנגנון הסקה: מסוגל להסיק את המשמעות של הביטוי ואת הטיפוס של התוצאה.

הערה: לשפה ישנם כללים לגבי טיפוסים וחוקיות פעולות עליהם והקומפיילר בודק לפיהם.

סיווג מערכות טיפוסים:

Existence: האם בשפה ישנה מערכת טיפוסים.

Type equivalence & subtyping: מתי טיפוס אחד יכול להחליף אחר- subtyping ומתי שני טיפוסים נחשבים שקולים- type equivalence.

Strength: עד כמה הקומפיילר אוכף את החוקים לגבי הטיפוסים.

Time of checking: באיזה שלב נעשית בדיקת התקינות של הטיפוסים (קומפילציה או זמן ריצה).

Responsibility for tagging: של מי האחריות להגדיר את הטיפוסים עבור הערכים.

Flexibility: באיזו מידה מערכת הטיפוסים מגבילה את המתכנת.

Existence:

Typed: ניתן לחלק את הערכים של השפה לקבוצות עם התנהגות דומה פחות או יותר.

שפות לדוגמא: C, Pascal, ML, Ada, Java.

Untyped: לכל ערך יש סט נפרד של פעולות מותרות ואסורות.

שפות לדוגמא: **Lisp:** כל הערכים הם מסוג S-expressions

Mathematica: כל הערכים הם "ביטויים סמבוליים"

C++ templates: הערכים הם types.

Degenerate: ישנו מספר מאוד מצומצם של טיפוסים או טיפוס יחיד.

שפות לדוגמא: **BPCL:** הטיפוס היחיד הוא: machine word.

DOS Batch: הטיפוס היחיד הוא מחרוזת.

C-Shell: שני טיפוסים- מחרוזות ומספרים.

Type equivalence:

נניח שפעולה מצפה לקבל טיפוס T. באיזה תנאים היא יכולה לקבל טיפוס T'?

אם היא יכולה, T' הוא **תת-טיפוס** של T.

טיפוסים שקולים: תת-טיפוס אחד של השני.

שקילות מבנית: לטיפוסים יש בדיוק את אותו המבנה.

הגדרת השקילות: אם הטיפוסים פרימיטיביים הם שקולים אם"ם הם זהים.

אם הם טיפוסים מורכבים:

- $T = A \times B,\ T' = A' \times B'$, or
- $T = A + B,\ T' = A' + B'$, or $T = A + B,\ T' = B' + A'$, or
- $T = A \rightarrow B,\ T' = A' \rightarrow B'$

ומתקיים: A, A' שקולים ו- B, B' שקולים, אז: $T \cong T'$.

<u>הגדרת שקילות לטיפוסים רקורסיביים:</u>

- $T = Unit + c(A \times T)$ ניתן לראות שאלה הם ביטויים שקולים אבל קשה מאוד לבדוק שקילות
- $S = Unit + c(A \times S)$ עבור טיפוסים רקורסיביים.

<u>הערות:</u>

- ברוב השפות רשומות בעלות שדות מאותו טיפוס אך עם שמות שונים, לא שקולות.
- בפסקל: אין שקילות מבנית, השקילות היא רק לפי שם.
- ב-C: יש שקילות מבנית, אבל ברשומות נדרשים שמות זהים לשדות.

<u>שקילות לפי שם:</u> הטיפוסים שקולים אם"ם הם בעלי אותו השם.

דוגמא:

```
TYPE  T1 = File of Integer;
      T2 = File of Integer;
VAR   f1: T1;
      f2: T2;
Procedure p(Var f: T1);
....
p(f1);  (* ok *)
p(f2);  (* compile-time error *)
```

הפרוצדורה p מוכנה לקבל רק פרמטר מסוג שנקרא: T1.

פסקל:

דוגמא: אם נגדיר: Sort(var a:array[0..50] of integer);

Sort לא תוכל לקבל שום-דבר כי הגדרנו שהיא מקבלת טיפוס ללא שם.

הדרך היחידה להפעלת פעולה על משהו בפסקל היא הגדרת שם הטיפוס:

Type i50 – array[0..50] of integer. וכעת נוכל להגדיר את sort כמקבלת טיפוס כזה.

דוגמא-שקילות בין תוכניות: בפסקל, שקילות לפי שם כוללת גם הגדרה באותו המקום. ולכן שתי תוכניות שונות בפסקל לא יכולות לתקשר ביניהן דרך קבצים עם טיפוסים שהוגדרו ע"י המשתמש, כי כל תוכנית מגדירה את הטיפוס אצלה, ולכן הם מוגדרים במקומות שונים.

```
program p1(f)
type T = file of Integer;
var  f: T;
begin
 ...
 write(f,...);
 ...
end;
```

```
program p2(f)
type T = file of Integer;
var  f: T;
begin
 ...
 read(f,...);  (* Type error *)
 ...
end;
```

זאת מכיוון שפסקל לא יכולה לוודא ששני הטיפוסים שהוגדרו באותו השם, הוגדרו באותו האופן ולא רוצה לאפשר פתח לשגיאות.

בנוסף, Include לאותה הגדרה כן יעבוד בפסקל, אבל זה לא חוקי מבחינת "שקילות לפי שם" כי ייתכן שבין הזמן שבו קובץ אחד עשה include וקומפל לזמן שבו השני עשה include, קובץ ה-h השתנה ולכן הטיפוסים בשני הקבצים יכללו הגדרות שונות.

בפסקל ישנו טיפוס: Text אשר הוגדר מראש למטרה של תקשורת בין תוכניות שונות.

Java: זו שפה שמאוד מקפידה על אכיפת חוקיות הטיפוסים (strongly typed). כאשר עושים include השפה כן מוודאת ששתי ההגדרות הן זהות.

<u>שקילות לפי הגדרה:</u> שני טיפוסים הם שקולים אם יש להם את אותה ההגדרה.

דוגמא: בפסקל בגרסה חדשה יותר, למרות שמות שונים אם ההגדרה זהה הקומפיילר יאשר את השימוש.

```
TYPE  T1 = File of Integer;
      T2 = File of Integer;
VAR   f1: T1;
      f2: T2;
Procedure p(Var f: T1);
....
p(f1);  (* ok *)
p(f2);  (* ok *)
```

Subtyping: ישנה אפשרות לקבל טיפוס אחר, אם הוא תת-טיפוס של השני (רק כיוון אחד של equivalence).

פסקל-דוגמא: ניתן להגדיר תת-תחום של הטבעיים. ולא נוכל למנוע מכך ש-age יהיה תת-טיפוס של height למרות שלא היינו רוצים לבלבל ביניהם.

```
type
      age = 0..120;
      height = 0..250;
```

חיסרון נוסף: לאחר כל הצבה למשתנה מטיפוס זה יש בדיקת התחום בזמן ריצה, פוגע בביצועים.

Ada- דוגמא: age ו-height הם תת-טיפוס של integer בלבד ולא אחד של השני.

אבל ניתן להגדיר תת-טיפוס של height למשל:

```
type
   age = derived integer range 0..120;
   height = derived integer range 0..250;
subtype child_height is height range 0..120;
```

ואז גובה של ילד הוא תת-טיפוס של גובה של בנאדם. זהו צמצום הטיפוס.

הערה: ב-C++ יש subtyping מסוג ירושה.

Branding: לפעמים נרצה ששני ערכים מאותו טיפוס לא יהיו שקולים. על-מנת להבדיל ביניהם, נותנים להם שמות שונים (tag).

דוגמא: רשומה בעלת שדות שונים:

```
User = Record
   name: String;
   address: String;
end
```

Strength:

החוזק מתייחס לרמה שבה הקומפיילר מבטיח כי הוא אוכף את חוקי ה-typing של השפה.

Strongly typed: לא ניתן להפעיל על ערך פעולה אשר לא מיועדת לפעול עליו.

שפות לדוגמא: ML, Eiffel, Modula, Java.

Weakly typed: ערכים אמנם מזוהים עם טיפוסים אבל אפשרי להתעלם מכך ולהתייחס לערכים אחרת או לשנות את הטיפוס שלהם.

שפות לדוגמא: C.

פסקל: שפה שהיא יחסית strongly typed, יש בה כמה חריגות שמאפשרות לעקוף את המנגנון:

Variant records: מאפשר שימוש ב-float למרות שמצפים ל-long.

```
union {
  float f;
  long l;
} d;
d.f = 3.7;
printf("%ld\n", d.l);
```

type casting: מאפשר פעולות חשבון בין מצביעים.

```
long i, j, *p = &i, *q = &j;
long ij = ((long) p) ^ ((long) q));
```

Time of checking:

Statically typed: הבדיקה נעשית בזמן הקומפילציה. לכל משתנה ולכל פרמטר אקטואלי יש טיפוס.

שפות לדוגמא: C, Pascal, Eiffel, ML, Java.

יש להצהיר בזמן כתיבת התוכנית מהו הטיפוס של כל ערך. אמנם בעיות רבות קשה למנוע מראש, אך ניתן למנוע מראש שגיאות זמן ריצה של typing. וכדאי מאוד לעשות כן.

יתרונות של בדיקה סטאטית:

- מונע שגיאות זמן ריצה (מספר לא נכון של פרמטרים, טיפוס לא נכון וכו').
- גילוי שגיאות מוקדם ככל האפשר חוסך בעלויות של פיתוח.
- ה-object code פחות מנופח- לא מלא בהגדרות הטיפוסים.
- זמן הריצה קצר יותר כי אין בדיקת טיפוסים לפני ביצוע של כל פעולה.

מגבלות של בדיקה סטאטית:

ישנן כמה פעולות מאוד הגיוניות שלא ניתן לוודא ע"י המנגנון הזה:

1. חלוקה ב-0 או בעיות מתמטיות אחרות. 2. גישה לא חוקית לאינדקס של מערך.

Dynamically typed: הבדיקה נעשית בזמן ריצה. למשתנים ולביטויים אין טיפוס המזוהה איתם. רק לערכים יש טיפוסים ובדיקתם נעשית סמוך לביצוע של כל פעולה.

שפות לדוגמא: Smalltalk, Prolog, APL, Awk, Python, Shell, VB.

"Postmortem typing": שפות שלא עושות שום בדיקת טיפוסים גם בזמן ריצה. אם התוכנית עושה טעות, היא פשוט ממשיכה לרוץ איתה. זאת בניגוד לשפות אשר מבצעות בדיקת שגיאות בזמן ריצה, שעוצרות כאשר יש שגיאה.

משתנים פולימורפיים: יכולים להכיל ערכים מכל טיפוס שהוא.

דוגמא: כאשר הנתונים יכולים להינתן בכמה דרכים ורוצים שהנתונים יוחזקו ע"י משתנה אחד:

```
procedure ReadLiteral(var item);
begin
   read a string of nonblanks;
   if the string constitutes an integer literal
   then
      item := numeric value of the string
   else
      item := the string itself
end
```

חסרונות של בדיקה דינאמית:

- בזבוז במקום: יש לשמור לכל ערך את הטיפוס שלו בזמן ריצה (ב-Java לכל אובייקט יש 8 בתים המיועדים רק עבור בדיקות ה-typing).
- איטי יותר: הבדיקה שנעשית בזמן ריצה מאטה את הקצב של התוכנית.
- באגים: ניתן למנוע בעיות רבות ע"י בדיקת הטיפוסים בזמן קומפילציה.

Mixed typing: חלק מהדברים נבדקים בזמן קומפילציה והשאר נבדק בזמן ריצה.

פסקל: רוב הדברים נבדקים בזמן קומפילציה, חריגה מגבולות המערך נבדקת בזמן ריצה.

Responsibility for tagging:

Explicit typing: המתכנת אחראי להצהיר על הטיפוס עבור כל משתנה או אובייקט אחר.

Implicit typing: הקומפיילר מסיק בעצמו את הטיפוס לפי אופן השימוש באובייקט.

שפות לדוגמא: ML (אך למתכנת מותר להוסיף הצהרות על-מנת להגביל את הטיפוסים).

Semi-implicit typing: הטיפוס נקבע לפי מבנה שם המשתנה. גם כאן אין צורך בהצהרת הטיפוס.

FORTRAN: המשתנים המתחילים באחת מתוך 5 אותיות הם integers. כל האחרים- real.

Perl, Basic: סיומות כמו: $,% קובעות האם המשתנה הוא מספר או מחרוזת.

Flexibility:

- מערכת לא אמורה לייצר הודעות שגיאה בתוכניות אשר לא יעשו שגיאות typing בזמן ריצה.
- מערכת אמורה לאפשר למתכנת להשתמש באותו הקוד לדברים שונים.

גמישות יתר: המערכת לא מודיעה למשתמש על שגיאות טיפוסים שהוא עושה.

חוסר גמישות: המערכת מונעת code reuse, בעיקר בפרוצדורות.

פסקל: מערכת מאוד לא גמישה. אפילו שתי הגדרות זהות לחלוטין אינן שקולות

דוגמא: פונקציה המקבלת a: array[1..300] of T לא תוכל לקבל את הדברים הבאים:

- **Arrays of real:** Procedure body and declaration has to be repeated with `T1=Real`
- `array[1..299] of T:` Array is too small.
- `array[1..500] of T:` Array is too big.
- `array[0..299] of T:` Mismatch in indices.
- `array[1..300] of T:` No name equivalence!!!!

בעקבות זאת, השפות שבאו אחרי פסקל הוסיפו גמישות למערכת:

C: weak typing- ניתן בקלות "לכופף את החוקים".דוגמא לפונקציה שיודעת למיין ערכים מכל טיפוס:

```
int qsort(char *base, int n, int width,
          int(*cmp)())
```

Smalltalk, Python: Dynamic typing, מתגבר על בעיות גמישות מורכבות.
Ada, Java, C++: פולימורפיזם ותכנות גנרי.
Go: המערכת לא מאוד גמישה.

שימושים נוספים למערכות טיפוסים:

- Java: לכל אובייקט יש גם רשימת טיפוסי exception שמותר לו לזרוק.
- Haskell: המערכת בודקת איזה פונקציות מבצעות פעולות I/O.

סיכום- איזו מן הגישות עדיפה?

Light-headed: לשפת סקריפטים אשר מיועדת לדברים קטנים שלא ייעשה בהם שימוש פעמים רבות, ואשר יעילות הקוד וזמן הריצה אינם כה חשובים:

- Weak typing: כדי לאפשר יותר פעולות.
- Dynamic typing: כדי לאפשר יותר גמישות.
- Semi-implicit typing: כדי לחסוך בזמן התכנות (אין צורך בהצהרה על טיפוסים).

Software engineering oriented: תוכניות אשר מפותחות ע"י מספר אנשים ומיועדות למערכות גדולות שירוצו מספר רב של פעמים, כאשר יעילות הקוד וזמן הריצה כן חשובים:

- Strong typing: כדי להפחית בשגיאות.
- Static typing: יעילות של הקוד וגם מאפשר קוד ברור יותר.
- No semi-implicit typing: למנוע יצירתם של משתנים שלא לצורך.
- Flexibility: כדי לאפשר למתכנתים להתרכז רק במה שחשוב.

פקודות

פקודה: ביטוי בתוכנית שמבוצע כדי לעדכן משתנים או כדי להשפיע על מצב התוכנית.

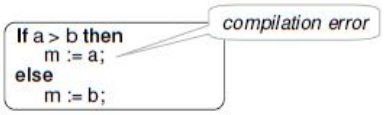
Misnomer: הצהרות.

Structured: נכנסים אליהם במקום אחד ויוצאים מהם במקום אחר.

- פקודה פרימיטיבית: לא מתפרקת לפקודות פשוטות יותר- skips, assignments.
- פקודה מורכבת: נבנית מפקודות פשוטות יותר.

Unstructured: נכנסים אליה במקום אחד ויוצאים בכמה מקומות (לדוגמא: if).

Separators:



Separatist approach: גישה מאוד מבלבלת ומטעה.

פסקל: יש ; רק בחלק מהמקרים, למשל ב-if statement לא שמים ; אחרי ה-then וכן שמים אחרי ה-else. מפריע מאוד להעביר פקודות למקומות אחרים.

Terminist approach: C: כל פקודה צריך לסיים ב-;.

Hybrid approach: סימן סיום הוא אופציונאלי.

C-דוגמא: מותר לשים , אחרי האיבר האחרון אבל לא חייבים. זה מקל על פירסור אוטומאטי של המידע באופן זהה ללא מקרי קצה עבור האיבר האחרון. `int primes[ ] = {2, 3, 5, 7, 11, 13, };`

Lax, modern approach: כל ה-separators הם אופציונאליים.

- הקומפיילר עובד קשה יותר על-מנת לפרש את התוכנית.
- JavaScript: הקומפיילר יכול לפרש מעבר שורה כסיום פקודה בטעות.

Building Blocks:

Skips: הפקודה הריקה, פקודה שלא עושה כלום.

C: ; **פסקל:** empty.

דוגמא: העתקת מחרוזת אחת לשנייה אבל גוף הלולאה לא עושה כלום- while(*t++ = *s++);.

Assignments: פקודות הצבה.

דוגמא: Gnu-C מאפשרת הצבה מסוג כזה (בניגוד ל-C): `(if ... then m else n) := 7`

Multiple assignment: C- השמה לכמה משתנים בו-זמנית `m := n := 0`

Simultaneous assignment: ML- הצבת ערכים שונים למשתנים שונים: `m, n := n, m`

Update assignment: שינוי ערך תוך כדי ההשמה

```
n += 1
add 1 to N
```

גישה למשתנה:

גישה יכולה להוות אחד מהשניים:

1. ערך המשתנה.
2. רפרנס למשתנה.

C, Pascal: משמעות הגישה נקבעת לפי ההקשר, אבל אופן הכתיבה הוא זהה.

ML: יש הבדל באופן הכתיבה. Val n=רפרנס ל-n, n=הערך של n.

קריאות לפונקציות:

פרמטר פורמאלי: הפרמטר שמופיע בהגדרת השגרה.

פרמטר אקטואלי: הפרמטר שמועבר בפועל להפעלת השגרה.

תוצרי לוואי: שינוי רפרנסים שהועברו אליה, שינו משתנים גלובליים, שינוי משתנים סטטיים.

בקרת זרימה:

Sequential commands: פקודות מורכבות. סדר ביצוע הפקודות הוא חשוב.

Collateral commands: סדר ביצוע הפקודות אינו משנה. `m := 7, n:= n + 1`

חישוב הוא **דטרמיניסטי** אם הוא תמיד מחזיר את אותה התוצאה.

אם בפקודה קולטרלית, סדר החישוב לא ישפיע על התוצאה אז זה עדיין דטרמיניסטי.

דוגמא: A+B. ישנן שפות שבהן מוגדר הסדר של החיבור וישנן כאלה שזה לא מוגדר בהן.

C: סדר החישוב של: *,+ אינו דטרמיניסטי.

דוגמא: D(x,y,z). את מי מהם מחשבים קודם? ב-C: z ואז y ואז x. פסקל: לא מוגדר הסדר.

Conditional commands: פקודה המכילה כמה תת-פקודות אשר רק אחת מהן תתבצע.

הערה: switch לא עונה להגדרה הזו כי לא מובטח שיהיה break ואז יותר מפקודה אחת תתבצע.

פקודה מותנית קולטראלית: בדיקת כל התנאים (באיזשהו סדר), ביצוע של אחד התנאים שמתקיים.

ML: מנגנון ה-pattern matching דומה לזה.

פקודה, ביטוי וההבדלים ביניהם:

ביטוי: משהו שמחשבים אותו והוא מחזיר ערך, לכל ביטוי יש טיפוס. בדרך-כלל אין תוצאות-לוואי.
דוגמאות: פונקציה, ביטוי מותנה.

פקודה: פעולה לביצוע. דוגמאות: פרוצדורה, פקודה מותנית.

דוגמא: A=B.

פסקל: מחזיר true אם A שווה ל-B ו-false אחרת.

C: מחשב את A, מחשב את B, מחזיר את B וכתוצר לוואי מציב את B ל-A.

דוגמא: בפסקל ייתכן כי חישוב של ביטוי יביא לתוצאת לוואי: חישוב ביטוי שמכיל קריאה לפונקציה שתשנה ערך גלובלי.

הערה: לולאות זה משהו ייחודי לפקודות.

Iterative commands:

Indefinite: לא ידוע מראש מהו מספר האיטרציות- while, repeat… until.

Definite: יש חסם על מספר האיטרציות.- for…to, for…downto.

C: לולאת for היא Indefinite ושקולה לגמרי ללולאות while ו-do-while.

פסקל: קובע ערך הפרמטרים רק בהתחלה ואם הם משתנים, מספר האיטרציות לא משתנה.

Collateral loops: סדר ביצוע הפעולות בגוף הלולאה לא מוגדר. שימושי לתכנות מקבילי.

משתני בקרה בלולאות:

משתנה האינדקס של הלולאה. בשפות רבות עליו להיות מוגדר לפני הלולאה.

פסקל: צריך להיות מוגדר מחוץ ללולאה, ערכו לא מוגדר מחוץ לה ואסור לשנותו במהלך הלולאה.

C++: ניתן להגדירו בתוך ה-for, והוא לא ימשיך להתקיים מחוץ לגבולות הלולאה.

הערה: ב-C האינדקס שהוגדר המשיך להתקיים גם מחוץ ללולאה.

Ada: מתייחסים למשתנה כאל קבוע בכל איטרציה ולא ניתן לשנות את ערכו, לא קיים מחוץ ללולאה.

ביטויים עם side-effects:

ביטויים המשמשים כסוג של פקודה.

דוגמא: התקדמות המחוון הפנימי של הקובץ כתוצאה מהקריאה הראשונה (מה שיביא את הקריאה השנייה לקרוא את התו שאחרי ולא את אותו התו).

```
if getchar(f) = 'F' then
     gender = female
else if getchar(f) = 'M' then
     gender := male
```

פסקל: לפונקציה יש side-effects כי לא ניתן למנוע ממנה לקרוא לפרוצדורה.

ML: שפה התומכת ב-side effects, אין בה הבדל בין ביטויים לפקודות.

הערה: בדרך-כלל הערך של השמה הוא הערך שההשמה הציבה במשתנה. ב-ML הערך של השמה הינו-() (טאפל בגודל-0).

Command expressions: ביטוי שכולל בתוכו פונקציה אשר מחשבת את הערך שהביטוי מחזיר.

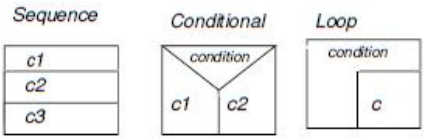
דוגמא: בשפה דמוית פסקל- הביטוי כולל לולאה המחשבת פולינום. הפקודה **yield** מחזירה את תוצאת החישוב.

```
var  p,r:Real;  i:Integer;
r := EvalPoly(
   begin
    p := a[n];
    for i:=n-1 downto 0 do p:=p*x+a[i];
    yield p;
   end;
   );
```

תכנות מבני:

תכנות המבוסס על פקודות המתבצעות ברצף, משפטי תנאי ולולאות. לכל רכיב יש כניסה אחת ויציאה אחת.

עיקרון ה- Compositional: ניתן להבין את התוכנית כולה ע"י הבנת כל אחד מהמרכיבים בנפרד והרכבה שלהם ביחד לאחר-מכן.



דיאגרמות Nassi-Shneiderman: שיטה לתיאור גרפי של תוכניות מבניות המדגימות את העיקרון הנ"ל.

כל תוכנית מבנית גם ניתנת לתיאור ע"י תרשים זרימה.

אך כמובן, לא כל תרשים זרימה מייצג תוכנית מבנית!

התיאוריה של תכנות מבני: ישנו אלגוריתם שבהינתן תרשים זרימה, יודע לבנות ממנו תוכנית מבנית.

Sequencers:

פקודות שמשנות את סדר הביצוע הרגיל של התוכנית.

Jumps:

קפיצה מפורשת למקום אחר (goto). יוצר תוכניות "ספגטי".

תוכניות שמשתמשות בקפיצות הן קשות להבנה. זה נחשב מכוער אך תמיד הרשו את הפעולה באיזשהו אופן כי ישנה אמונה שיש דברים שלא ניתן לעשות בלי זה.

Labels: סימון בתוכנית שאליו ניתן לקפוץ. קיימים שני סוגים:

בעל משמעות: מילה- מזהה כלשהו בעל משמעות (C, Assembly).

מספר: מספר השורה בקוד (Pascal, Basic, FORTRAN).

Labels checking: האם יש קפיצה לכל אחת מהתוויות שהוגדרה בתוכנית?

דוגמא: תוכנית ובה הערה שנמצאת בתוך תווית. ולכן הקוד מתקמפל ב-C:

```
int main() {
  http://www.cs.technion.ac.il/234319;
  printf("Page loaded successfully\n");
}
```

מגבלות על קפיצות:

- רק בתוך אותו בלוק (FORTRAN).
- רק בתוך הבלוק ובבלוקים שלמעלה בהיררכיה (Pascal, Algol).
- קפיצה לכל מקום (אסמבלר).

Escapes:

מסיים ביצוע פעולה של פקודה מורכבת. מעין jump אבל מובנה יותר.

מאפשר כניסה אחת ומספר יציאות.

- יציאה מהלולאה הנוכחית: break (C).
- יציאה מכל הלולאות החיצוניות: exit L (Ada), break L (Perl).

- סיום פונקציה: return (C).
- יציאה מכל התוכנית: exit (למרות שזו אינה פקודה בשפה).
- סיום מיוחד: break בתוך switch ב-C.

לולאות מקוננות-דוגמא: ב-Java ניתן לעשות escape ללולאה חיצונית באופן מפורש:

```
A: for() {
      B: for() {
            C: for() {
                  Break A;
            }
      }
}
```

פיתרון בשפות אחרות: הכנסת הלולאות לפונקציה, וכאשר רוצים להפסיק את כולן, עושים מהפונקציה- return.

Continue: ממשיך לאיטרציה הבאה של הלולאה.

Exceptions:

לפעמים מתבצעים מקרים חריגים ורוצים שהתוכנית תדע להתאושש ממקרים אלה.

אסטרטגיות לטיפול בבעיות:

- ביצוע חוזר- בדרך כלל אסטרטגיה לא טובה.
- עצירת התוכנית בכל שגיאה.
- Resumption: המשך ביצוע הפקודה כרגיל לאחר פעולת ה-handler.
- Explicit: הפרוצדורה עושה return עם ערך השגיאה.
- Long jump: בבת-אחת חוזרים אחורה למקום שבו יודעים לטפל בבעיה.

  פסקל: בעזרת goto.

  C: בעזרת הפקודות setjmp, longjmp המאפשרות שמירת הסביבה הנוכחית ושחזור שלה לצורך קפיצות למקומות מוקדמים יותר בתוכנית.
- הצמדת שגרה לטיפול, לפרוצדורה עצמה.

ערכי exceptions:

- **פסקל:** Unit זו התוצאה של קפיצת goto לפרוצדורה המכילה.
- **C++:** יכול להיות כל טיפוס שהוא.
- **Java:** כל תת-טיפוס של המחלקה Throwable.

Coroutines:

שגרה המסוגלת לעצור בנקודה מסוימת בביצוע, לחזור למקום הקריאה אליה, וכאשר תיקרא שוב, להמשיך מהמקום אשר בו הפסיקה קודם-לכן.

כך השגרה מסוגלת למשל להחזיר מספר רב של ערכים לפי סדר מסוים.

דוגמא: preorder(T)- יחזיר בכל-פעם צומת אחר בעץ לפי סדר preorder.

דוגמא: במהלך איטרציה, ניתן להשתמש באיטרטור כאינדקס בקרה, יחזיר בכל פעם ערך אחר.

איטרטור מחזיר ערך בשימוש ב-yield:

```
iterator A() : integer;
   begin
    yield 3;
    yield 4;
   end;
…
for i:=A() do   -- ranges over 3 and 4
   …
End;
```

פולימורפיזם

מונומורפיזם ופולימורפיזם:

מונומורפיזם: לכל ביטוי יש טיפוס יחיד.

דוגמאות: ליטרלים, קבועים, משתנים, פרמטרים, תוצאות של פונקציה.

פסקל: מערכת הטיפוסים היא בעיקרה מונומורפית:

- יש להצהיר מהו הטיפוס המדויק של כל פרמטר פורמאלי ושל כל ערך שחוזר מפונקציה.
- כל פרוצדורה ופונקציה המוגדרת היא מונומורפית.

תכונות פולימורפיות של פסקל:

- פונקציות built-in: read, write הן פונקציות פולימורפיות- יודעות לקבל מספר טיפוסים שונים.
- הפונקציה eof יודעת לקבל קובץ מכל סוג ולהחזיר לגביו true או false.
- האפשרות ל-subrange מאפשרת ירושה.
- הקבוע nil: מסוגל להצביע לכל טיפוס.

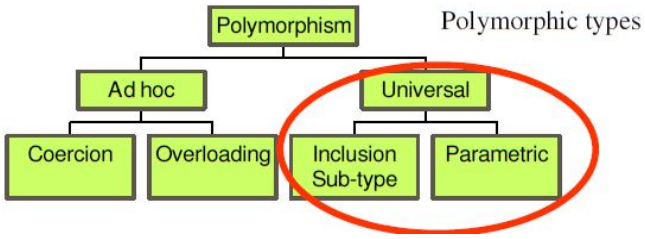
```
type CharSet = set of Char;

function disjoint (s1, s2: CharSet) :Boolean;
  begin
    disjoint := (s1 * s2 = [])
  end
```

The * operator in Pascal is *polymorphic*. It can be applied to any two sets of the same kind of elements

דוגמא: פסקל לא מאפשרת פולימורפיזם, ולכן במקרה הזה, פונקציה אשר בודקת האם שני סטים הם זרים, יהיה צורך לכתוב אותה עם קוד זהה לכל אחד ואחד מהטיפוסים עליהם נרצה להפעיל את הפונקציה.

פולימורפיזם: היכולת של אובייקט להיות מזוהה עם כמה טיפוסים שונים.

סוגים שונים של פולימורפיזם:



פולימורפיזם Ad hoc: מסוגל לעבוד עבור מספר טיפוסים קטן וידוע מראש. לא תמיד ביצוע הפעולה הוא אחיד בין הטיפוסים השונים. לתמיכה בטיפוס חדש צריך באופן ידני להרחיב את הפולימורפיזם כדי לתמוך גם בו.

דוגמא: פעולת חיבור בפסקל מוגדרת ל-real, integer וכו' אבל לא תהיה מוגדרת למספרים מרוכבים (אם המתכנת יגדיר כזה טיפוס). בפסקל לא ניתן להרחיב את הגדרת האופרטור לטיפוסים נוספים.

פולימורפיזם אוניברסאלי: מסוגל לעבוד על מספר אינסופי של טיפוסים, אבל לא בהכרח על כל הטיפוסים בעולם. בדרך-כלל יש מכנה משותף בין כל הטיפוסים עליהם הפולימורפיזם עובד.

דוגמא: בפסקל אופרטור הכפל כחיתוך של קבוצות (בדוגמא הקודמת) עובד רק על טיפוסים שניתן לייצר מהם קבוצות. אך זהו עדיין מספר אינסופי של טיפוסים.

Overloading:

שימוש באותו השם לכמה פונקציות שונות. בכל קריאה ניתן לקבוע לאיזו מן הפונקציות מתייחסת הקריאה.

- `VAR s: Set of Char`: type declaration
- `Case month of …`: conditional statement

דוגמא: פסקל: שימוש במילה of במשמעות שונה:

לכל הגדרות הפונקציה או האופרטור יש את אותו השם.
לפעמים ההבדל הוא בסוג הטיפוסים ולפעמים במספר הפרמטרים.

Pascal, C, ML: רק אופרטורים ומזהים built-in בשפה יכולים להיות Overloaded.

C++: המתכנת יכול להגדיר overloading בעצמו.

סיבה נוספת ל-overloading: לפעמים לפונקציה יש שם מוצלח ונרצה להשתמש בו לכל המימושים השונים. **דוגמא:** הפונקציה max

```
double max(double d1, double d2);
char max(char c1, char c2);
char *max(char *s1, char *s2);
const char *max(const char *s1, const char *s2);
```

Overloading תלוי הקשר:

Context-independent: הפונקציה מזוהה לפי הפרמטרים האקטואליים בלבד (C++).

Context-dependent: כאשר לא ניתן להכריע בין שתי פונקציות רק על סמך הפרמטרים המועברים לפונקציה, נכנס כפקטור נוסף גם ה-context, ההקשר של הטיפוס המוחזר והסביבה שלו (Ada).

Java, C#, Go: אין אופרטור overloading.

Scope Hiding: מזהה המוגדר ב-scope פנימי מסתיר את המזהה של ה-scope החיצוני.

ההבדל מ-overloading: משמעות אחת מסתירה באופן זמני את המשמעות האחרת בעוד שב-overloading שתי המשמעויות קיימות בו-זמנית.

```
static long tail;

…

int main(int argc, char **argv)
{
   char **tail = argv+argc-1;
   …
}
```

Coercions:

המרה implicit מטיפוס אחד לטיפוס אחר.

דוגמא: בפסקל יש מיפוי מ-real ל-int בשני הכיוונים, ולכן ניתן לקרוא לפונקציה sqrt(n) המוגדרת על real גם עם integer, אין צורך להגדיר את הפונקציה פעמיים.

C++: ניתן להגדיר המרות ע"י המתכנת.

Explicit: לפעמים במקרה שבו נעשה coercion אוטומאטי זה יגרום לתוצאות לא רצויות. ולכן הוספת explicit אומרת לקומפיילר לא לבצע coercions.

Go: לליטרלים יש coercion אוטומאטי לכל הטיפוסים.

```
class Rational {
   public:
explicit Rational(double);
      operator double(void);
      ...
} r = 2;
```

דו-משמעות בשימוש ב-coercion:

לעיתים נדרשות מספר המרות כדי לעבור מטיפוס אחד לאחר. המסלול של ההמרה (ולעיתים ייתכן יותר מאחד) יכול להשפיע על התוצאה הסופית.

דוגמא:

$$\texttt{unsigned char} \to \texttt{char} \to \texttt{int} \to \texttt{long} \to \texttt{double} \to \texttt{long double}$$
$$\texttt{unsigned char} \to \texttt{unsigned} \to \texttt{unsigned long} \to \texttt{long double}$$

במקרה זה המסלול השני הוא עדיף כי במסלול הראשון ייתכן איבוד של מידע, עקב מעבר בטיפוסים המיוצגים ע"י מספר ביטים קטן יותר.?

הערה: גרף ההמרות האפשרי גם יכול להיות מעגלי.

תחרות בין coercions לבחירת הפונקציה:

ב-C++ כאשר ישנן כמה פונקציות אפשריות, כל coercion מקבל "דירוג" לפי הסולם הבא:

1. אין צורך בהמרה או צורך בלתי נמנע להמרה: array → pointer, T → const T.
2. הגדלת גודל הטיפוס (אין איבוד מידע): short → int, float → double.
3. המרה סטנדרטית בשפה: int → double, double → int.
4. המרות שהוגדרו ע"י המתכנת.

הבחירה בין הפונקציות השונות נעשית ע"י "תחרות" בין הפונקציות. המנצח חייב להיות טוב כמו האחרים בכל הפרמטרים וטוב יותר מהאחרים בלפחות פרמטר אחד.

אם אין מנצח מובהק בתחרות, הקומפיילר מודיע על שגיאה.

פולימורפיזם פרמטרי:

ביצוע פעולה זהה עבור מספר רב מאוד של טיפוסים (שאינם בהכרח קשורים אחד לשני).

פסקל-דוגמא: לולאת for היא מבנה פולימורפי שמסוגל לעבוד על כל טיפוס דיסקרטי.

האופרטור > הוא פולימורפי לכל טיפוס שהוגדר ע"י enum.

```
for m := January to December do
  for d := Saturday downto Sunday do
    case suit of
      Club, Heart:
        suit := succ(suit);
      Diamond, Spade:
        if suit < Heart then
          if ord(m) < ord(d) then
            suit := pred(suit);
  end;
```

פולימורפיזם פרמטרי בפסקל:

- פעולות על סטים: -,+,* יעבדו לכל סט שהוא:
  - type is Set(τ)xSet(τ)→Set(τ)
- הפעולות להקצאה ושחרור זיכרון:
  - type is Pointer(τ)→Unit
- הערך nil שיכול לקבל טיפוס של כל מצביע:
  - Type is Pointer(τ)

הערה: אמנם ישנם parametric polymorphism שהם built-in בשפה, אבל בפסקל אין אפשרות להוסיף ולהגדיר נוספים ע"י המתכנת.

C-דוגמא: ההמרה מ-*void לכל טיפוס מצביע אחר ובחזרה היא פולימורפיזם אוניברסאלי כי היא קיימת לכל סוגי הטיפוסים ומתבצעת באופן דומה עבור כולם.

C++-דוגמא: ניתן לכתוב טפלייט שמקבל פרמטר שהוא בעצמו טיפוס. הביטוי max הוא פולימורפי אוניברסאלי אך הוא עובד רק עבור טיפוסים שמוגדר עבורם אופרטור ההשוואה <.

```
template<typename Type>
  Type max(Type a, Type b)
  {
    return a > b ? a : b;
  }
…
int x,y,z;
double r,s,t;
z = max(x,y);
t = max(r,s);
unsigned long (*pf)(unsigned
long, unsigned long) = max;
```

המרות מפורשות ב-C++:

< τ >static_cast: המרה מפורשת לסוגי המרות המוגדרות מראש.

< τ >Reinterpret_cast: הנחייה לפענח את הביטים שבזיכרון לפי טיפוס אחר.

< τ >Const_cast: הופך טיפוס שהוא const לכזה שהוא לא.

< τ >Dynamic_cast: הנחייה להתייחס לתת-טיפוס של הטיפוס.

אלה הן פונקציות שמעבירים להן טיפוס והן מחזירות אופרטור שממיר לטיפוס הזה. פולימורפיזם פרמטרי- לכל טיפוס יוחזר אופרטור מתאים.

ML-דוגמא: ניתן להשתמש ב-type variables בהגדרת הפונקציה. כאן ישנה פונקציה המקבלת שני פרמטרים ומחזירה את השני. יעבוד לכל שני פרמטרים מטיפוסים: σ ו- τ כלשהם.

```
fun second(x:σ, y:τ) = y
 The function is of type σ * τ →τ
 σ and τ each stand for any type whatsoever.
```

Polytypes: טיפוס המכיל משתנים שהם טיפוסים. מכל טיפוס כזה ניתן לגזור קבוצה של טיפוסים.

דוגמאות: List(τ), $s \times t \to \tau$.

Monotype: טיפוס שאינו מכיל טיפוסים. בשפות מונומורפית, כל הטיפוסים הם monotypes ורק טיפוסים built-in בשפה יכולים להיות כאלה.

פסקל: file of τ, שנעשה בו שימוש בפונקציה EOF, הוא built-in.

ML: המתכנת יכול להגדיר בעצמו: type τ pair = $\tau * \tau$.

ערכים של טיפוס polytype: מהי קבוצת הערכים החוקיים ש-polytype יכול לקבל?

הגדרה: קבוצת הערכים היא החיתוך של כל ערכי הטיפוסים שהוא יכול לקבל.

דוגמאות:

- List of τ, החיתוך הינו: הרשימה הריקה ולכן זהו הערך החוקי היחיד.

- $\tau \to \tau$ מכיל בחיתוך רק את פונקצית הזהות.
- $(\tau \to \tau) \to (\tau \to \tau)$ מכיל בחיתוך אינסוף ערכים אפשריים.
- Pair(τ)=$\tau \times \tau$, אין שום ערך בחיתוך, אין אף ערך שמתאים לכל τ שהוא.

הסקת טיפוסים:

הדרך שבה הקומפיילר מסיק מהם הטיפוסים של הערכים.

פסקל: בשפות סטטיות, יש להצהיר לכל ערך מהו הטיפוס שלו.

ML: יש מנגנון מתוחכם שיודע להסיק לבד מהו טיפוס הפונקציה. אם אין מידע מספיק כדי להחליט מהו הטיפוס, נוצרת פונקציה פולימורפית אפילו ללא הגדרה מפורשת שזהו פולימורפיזם.

דוגמא: fun id (x) = x. נוצרת פונקציה שהיא מטיפוס: $\tau \to \tau$.

Inclusion:

פולימורפיזם אוניברסאלי אשר עובד על קבוצת טיפוסים שיש ביניהם קרבה. בדרך כלל עבור טיפוס וכל תתי-הטיפוסים שלו. הפונקציה תעבוד לאינסוף טיפוסים אבל לא לכל טיפוס שהוא.

הגדרת Subtype:

1. טיפוס A הוא תת-טיפוס של B אם טיפוס A מוכל בטיפוס B.
2. טיפוס A הוא תת-טיפוס של B אם לכל ערך של A מוגדר coercion לערך של B.

דוגמאות-Built in:

פסקל: Nil הוא תת-טיפוס של כל סוגי המצביעים.

C: הערך 0 שייך לכל סוגי המצביעים.

C++: הערך *void קיימת לו המרה בשני הכיוונים לכל סוגי המצביעים.

דוגמאות- user defined:

פסקל: subranges של טיפוסים.

הערה: הקומפיילר מוסיף בדיקה בזמן ריצה, לבדיקה האם זהו הערך המתאים של התת-טיפוס. הבדיקה אף עשויה להיכשל בזמן ריצה ופוגעת בביצועים של התוכנית.

C++: ירושה.

Duck typing: באופן תיאורטי אם ב-ML הייתה ירושה, היה ניתן להסיק אותה מהמבנה של הטיפוסים (if it looks like a duck and sounds like a duck…).

- מערכת שיש בה רק Overloading ו-coercion היא מערכת טיפוסים מאוד מנוונת והם לבדם לא הופכים את המערכת להיות פולימורפית.
- **Go:** אין ירושה ואין טפלייטס. רוצים בעתיד להרחיבה ע"י הוספת פולימורפיזם פרמטרי.

Bindings

Binding: שיוך של מזהה ליישות בתוכנית.

Declaration: הפעולה המקשרת בין מזהה ליישות.

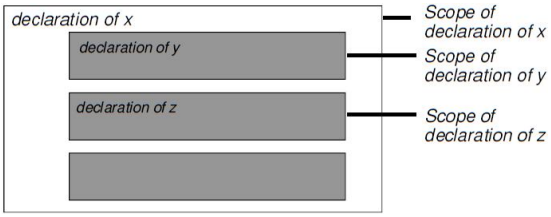
Environment: הסביבה- אוסף של bindings. כל ביטוי בתוכנית מתפרש ביחס לסביבה אליה הוא שייך.

Scope: קטע מהתוכנית שבו binding מסוים תקף.

בלוק: מבנה בתוכנית שמגביל את ה-scope של ההצהרות שבתוכו.

סוגים של Block structure:

Monolithic Block structure: כל התוכנית כולה היא בלוק יחיד. היה קיים בשפות מאוד ישנות. מאוד קשה לעבוד בצוותים על פרויקט גדול כאשר כל שמות המשתנים הם גלובליים לכל התוכנית.



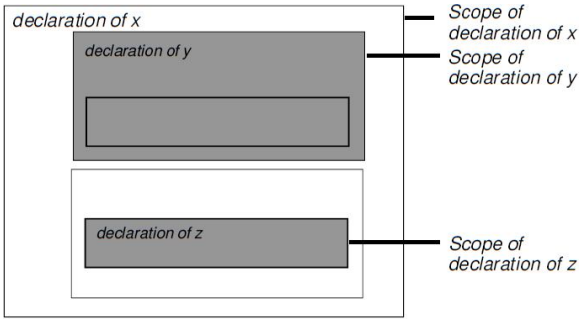
Flat Block structure: התוכנית מחולקת לבלוקים שאינם מקוננים. כל הגדרה באחד הבלוקים, מקומית לאותו הבלוק. כל הגדרה מחוץ להם, היא גלובלית. זהו מבנה יחסית נוח אבל גם הוא לא הצליח.

Nested Block structure: קיים קינון של בלוקים. לכל משתנה, ה-scope שלו זה הבלוק הקטן ביותר שמכיל אותו.

קיים בכל שפות התכנות המודרניות.

עיקרון: ניתן לעשות binding לאותו המזהה עבור יישויות שונות בבלוקים שונים.

Hiding Declarations:



בלוק חיצוני ובלוק פנימי אשר להם הגדרה בעלת שם זהה: בבלוק הפנימי, ההגדרה הפנימית תסתיר את החיצונית.

Hiding & Overloading: כאשר שפה מאפשרת את שניהם, מתי הצהרה אחת תסתיר את השנייה ומתי הן יתקיימו בו-זמנית? מאוד מסובך לקבוע זאת.. (תלוי בחוקים ספציפיים של השפה)

הערה: overloading יכול להיווצר גם בין פונקציה למשתנה.

Static & Dynamic Binding:

Static Binding: נעשה בזמן קומפילציה לפי המבנה של התוכנית. ההצהרה התקפה בכל פקודה היא זו שהכי קרובה- הבלוק המכיל הקטן ביותר.

שפות לדוגמא: Fortran, Algol, Pascal, Ada, C.

Dynamic Binding: נעשה בזמן ריצה לפי התנהגות התוכנית בזמן הריצה. ההצהרה התקפה בכל פקודה היא זו שהוגדרה אחרונה (most recent).

שפות לדוגמא: Lisp, Smalltalk.

הערות:

- Dynamic יכול לשמש להעברת פרמטרים לפונקציה כי סביבת הפונקציה נשלטת לחלוטין ע"י הפונקציה הקוראת (המקום בו ההגדרות העדכניות ביותר בטרם הפונקציה נקראה).
- Static binding עדיף כי יש יותר מניעת טעויות לפי הריצה, והוא גם יותר מובן בטרם הרצת הקוד.

דוגמא:

Static: השורה scaled:= d*s מתייחסת ל-const s = 2 המוגדר במקום הכי קרוב מבחינת מבנה התוכנית. ולכן נקבל: 2*5=10.

Dynamic: השורה scaled:= d*s מתייחסת ל-const s = 3 אשר נמצא ב-P הפונקציה הקוראת ל-scaled. ולכן נקבל: 3*5=15.

```
const s = 2;
  function scaled (d: integer): integer;
  begin
      scaled := d * s;
  end;
  procedure P;
  const s = 3;
  begin
    ... scaled(5) ...
  end;
begin
  ... scaled(5) ...
end
```

Static binding: Value is 10
Dynamic binding: Value is 15

Value is 10

סוגי הצהרות:

Definition: סוג של הצהרה אשר יוצר רק binding ולא יוצר טיפוס או משתנה חדש.

Collateral declaration: אוסף של הצהרות בלתי תלויות. לא יכולות להשתמש במזהים של הצהרות אחרות מהאוסף.

Sequential declaration: אוסף של הצהרות שיש ביניהן קשר. יתבצעו בסדר מסוים ולכן הצהרה יכולה להשתמש במה שהוגדר לפניה.

דוגמא: (ML)

```
val Pi  = 3.14159;
val TwoPi = 2 * Pi;
```

הצהרות רקורסיביות: הצהרה שיוצרת משהו חדש ומשתמשת בו לפני שיצירתו הסתיימה.

C: בכל מקום בתוכנית מותר לכתוב struct <name> *a, לא משנה מי זה <name> ולכן הכתיבה הרקורסיבית מצליחה.

```
struct Node {
  int data;
  struct Node *next;
};
```

פסקל: אין תמיכה בהגדרות רקורסיביות מלבד מקרה אחד: הגדרת טיפוס שמתייחס לפויינטר לעצמו, וכך ניתן לעקוף את הבעיה בצורה מסורבלת.

```
type
   IntList = ^ IntNode;
   IntNode = record
      head: Integer;
      tail: IntList;
   end;
```

Forward declarations: הצהרה לקומפיילר בלבד. הצהרה על משהו שיוגדר בהמשך כדי לאפשר שימוש בו בטרם יוגדר.

פסקל: בפסקל אסור להתייחס למשהו שלא הוצהר.

C: לא צריך את המילה forward כדי להצהיר מראש. ההצהרה מראש נועדה בין השאר גם עבור חלוקה לקבצי h שבהם יש רק הצהרות של הפונקציות ללא המימוש.

```
procedure Bar; forward;

procedure Foo;
begin
   ...
   Bar;
   ...
end;
procedure Bar;
begin
   ...
   Foo;
   ...
end;
```

סוגים שונים של בלוקים:

Block Commands: פקודה המכילה בתוכה הצהרות. ה-binding של ההצהרות תקף רק בתוך הפקודה עצמה.

פסקל: ההגדרות בבלוק עושות overriding להגדרות שמחוץ לפקודה. חייבים להגדיר את המשתנים בהתחלה ולא בסמוך לשימוש בהם.

C: בכל תחילת בלוק ניתן להגדיר משתנים, אפילו בבלוק if למשל.

Block Expressions: ביטוי המכיל הצהרות שתקפות רק לביטוי. קיים בשפות פונקציונאליות כמו ML, לא קיים ב-C.

Block Declaration: הגדרה של הצהרות שתקפות רק בתוך הצהרה אחרת, עוזרות לחישוב ערכו של הבלוק.

```
let
  val
    s = (a + b + c) * 0.5
  in
    sqrt( s * (s-a) * (s-b) * (s-c) )
end
```

```
local
  fun multiple(n: int, d: int) = (n mod d = 0)
in
  fun leap(y: int) = (multiple(y, 4)
    andalso not multiple(y, 100))
    orelse multiple(y, 400)
end
```

Abstraction

הפשטה: איתור תכונות משותפות הרלוונטיות לעצמים שונים ולכידתם באובייקט אחד.

רמות של הפשטה:

כל שפת תכנות היא בעצמה אבסטרקציה של שפת מכונה.

Data: מה מבני הנתונים בתוכנית מייצגים ואיזה פעולות ניתן להפעיל עליהם ולא כיצד הם ממומשים.

Control: הסדר שבו הפקודות מתבצעות.

Procedural: מה השגרה עושה, ולא כיצד היא עושה זאת.

מוטיבציה: כאשר ישנן חזרות רבות בקוד של אותו הקונספט, במהלך אחד מהשינויים שהקוד יעבור ייתכן כי נבצע את השינוי רק באחד מהמקומות הנדרשים ולא בכולם.

Abstraction mechanism: מבנה בשפת התכנות שמאפשר למתכנת ללכוד את האבסטרקציה.

מנגנוני אבסטרקציה:

מאקרו: הדרך הפרימיטיבית ביותר ללכוד אבסטרקציה. אמנם אין חיסכון כי הקוד מוחלף בכל קריאה בקוד של המאקרו אבל יש חיסכון בכתיבה, וגם המימוש מרוכז במקום אחד.

רשימות-דוגמאות:

Java: for each - בחלק מהשפות יש מנגנון אבסטרקציה המאפשר מעבר על כל אברי רשימה.

```
List<ElemType> list = ...
for (ElemType e : list) {
  ... work with e ...
}
```

Python: חילוץ תת-רשימה העונה על פרדיקט, מתוך רשימה.

```
[p.id for p in people]
[p for p in people if p.age > 13]
```

פונקציה:

האבסטרקציה הבסיסית ביותר- החישוב המגולם ע"י הפונקציה יתבצע בכל פעם אבל המימוש נכתב רק פעם אחת.

אבסטרקציה פונקציונאלית: אבסטרקציה של ביטוי- משהו שמחשב ומחזיר ערך.

אבסטרקציה פרוצדוראלית: אבסטרקציה של פקודה-משהו שמעוניינים ב-side effects שלו.

האפשרות להעביר פרמטרים שונים לאבסטרקציה בכל פעם, הופכת את העניין למאוד מועיל.

הערה: קצת קשה לעשות את האבחנה בין שני הסוגים ברוב השפות, כי הפונקציות כן יכולות לבצע side effects.

ML: לאובייקטים בחוץ אין מצב ולכן לא ניתן לשנות אותו ולפונקציה אין side effect.

```
fun power(x: real; n: int) =
 (* assume that n > 0 *)
   if n = 1 then
      x
   else
      x * power (x, n-1)
end
```

פונקציה-הגדרה: מיפוי בין ארגומנטים לתוצאה.

$$\texttt{function}\ I(FP_1; \ldots; FP_n)\ \texttt{is}\ E$$

ML: יש הבדל בין יצירת הפונקציה לבין binding של מזהה עבורה. ניתן לעשות אותם בנפרד.

ברוב השפות האחרות, יש להגדיר שם לפונקציה על-מנת להגדיר את מימושה.

Declaration:

אבסטרקציה שלא מבצעת חישוב אלא מבצעת הגדרה. מאפשר להעביר פרמטרים שונים המשנים את ההגדרה.

C++: מנגנון ה-templates. **Ada, ML, Java:** מנגנון ה-Generics.

התייחסות למשתנים:

```
fun circum(r: real) = 2 * pi * r;
```
formal parameters

Selector: מנגנון שבהינתן פרמטרים, מחליט איך להתייחס אליהם.

פרמטר פורמאלי: השם של הפונקציה שנקראת לפרמטר שהועבר אליה.

```
circum(1.0);
circum(a+b);
```
actual parameters

פרמטר אקטואלי: הפרמטר של הסביבה הקוראת שנשלח בפועל אל הפונקציה.

מה ניתן להעביר כארגומנט?

פסקל: ערכים פרימיטיבים, ערכים מורכבים, מצביעים, רפרנסים למשתנים, אבסטרקציות של פונקציות ופרוצדורות.

ML: ערכים פרימיטיבים, ערכים מורכבים, רפרנסים למשתנים, אבסטרקציות של פונקציות.

מנגנונים להעברת פרמטרים:

בכל העברה של פרמטר מתבצע חיבור בין שני חלקים בתוכנית, ויש כמה מנגנונים לכך.

מנגנונים מבוססי העתקה:

By value: בכניסה נעשית השמה מהפרמטר האקטואלי לפורמאלי ע"י העתקה. לאחר-מכן הם ב"ת.

By result: בכניסה לא נעשית השמה, אלא ביציאה, מהפרמטר הפורמאלי אל האקטואלי.

By value-result: בכניסה נעשית השמה מהאקטואלי לפורמאלי וביציאה נעשית השמה בחזרה לפרמטר האקטואלי.

מנגנונים מבוססי הגדרה:

הפרמטר הפורמאלי מצומד לפרמטר האקטואלי.

By constant: הפונקציה מתחייבת לא לשנות את ערך הפרמטר הפורמאלי במהלך הריצה. מאפשר לקומפיילר לעשות כל-מיני אופטימיזציות.

By reference: בכניסה נעשית פעולה של referencing, הפרמטר הפורמאלי הופך לרפרנס לפרמטר האקטואלי. כל פעולה בגוף הפונקציה משפיעה על הפרמטר האקטואלי.

Procedural/functional: העברת מצביע לאבסטרקציה עצמה (פונקציה או פרוצדורה). כל קריאה לפונקציה היא קריאה עקיפה לאבסטרקציה שהועברה אליה.

By name: למעשה מועבר "השם הסימבולי" והחישוב שלו ערכו נעשה רק לפני שימוש.

דוגמא: בהרצת הקוד שמימין, לפונקציה f מועבר הפרמטר a[i] ולכן כאשר ערכו של i משתנה בתוך הפונקציה, יודפס a[0] ולא a[1]0.

```
f(by name in b) {
    print b; //a[1]=4
    i--;
    print b; //a[0]=3
}
```

```
i = 1;
int a[3];
a[1] = 4;
a[0] = 3;
f(a[i]);
```

Aliasing:

לאותו משתנה יכולים להיות כמה שמות שונים.

בעיה: יכול לקרות מצב שבו שני משתנים בעלי שמות שונים המועברים לפונקציה הם אותו המשתנה.

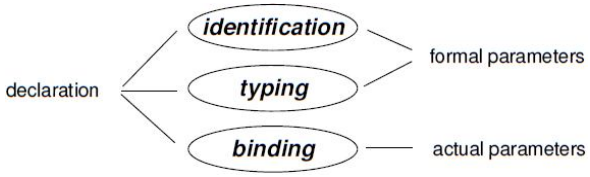
דוגמא: ישנן פונקציות שלא עובדות כאשר מועבר אותו המשתנה- swap2(*pa, *pb). אם מדובר באותו המשתנה, הפונקציה מאפסת אותו במקום לעשות swap.

```
procedure swap2(var a,b: Integer)
begin
    a := a + b;
    b := a - b;
    a := a - b;
end;
```

Fortran: שם אין aliasing והתוכניות בדרך-כלל רצות הרבה יותר מהר מ-C.

C: מניחים aliasing (worst case scenario) ולכן לקומפיילר לוקח יותר זמן.

The Correspondence Principle: לכל סוג של הצהרה יש מנגנון פרמטרי מתאים.



סדר השערוך:

מתי מחשבים את ערכו של הארגומנט שהועבר לפונקציה? מוגדר בשפה.

Eager evaluation: כל הארגומנטים מחושבים מייד בהעברה לפונקציה.

Normal-order evaluation: הארגומנטים מחושבים רק כאשר יש בהם שימוש. ולפני כל שימוש.

יתרון: אם לא נשתמש בהם לא נצטרך לחשב אותם.

חיסרון: אם נשתמש בהם כמה פעמים, חישוב הארגומנט יתבצע כמה פעמים.

Lazy evaluation: הארגומנטים מחושבים רק בשימוש הראשון, וערכם נשמר עבור שימושים נוספים.

יתרון: ניתן להפריד בין חישוב לבקרה על החישוב.

חיסרון: כל פעם שניגשים למשתנה צריך לבדוק אם הוא כבר חושב וזה- overhead.

הערה: בשפות אימפרטיביות מוגדר סדר החישוב. בשפות פונקציונאליות הוא לא מוגדר.

C: normal order- במאקרו הבא-

```
#define MAX(a, b) (a < b ? a : b)
```

ההפעלה הזו: `MAX(getChar(f1), getChar(f2))` קודם כל תקרא תו מ-f1 אח"כ תקרא תו מ-f2 ואז תשווה ביניהם וכדי להחזיר את a או b תקרא תו נוסף מהקובץ שבו התו הקטן במקום להחזיר את התו הקטן.

Extremely Lazy evaluation: ניתן לחשב רק את חלקו הרצוי של הביטוי ולא את הביטוי כולו.

פונקציות strict ו-Non strict:

Strict: הפונקציה היא strict בארגומנט ה-n שלה אם לא ניתן לשערך את הפונקציה ללא שערוך הארגומנט ה-n.

Non-strict: הפונקציה היא non-strict בארגומנט ה-n שלה אם יש מקרים בהם ניתן לשערך את ערך הפונקציה ללא שערוך ערכו של הארגומנט ה-n.

The Church-Rosser Property:

שפת תכנות מקיימת את תכונת Church-Rosser אם כל ביטוי בשפה מקיים את התכונה.

קיום התכונה: אם ניתן בכלל לשערך את הביטוי, ניתן לעשות זאת באופן עקבי עם Normal order evaluation. במידה וישנן כמה דרכים לשערוך הביטוי בסדר שונה, כל סדר שערוך ייתן את אותה התוצאה.

הערות:

- רוב שפות התכנות לא מקיימות את התכונה.
- כל שפת תכנות שמאפשרת side effects לא מקיימת את התכונה.

הידור והרצה של תוכניות Java

הקדמה:

הבדלים בין C ל-Java:

1. קובץ המטרה בשפת C גדול הרבה יותר מקובץ המטרה בשפת Java.

```
% ls -lsh Hello.*
4.0K -rw-r--r-- 1 yogi yogi 418 2011-01-19 14:09 Hello.class
4.0K -rw-r--r-- 1 yogi yogi 119 2011-01-19 14:09 Hello.java

% ls -lsh hello*
8.0K -rwxr-xr-x 1 yogi yogi 7.0K 2011-01-20 14:29 hello*
4.0K -rw-r--r-- 1 yogi yogi   59 2011-01-20 14:22 hello.c
```

2. קובץ המטרה ב-Java מכיל הרבה יותר מידע טקסטואלי מאשר קובץ המטרה ב-C.
3. קובץ המטרה ב-C ניתן להרצה ישירה ע"י הפקודה: hello/. לעומת-זאת יש להשתמש בפקודה מיוחדת כדי להריץ את קובץ המטרה ב-Java: java hello.

JVM:

תוכנית ב-Java אינה מהודרת לשפת מכונה עליה תרוץ התוכנית. היא מהודרת לשפת מכונה של מכונה וירטואלית: JVM. הקובץ filename.class מכיל פקודות לביצוע עבור המכונה הזו.

קיימות תוכניות מחשב שיודעות לקרוא את הפקודות של ה-JVM, לפענח ולבצע אותן.

אפשר לממש את ה-JVM בשפות: C (מה שקורה בדרך-כלל), Assembly ואפילו ב-Java.

הידור של תכנית Java הינו תרגום תוכנית בשפת Java לתוכנית שקולה בשפה שאותה מכירה מכונת ה-JVM. השפה הזו נקרא: Java Byte Code. מימוש ה-JVM הוא תלוי מכונה ותלוי מערכת הפעלה.

יתרונות:

- מהרגע שהותקן במחשב מימוש כלשהו של JVM, ניתן להריץ עליו כל תכנית לאחר שקומפלה ל-Java Byte Code. זה מאפשר הרצת תוכניות Java על כל מחשב ללא צורך בהתקנה מיוחדת.
- Safety: ה-JVM תוחם את מרחב הפעולה של תכניות ויכול למנוע מהן לבצע פעולות שאינן מותרות להן, כמו גישה לכתובות זיכרון לא חוקיות. JVM מסוגל להבטיח Strong typing.

חיסרון עיקרי: ביצועים איטיים יותר.

פיתרון: ישנם JVM-ים מתוחכמים ובהם ישנ: **JIT=Just In Time compiler** שזו טכנולוגיית האצה. ה-JIT מזהה קטעים בתוכנית אשר מתבצעים שוב ושוב ומתרגם אותם תוך כדי ריצה לשפת מכונה פיזית וזה מביא לשיפור גדול במהירות הביצוע.

הערה: JVM לא משמש רק ל-Java, ישנן שפות אחרות המהודרות ל-Java Byte Code.

הידור תוכנית ב-Java:

כל תוכנית ב-Java מורכבת מקבצים. בכל קובץ:

- הגדרה של מחלקה ראשית ששמה זהה לשם הקובץ.
- מחלקות משנה (בדרך-כלל מחלקות המקוננות במחלקה הראשית, השימוש באלה שאינן מקוננות- non public הוא נדיר).

הערה: גם בתוך המחלקות המשניות ניתן להגדיר מחלקות מקוננות אך השימוש בכך נדיר.

בדרך-כלל מספיק להדר קובץ אחד, את הקובץ המכיל את המחלקה שבה מתחיל ביצוע התוכנית. וזאת מכיוון שהידור הקובץ הזה יגרום לקומפיילר לחפש את כל המחלקות האחרות שבה עושה שימוש המחלקה הזו ולהדר אותן במידת הצורך.

החיפוש וההידור ממשיכים באופן רקורסיבי עבור כל המחלקות.

ארגון ה-packages בשפה:

כל חבילה מהווה מרחב שמות אחד והחבילות כולן מאורגנות במבנה של עץ.

דוגמא: תוכניות שנכתבו בפקולטה למדעי המחשב בטכניון, צריכות להימצא בחבילה ששמה: ll.ac.technion.cs (cs שבתוך technion וכן הלאה..)

מערכת הקבצים: ישנו מיפוי היררכי של מבנה עץ החבילות לתת-עץ של מערכת הקבצים.

דוגמא: כל המחלקות השייכות לחבילה הקודמת יימצאו בתיקייה: /ac/technion/cs/. כאשר ℘ מציין את המקום שבו מתחיל תת העץ של מערכת הקבצים.

בתיקייה זו, תימצא המחלקה: <classname>.java וגם תוצאת ההידור: <classname>.class.

כדי לאתר קובץ, המהדר חייב לקבל את מיקום הנקודה: ℘ שם תיקייה במערכת הקבצים שממנה מתחיל המיפוי. בדרך-כלל נוהגים למפות למספר תתי-עצים. בכל אחד מהם קבצים מסוג אחר.

CLASSPATH: רשימת המקומות שמהם יש להתחיל בחיפוש.

כדי להדר מחלקה, יש תחילה לקבוע את ה-CLASSPATH. נעשה באחת משתי דרכים:

1. הגדרת משתנה סביבה: export CLASSPATH=/home/…/path1:/home/…/path2
2. העברת ה-CLASSPATH בזמן הידור (כלומר: בזמן הפעלת הפקודה javac).

```
% javac -classpath /home/yogi/java:/home/yogi/import/lib StringParserDemo.java
```

זו שיטה פחות יעילה, כי בדרך-כלל נרצה לקבוע את ה-CLASSPATH להרצות רבות של המהדר.

אם המחלקה שהודרה משתמשת במחלקות נוספות, המהדר יוודא שגם הן קיימות ומהודרות.

החיפוש יביא לאחת מבין שלוש תוצאות:

1. מציאת קובץ מקור: x.java אז המהדר בודק האם גם הקובץ x.class נמצא שם.
   אם הקובץ x.class לא נמצא או שהתאריך שלו מוקדם לתאריך של הקובץ x.java המהדר יהדר את הקובץ x.java וייצור את קובץ ה-x.class באותה התיקייה.
2. מציאת קובץ מטרה ללא קובץ מקור: אם הקובץ x.java לא נמצא באף אחד מהתיקיות ב-CLASSPATH ובאחת מהן נמצא קובץ x.class אז המהדר ממשיך הלאה.
3. העדר קובץ מטרה וקובץ מקור: אם הקבצים x.java ו-x.class לא מצויים באף-אחד מהמקומות, המהדר מודיע על שגיאה ועוצר.

לאחר שהמהדר יצר את הקובץ class. הוא מחלץ ממנו את רשימת כל המתודות המוגדרות במחלקה ורשימת כל המשתנים המוגדרים במחלקה, ובודק שהשימוש של המחלקה במחלקות האחרות הוא נכון מבחינת טיפוסים- static typing.

הערה: ניתן להדר תוכניות Java שיש בהן תלות מעגלית.

דוגמא:

בהידור המחלקות הבאות:

```
% echo "public class A1 { public A1() { new A2(); }}" > A1.java; \
  echo "public class A2 { public A2() { new A3(); }}" > A2.java; \
  echo "public class A3 { public A3() {}}" > A3.java
```

כדי להדר את A1, המהדר צריך למצוא ולהדר את A2.

```
[checking A1]
[loading ./A2.java]
[parsing started ./A2.java]
[parsing completed 0ms]
[wrote A1.class]
```

אבל כדי להדר את A2 יש למצוא ולהדר גם את A3 באותו אופן:

```
[checking A2]
[loading ./A3.java]
[parsing started ./A3.java]
[parsing completed 1ms]
[wrote ./A2.class]
```

מחלקה A3 לא תלויה באף מחלקה אחרת

```
[checking A3]
[wrote ./A3.class]
```

אם ננסה להדר שוב את A1.java, המהדר יאתר את A2.class ומכיוון שהוא מעודכן יותר מ-A2.java, המהדר לא יהדר אותו ולא יטרח לבדוק בכלל את A3.

מבנה קובץ Class:

בקובץ Class יש מידע מספיק המאפשר לעשות type checking. בקובץ ישנם 3 חלקים עיקריים:

1. **Header:** מספר קטן של בתים המזהה את הקובץ.
2. **Pool tables:** מכילות את כל המחרוזות בהן משתמשת המחלקה ואת רשימת המתודות והמשתנים המוגדרים במחלקה.
3. **Byte code:** הפקודות בשפת ה-Java Byte Code שיש בגוף המתודות.

ניתן בקלות לשחזר מתוך קובץ class את החתימה של המחלקה המקורית:

המחלקה:

```
public class Hello {
  public static void main(final String[] args) {
    System.out.println("Hello, World!\n");
  }
}
```

התיאור שלה בקובץ Class:

```
% javap Hello
Compiled from "Hello.java"
public class Hello extends java.lang.Object{
    public Hello();
    public static void main(java.lang.String[]);
}
```

הרצה של תוכניות Java- הפעלת ה-JVM:

Engine: מבצע את מחזור הפעולות הרגיל שמבצע ה-CPU במכונה פיזית:

1. קידום ה-PC.
2. אחזור הפקודה הבאה.
3. פענוח הפקודה.
4. ביצוע הפקודה.

ישנם משתנים בתוך ה-JVM שמדמים את הרגיסטרים של המכונה, מערכים המדמים את מרחב הזיכרון ועוד.

Class Loader: בכל פעם שהביצוע של ה-engine מגיע לפקודה שבה יש גישה למחלקה אחרת, טוען המחלקות מוודא שהמחלקה הזו טעונה. יש לו טבלה גדולה ובה רשימת כל המחלקות שנטענו כבר.

Dynamic loading: אם המחלקה בטבלה, ביצוע המנוע ימשיך כרגיל. אם אינה טעונה, טוען המחלקות ישתמש ב-CLASSPATH לאיתור קובץ ה-class של המחלקה.

יתרון השיטה: אין צורך לטעון חלקי תוכנית שייתכן שלא יבוצעו.

חיסרון השיטה: בזמן ריצה ייתכן שיהיו חלקים חסרים או בלתי תקינים.

לאחר שאותר הקובץ, קורא אותו טוען המחלקות ובודק אותו דקדוקית וטכנית. לאחריה, מתבצעת בדיקת טיפוסים. זאת בנוסף לבדיקת הטיפוסים שכבר נעשתה בזמן הידור.

בדיקה זו מיועדת להגנה מפני שינוי קובץ ה-Class לאחר ההידור.

שגיאות המתגלות בשלב זה, ייגרמו להפסקת ביצוע התוכנית.

המאמת:

מכיוון שהשפה: Java Byte Code היא strongly typed, בתוך טוען המחלקות ישנו "המאמת": **Byte Code Verifier**. תפקידו לבדוק שה-Byte Code לא מבצע שגיאת טיפוס. הוא פועל עם טעינת המחלקה לזיכרון.

<u>פעולות המאמת:</u>

1. <u>חלוקת הקוד לביצוע לבלוקים בסיסיים:</u> בלוק=סדרת פקודות מכונה רצופות המתבצעות תמיד ביחד. הפקודה האחרונה בבלוק תהיה sequencer (פקודה המשנה את הסדר הרגיל של הביצוע).
2. <u>בנית גרף המעברים האפשריים בין הבלוקים הבסיסיים:</u> שני בלוקים יחוברו בקשת מכוונת אם ייתכן שיבוצעו ברציפות
3. <u>איתור רגיסטרים שבהם נעשה שימוש בכל בלוק בסיסי</u>.
4. <u>איתור פקודות המחסנית שבכל בלוק בסיסי</u>.

לאחר-מכן, המאמת ינסה לפתור מערכת משוואות שבה הנעלמים הם:

- הטיפוסים של הערכים השמורים ברגיסטרים בכל פקודה.
- עומק המחסנית בכל פקודה.
- הטיפוס של כל אחד מהערכים במחסנית בכל פקודה.

הנתונים אשר כן ידועים למאמת הם טיפוסי השדות והמתודות.

פיתרון מערכת המשוואות נעשה ע"י ניחוש פתרון ותיקונו עד להתכנסות לפיתרון הנכון.

שגיאות טיפוס המתגלות בשלב זה יגרמו להפסקת ביצוע התוכנית.

<u>אתחול מחלקה:</u>

לאחר הבדיקה של המאמת, טוען המחלקות טוען את המחלקה.

האתחול שלה כולל יצירת כל המשתנים הסטטיים של המחלקה ואתחול שלהם. במקרים רבים, אתחול משתנים סטטיים כרוך בביצוע קוד וייתכן שביצועו יגרום לטעינתן של מחלקות נוספות.

Google's Go Programming Language

ב-10/11/2009 גוגל הכריזה על שפת תכנות חדשה.

Give us the good old C back, but better…

תכונות בסיסיות:

Compiled, concurrent, imperative, structured, latency free garbage collection

בכל package יש פונקציות שניתן לעשות להן import.

אין צורך בשמות טיפוסים, אבל השפה אינה דינאמית כמו ML.

```
package main
import "os"
import "flag"
var nFlag = flag.Bool("n", false, `no \n`)
func main() {
    flag.Parse()
    s := "";
    for i := 0; i < flag.NArg(); i++ {
        if i > 0 { s += " " }
        s += flag.Arg(i)
    }
    if !*nFlag { s += "\n" }
    os.Stdout.WriteString(s);
}
```

Design Principles:

אורתוגונאליות: מעט קטעים בלתי תלויים עובדים יותר טוב מהרבה קטעים חופפים (מקביליות).

דקדוק פשוט: מעט מילים שמורות, ניתן לפרסר בלי symbol table.

פחות typing: השפה תבין לבד מה שהיא יכולה במקום להגדיר מראש את הטיפוסים המסובכים.

אין היררכית טיפוסים, מערכת הטיפוסים ברורה יותר, פחות בטוח לכתוב קוד שבו יש היררכיית טיפוסים.

דברים נוספים:

נקודה פסיק: אין הכרח לשים. דרוש רק עבור שתי פקודות באותה השורה.

סוגריים מסולסלים: חובה בכל פקודה, גם בכאלה של שורה אחת.

הצהרה על טיפוסים: אין צורך להצהיר, השפה תבין לבד מהו הטיפוס.

ערכים פרימיטיבים: Boolean, Integral (int), Unsigned, Float, Complex.

מחרוזות: לא ניתן לשנות תו אחד במחרוזת, אבל ניתן לקרוא תווים בודדים.

מערכים: כמו ב-C, מתחילים מ-0 עד ל-size. גודל המערך צריך להיות ידוע בזמן הקומפילציה (static), אסור pointer arithmetic.

המרות: אין המרות implicit, כל ההמרות צריכות להיות explicit.

מערכת הטיפוסים של Go:

קיום: ב-Go קיימת מערכת טיפוסים.

שקילות ותתי-טיפוסים: שני טיפוסי הם שקולים אם שניהם פרימיטיבים המוגדרים בשפה, או ששניהם טיפוסים חסרי שם המכילים אותם שדות בדיוק. ניתן לדמות תתי-טיפוסים ב-Go בעזרת שימוש ב-interface, כאשר טיפוס א' הוא תת-טיפוס של טיפוס ב' אם הוא מממש תת-קבוצה של המתודות של טיפוס ב'. בפרט, ה-interface הריק ({}) הוא תת-טיפוס של כל טיפוס אחר.

חוזק: מערכת הטיפוסים היא Strongly Typed. לא ניתן לשבוק את הזיקה בין ערך לבין הטיפוס שלו. למשל, לא ניתן לשלוח משתנה מטיפוס int לפונקציה שמצפה לקבל פרמטר מטיפוס float.

זמן בדיקה: מערכת הטיפוסים היא Statically typed. אכיפת מערכת הטיפוסים נעשית בזמן הקומפילציה. עם זאת, למשתנה מסוג interface יש גם טיפוס דינאמי, המציין את הטיפוס של הערך שמוחזק בזמן ריצת התוכנית.

אחריות: בשפת Go קיימת מערכת הסקת טיפוסים אוטומאטית (implicit typing) חלשה מאוד, המאפשרת להסיק את הטיפוס של ליטרל לפי אופן השימוש בו ואת הטיפוס של משתנה לפי הערך

אליו הוא מאותחל. לא ניתן להסיק לבד את הטיפוס של פונקציה, למשל, בניגוד לשפת ML. ניתן לציין בצורה מפורשת גם את הטיפוס של כל משתנה, ולכן קיימת גם מערכת explicit typing.

גמישות: בשפת Go אין פולימורפיזם, מלבד העמסה של אופרטורים מובנים, כגון אופרטור +.

Sequencers:

return: סיום פונקציה.

go: הפקודה מתחילה ביצוע של פונקציה או מתודה במקביל לריצת התוכנית.

break: סיום ביצוע של לולאה פנימית יותר.

continue: מעבר לאיטרציה הבאה בלולאה פנימית ביותר.

goto: ביצוע קפיצה לפקודה אחרת.

fallthrough: קפיצה לפקודה הראשונה של ה-case הבא בפקודת switch.

defer: קריאה לפונקציה, כך שהיא תופעל רק כאשר הפונקציה הנוכחית תסיים את ריצתה.

panic, recover: מנגנוני טיפול בשגיאות.

List Processing with C++ Templates

פעולות בסיסיות:

Cons: יצירת רשימה מהפרמטרים head ו-rest. הפרמטר השני בדרך-כלל יהיה טיפוס המייצג רשימה נוספת, שגם נוצר באמצעות Cons.

```
template<typename H, typename T>
  struct Cons {
    typedef H Head;
    typedef T Tail;
  };
```

מבנה ה-template class: הגוף יכול להיות ריק כי כל שברצוננו הוא ליצור טיפוס חדש. נעשה שימוש בשני typedef-ים אשר יאפשרו גישה מבחוץ ל-Head ול-Tail.

יש שימוש ב-struct במקום ב-class כי עבור class היה צורך להגדיר את ה-typedef-ים כ-public כדי לאפשר גישה מבחוץ.

```
class Null {};
```

סיום רשימה: סימן מיוחד ה-class Null.

הערה: למרות שה-class Null הוא ריק, instances שלו כן תופסים מקום בזיכרון.

דוגמאות לרשימות:

רשימה עם איבר אחד:

```
typedef Cons<double, Null> X;
```

רשימה עם שני איברים:

```
typedef Cons<char, Cons<int *, Null> > Y;
```

List processing עושים בעזרת רקורסיה. לא נצטרך לדאוג להקצאת זיכרון מכיוון שהקומפיילר של C++ מנהל בעצמו את הזיכרון לכל הטיפוסים שהוא יוצר מטיפוסים קיימים.

Value semantics: הטיפוסים שמיוצרים הם immutable ולכן הקומפיילר מאפשר שיתוף זיכרון- כאשר מועבר פרמטר ל-template הקומפיילר לא צריך להעתיק אותו.

The Length template:

חישוב האורך של רשימה. הפונקציה תיצור טיפוס חדש עבור ההפעלה שלה, והאורך יאוחסן כ-integer בטיפוס הזה ע"י הגדרת enum בתוך הטמפלייט.

```
template <typename S>
  struct Length {
    enum {
        result = 1 + Length<typename S::Tail>::result
        // the above does the recursive call.
    };
};
```

C++ היא dynamically typed, הקומפיילר לא עושה שום בדיקת typing לפני שהוא יוצר את הטמפלייט. הטמפלייט נוצר בכל פעם שהוא מקבל פרמטר. הוא מצפה לקבל S שהוא מטיפוס רשימה עם Cons אשר יש לו את השדה Tail. אם יקבל משהו אחר, רק במהלך הריצה תתקבל הודעת שגיאה.

יש צורך לשים typename לפני S::Tail כי הקומפיילר לא יסכים לקמפל את הטמפלייט אם הוא לא ידע שזהו סוג של טיפוס.

תנאי העצירה: נעשה עם template specialization.הוא אומר לקומפיילר שאם הטמפלייט נוצר עם הפרמטר Null אז הגדרתו תהיה כדלקמן.

```
template<>
    struct Length<Null> {
        enum { result = 0 };
    };
```