

# סיכום שפות תכנות לבחן

## סיכום, אסף, איגד ומה לא – נחום בונדק.

הסיכום נכתב על בסיס מצוגותיו וכתביו של פרופ' גיל – נעשה שימוש בגרסאות ה- Beamer.

ובסיוע סיכומיים של תומר דרגוצקי, רוז פקטרמן, רבקה אברהם וAIRIS קלקה (פרק פרקים 2-1).

הקבצים בעברתי עלייהם בניסיון לאגד את החומר – ונמצאים פה:

06 – StructuralandNominalTyping.pdf -

First – steps\_summary.pdf -

pdf. הריצאות סיכום תכנות שפות – סיכום של רבקה אברהם מהפייסבוק.

pdf. שפות סיכום – סיכום של תומר דרגוצקי מהפייסבוק.

First Steps.pdf – "Types of identifiers" – First Steps.pdf -

Type Equivalence Spring 2015.pdf – של דר. שרה פורת.

– של דר. שרה פורת. Parameter Passing.pdf -

– עברו נושאים שלא היו ברורים לי. Wikipedia -

האתר – עברתי עליו כמעט במלואו והעכrichtי את כל מה שיכלתיلقאו. Safot.net -

הקובץ שחילקית ממנו נמצא כאן:

0102\_Introduction.pdf -

קבצים שמיידם לא הועבר לך:

Examples.pdf -

Tree Parse -

פרק 6 מהציגת של ה- Beamer – הפרק של Exceptions לא מסוכם – לא בחומר של הקורס בסמסטר שלי.

הערות:

-

כשהדף מוחולק ל-2 או אם התמונות מוחולקות ל-2 אז הקריאה מתבצעת תחיליה מעמודה שמאלית למעלה – עד למיטה ואח"כ

מעמודה ימנית למעלה – עד למיטה.

-

מה שבאודם זה דברים שצריך לוודא או דברים שאני עוד צריך להשלים.

-

יש המון חומר שלקחתי מהאתר של Safot.net ושמתי כאן. חלקם אולי לא מדויקים לגמרי או לא חci נכונים – ומה שעול

-

להיות לא מדויק – מזוכר. אני אישית מסתמכ עליהם וזאת למורות שלא את כולם אישר פרופ' גיל. צריך להשתמש בשכל

...הישר...

-

שאלות רבות שרשומות פה הן שאלות מבחנים, כך שאם אתם מתכוונים לפתרן מבחנים, קחו בחשבון שתראו חלק

מהשאלות שנמצאות פה ב מבחנים עצם.

# 1. מבוא והגדרה של שפת תכנות (צעדים ראשוניים)

פרדיגמה – דרך מחשבה וסגנון תכנון, אשר שפת התכנות מכתיבה למתכנת.

הפרדיגמה ה~~אווית~~הפרוצדורי/~~אימפרטיבית~~(Imperative):

- בפרדיגמה זו נמצא משתנים, ביטויים, פקודות, כולל פקודות תנאי ופקודות לולאה, פונקציות ופרוצדורות.
- תכנית הכתובה בשפה אימפרטיבית יש בכל רגע מצב (State) המוגדר ע"י תוכן הזיכרון (אוסף המשתנים ה"חיים" ונקודת הריצה הנוכחית).

תכנית בפרדיגמה זו נעשו באמצעות ציוקי: כתיבת פקודות אשר מצוות על התכנית לשנות את מצבה.

פקודות בשפה אימפרטיבית יכולות להיות מאוגדות בפרוצדורות.

אחד הסיבות לכך שהפרדיגמה האימפרטיבית היא הנפוצה ביותר היא ששפות המכונה הן אימפרטיביות. תרגום תכנית בשפה אימפרטיבית לשפת מכונה יכול להיות על כן עיל במוחך.

לדוגמה השפות: C ,Bash ,Pascal

הפרדיגמה הפונקציונלית:

- בפרדיגמה זו אין משתנים כלל ואין פקודות.
- את מקום פקודות התנאי מחליף "ביטוי מותנה". את הלולאות מחליף רקורסיה. באופן מפתיע (כמו ב- ML) ניתן לתכנית ללא שימוש במשתנים.

כתוצאה מהעדר המשתנים והפקודות, לתכנית אין "מצב בזמן ריצה", פרט לנוקודות החישוב הנוכחיות.

שם הפרדיגמה נובע מכך שפונקציות מהוות את הדריך העיקרי לתיאור החישוב. תכנית בפרדיגמה זו מרובה לחשב ביטויים באמצעות קריאה לפונקציות, להעביר פונקציות כפרמטר לפונקציות אחרות, וכתייבת פונקציות אשר מייצרות ומחזירות פונקציות אחרות.

נשים לב שבהיעדר פקודות כל מה שפונקציה יכולה לעשות הוא להחזיר את ערכו של ביטוי, אשר יכול להיות מוחושב באמצעות קריאה לפונקציות אחרות, או באמצעות האופרטורים היסודיים.

שפות פונקציונליות הן לרוב אלגנטיות במובן זה שיש מספר קטן של מושגים אוטם יש להבין כדי לתכנת בשפה, וגם במובן זה שהן מאפשרות ביטוי תמציתי של דרך החישוב. באופן טיפוסי, תכנית לפתורן בעיה מסוימת בשפה פונקציונלית תהיה קצרה בכמה מונים מ\_ticks דומה בשפה ציונית.

לדוגמה השפות: LISP, Haskell ,Scheme ,ML

הפרדיגמה מונחתית העצמים:

- מכילה אובייקטים אשר בנויים מידע ומתחודות.
- בפרדיגמה מונחתית עצמים, כל החישוב כולם הוא העברת הודות לעצמים, אשר מגיעים אליהם בחישוב קצר בהתאם לטיבו של העצם המתקבל.

השפות + C++ ו- Java הן דוגמאות שלה, אף שאינן תכונות מונחות עצמים טהורות, בנגדו ל- Eiffel ו- Smalltalk. הסיבה לכך:

- הדוגמא הכי טובה היא הטיפוסים הפרימיטיביים. ב- Smalltalk ו- Eiffel גם הטיפוסים הפשוטים כמו int הם אובייקטים. ב- Java ו- C++ יש טיפוסים פרימיטיביים שהם לא אובייקטים.
- ביצוע לולאת Smalltalk לדוגמה ב- For זה שליחת הודה לאובייקט 1 מסוג do: to: שמקבלת 2 פרמטרים ה- n והבלוק וכך הלאה. ב- C++/Java הוללת For הרגילה נועשית ללא שליחת הודה לאובייקטים, אלא זה פשוט מתקפל לפועלות ישרות.

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

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

(כמובן לנוקודה השנייה) הביטוי `a < b` שנראה כביוול כביטוי אינו ביטוי כלל וכלל, אלא שליחת הודה < עם הfrmater `b` לעצם `a`. בתגובה להודה זו העצם `a` יחזיר את העצם `true` או העצם `false`.

- לתוצאה המוחזרת תשלוח הודה

`ifTrue: ... ifFalse: ...`  
אשר לה שני ארגומנטים, אשר הם בלוקים המוכנים לחישוב.

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

כעת העצם true יחשב את הארגומנט הראשון ויתעלם מהשני, ואילו העצם false יתעלם מההשניה ויתחשב את השני.

#### הפרדיגמה הלוגית:

- מכילה עובדות וכללים. מדגישה את מה שצורך ולא כיצד מוחשב.
- החשוב מרוש – המונע מוכיה את המשפטים פטור בעזרת הנחות (עובדות) שניתנו לו.
- הפרדיגמה הלוגית מתאפיינת בכך שאין בה לא פקודות הצבה, לא פקודות תנאי ולא פקודות לולאה. בפרדיגמה זו אין גם פונקציות ולא פרוצדורות, ואין (כמעט) מילים שמורות. בשפות בפרדיגמה הלוגית יש אמן "משתנים", אך אלו שונים לחלוין בטיבם מהמשתנים המוכרים לנו בשפות אימפרטיביות.

לדוגמה שפת Prolog

- במקום כל אלו, תכנית בפראולוג מורכבת משלושה חלקים:

a. עובדות יסוד.

b. כללי היסק.

c. מטרת החישוב.

כל אחד משלושת אלו מנוסח כטענה לוגית. יתרה מכך, עובדות הן למעשה כליה היסק, אשר בסיס היסק של זה ריק. ניתן לנצלו ל开玩笑 עובדות היסוד וכלי היסק אקסומות, הנכתבות כפרדייקטים לוגיים. המתכונת הכותבת בפראולוג, מגדר אווסף של אקסומות, ולאחר כך, החישוב בפועל מתבצע ע"י הצבת "מטרה", שהוא פרדייקט בלוגייקה.

בפראולוג יש מונע פנימי המנסה להוכיח את פרדייקט המטרה מתוך האקסומות. ההוכחה יכולה להיות של הפרדייקט הכללי, או של מקרים פרטיים של המשפט המתקבלים באמצעות קביעה ערך של משתנים עבורי. טרם תחילת הביצוע בפראולוג, מונע החישוב מספר רב של פרדייקטים.

#### שפה אוניברסלית או שפה טיורינג שלמה (Turing Complete) – זהה שפה השקולה החישובית למוכנות טיורינג.

- בפרט, שפת תכנות המכילה תנאים לוגיים ורקורסיביים (או להילופין, לולאות), שקופה מבחינה החישובית למוכנות טיורינג ולכל תקרה שפה אוניברסלית לשפה טיורינג שלמה.
- השיקולות מתייחסת לשאלה של פתרונות – ולא לשאלה של יעילות. ככלומר רק אם קיימת תכנית הפותרת בעיה מסוימת, בלי קשר לכמה היא עיילה.
- "מוכנות טיורינג" – מודול חישובי.
- המציג: Alan Turing.
- "תחשיב הלמבדא" – שיטה מתמטית להגדרת חישוב.
- המציג: Alonzo Church.
- "מוכנות טיורינג" ו"תחשיב הלמבדא" הם פורמליזומים זרים ושניהם אומרים בגודל "כל המודלים החישוביים זרים".
- שקופה החישובית לשפת C שקופה החישובית לשפת Prolog.

#### בסיום – Introduction 2010 – סיכום הרצאות 1 ו-2 של איריס קלקה – יש עוד חומר שלא הנטתי כאן:

- שיקולים בתכנון שפות תכנות.
- העקרונות שנitinן למצוא בשפת C בהקשר של השיקולים הנ"ל.
- הכללים הנפוצים לתיאור משמעות של תכנית.

פעמים רבות הסייעו בפתרונות התוכנות מוגדר בעזרת קבוצות מוגדרות רקורסיבית. בודומה לנלמד בלוגיקה, קבוצה זו מורכבת מאטומיים וכלי היסק אותו נוכל להפעיל על האיברים שכבר נמצאים בקבוצה. כמו כן כל איבר אחר לא שייך לקבוצה.

למשל, הגדרת אווסף המחרוזות מעל א"ב מסוים:

Given an alphabet  $\Sigma$ , the set  $\Sigma^*$  is defined by

- $\epsilon$ , the empty string is in  $\Sigma^*$ ,

$$\epsilon \in \Sigma^*$$

- if  $\ell$  is a letter,  $\ell \in \Sigma$ , and  $s \in \Sigma^*$  is a string, then

$$\ell s \in \Sigma^*,$$

where  $\ell s$  is the string obtained by prefixing  $\ell$  to  $s$ .

- there are no other members of  $\Sigma^*$

#### ביטויים רגולריים:

- משמשים להגדרות דקדוקיות פשוטות. כך למשל, הגדרת פקודת תנאי בפסקל בפורמליזם זה נראה כך:  $(_a-zA-Z)_a-zA-Z)^*$
- זהה לבוצעה מוגדרת רקורסיבית:

Given an alphabet  $\Sigma$ , the set  $\text{RE}(\Sigma)$  is defined by:

- $\Sigma^* \subseteq \text{RE}(\Sigma)$
- if  $e_1, e_2 \in \text{RE}(\Sigma)$  then
  - alternatives  $(e_1 | e_2) \in \text{RE}(\Sigma)$
  - concatenation  $(e_1 e_2) \in \text{RE}(\Sigma)$
  - Kleene closure  $(e_1^*) \in \text{RE}(\Sigma)$

- לרוב בעת אימוץ הביטויים הרגולריים מוסיפים סופר סינטטי = דרך להביע את אותו ביטוי רגולרי בצורה קלה/קצירה יותר. למשל [a-z] [ ] משמעתו אחת מהאותיות a-z.
- בשפת התכנות C הביטויים המתמטיים, הפקודות והטיפוסים כוללים מוגדרים רקורסיבית.
- כוח הביטוי של ביטויים רגולריים הוא מוגבל, למשל אין דרך שמספר הסוגרים בביטוי יהיה מואزن, ועל כן השימוש בביטויים רגולריים מוגבל להגדרות פשוטות של אבני הבניין של השפה: משתנים, הערכות, מספרים וכו'.

להגדרות מורכבות יותר יש להשתמש במנגנון הידוע בשם דקדוק חסר הקשר בشرط הסימון הידועה בשם BNF.

#### BNF – שיטת סימון של מנגנון כתיבת דקדוק חסר הקשר:

הגדרת דקדוק בשיטת סימון זו מרכיבת ארבעה חלקים:<sup>1</sup>

1. קבוצה של סימנים סופיים, **Terminals** (ה-TERMINALS לעתים קרויים גם Tokens או אסימונים). אלו הם "תווים אוטומטיים". הדקדוק מPTR שפה מעל האלפבית שייצרים הסימנים הסופיים.
2. קבוצה של סימנים לא סופיים, **Non Terminal Symbols**, המשמשים ככל' עוז להגדרת הדקדוק. סימני עוז אלו דומים מעט לשימוש בשמות לביטויים רגולריים חלקיים, אלא שהגדורתם של סימני העוז יכולת להיות רקורסיבית ונitinן להגדירן וותר מאשר פעם אחת. סימנים אלו לא חלק מהשפה ורק משמשים עוז להגדרתה.
3. קביעה של **Start Symbol** – קביעה של אחד מהסימנים הלא סופיים כסימן התחלה – Start Symbol.

<sup>1</sup> נלקח מהסבירו של איריס קלקה – סיקום הרצאות 1-2. כמעט כל החומר של התחלה שלו נלקח ממש – ואלו הוספות דברים שנראו לי בחשובים והסבירים נוספים.

4. אוסף של כללי גזירה, כאשר לכל הגזירה יש שני חלקים:

- ראש הכלל כתוב בצד שמאל של הכלל, והוא תמיד סימן לא סופי ( Non Terminal Symbol ).
- ואילו גופו הכלל כתוב בצדיו הימני, הוא סדרה (היכול להיות ריקה) של סימנים סופיים ולא סופיים ( Terminals & Non-Terminals symbols ).

כללי הגזירה נכתבים כך שישנו חז' המוביל מראש הכלל אל גופו.  
בפועל נהגים להשمي את המרכיבים 1 עד 3 של הגדרת הדקדוק ולהסתפק בכללי הגזירה בלבד. קל להבחין בין סימנים סופיים ולא סופיים בכללי הגזירה, משום שסימן סופי לא יופיע לעולם בראש כלל. גם סימן ההתחלה ברור בד"כ מההקשר.

הפורמליזום של דקדוק BNF חזק יותר מהפורמליזום של ביטויים רגולריים שכן הוא מתייר הגדרות רקורסיביות.  
לביטוי חקי נוכל למצוא עץ יצירה באמצעות הכללים שהגדכנו בהגדרת הסינטקס.  
نبטא באמצעות דקדוק חסר הקשר: מזהים, מילים שמורות, מיולונים ועוד ...

נראה הגדרה של דקדוק BNF באמצעות BNF ( מתוך [ויקיפדיה](#) ):

BNF's syntax itself may be represented with a BNF like the following:

```
<syntax>      ::= <rule> | <rule> <syntax>
<rule>        ::= <opt-whitespace> "<" <rule-name> ">" <opt-whitespace> "::=" <opt-whitespace> <expression> <line-end>
<opt-whitespace> ::= " " <opt-whitespace> | ""
<expression>   ::= <list> | <list> "|" <expression>
<line-end>    ::= <opt-whitespace> <EOL> | <line-end> <line-end>
<list>         ::= <term> | <term> <opt-whitespace> <list>
<term>         ::= <literal> | "<" <rule-name> ">"
<literal>      ::= ' ' <text> ' ' | '\" <text> '\"
```

## 2:EBNF

מוסיף ל-BNF סוכר סינטקטי - כמו למשל כתיב של {}, המיצג 0 או יותר חזרות של מה שיש בפנים.  
פורמליזום זה דומה בעיקרו לפורמליזום של דקדוק BNF, אלא שהוא של כלל הגזירה יכול להיות רגולרי מעל אוסף הסימנים הסופיים והלא סופיים כאחד.

שימוש בביטויים רגולריים כאלו הוא בבחינת Syntactic sugar לדקדוקי BNF.  
ההרחבה עצמה (EBNF) אינה מאפשרת הגדרת שפות פורמליות נוספות פרט לאלו הנינתנות להגדרה בדקדוק BNF,  
אך ניתן באמצעות הרחבה זו להגדיר שפות פורמליות נוספת תמציתות.  
EBNF גם כן מוגבל, לדוגמא לא ניתן לבדוק אם משתנה הוגדר קודם לכן, או אפשר לבדוק שמספר הארגומנטים  
לפונקציה הוא תקין וכו'.

"הערות" לא מופיעות ב- BNF ולא ב- EBNF.

דוגמה ליטרניים בכתיבה EBNF:

- {} – מסמנת חזרה של אפס או יותר פעמים של הביטוי שנמצא בתוך הסוגרים המסלולים.
- לדוגמה: identifier {,identifier}
- [] – ביטוי אופציוני עטוף בסוגרים מרובעים.
- לדוגמה: [label-declaration;] [label-declaration;] אומר ש- label-declaration הוא אופציוני.
- לעיתים יש כללי קדימיות לדוגמא:
  - Identifier | record field-list end
  - מתפרק ככה:
  - Identifier | (record field-list end)
  - ולא ככה:
  - Identifier | record field-list end

## מה השווא והשונה בין שפות תכנות ובין שפת הביטויים הרגולריים<sup>3</sup>:

"שפה הביטויים הרגולריים הוא שפה פורמלית (בניגוד לשפה טבעית) גם שפת תכנות היא שפה פורמלית.

שפה פורמלית היא שיש חוקים פורמליים הקובעים כיצד מילה שייכם לשפה, ומה המשמעות של כל מילה (תכנית).  
בשפה תכנות המשמעות של כל מילה היא הוראות לביצוע במחשב, אבל שפת הביטויים הרגולריים אינה שפת תכנות. המשמעות של מילה בשפה אינה הוראות לביצוע במחשב".

## מה השווא והשונה בין שפות תכנות ובין דקדוקים חסרי הקשר<sup>4</sup>:

<sup>2</sup> נלקח מהסיקום של אריס קלקה – סיקום הרצאות 1-2.

<sup>3</sup> נלקח מהאתר Safot.net – [קישור](#).

<sup>4</sup> נלקח מהאתר Safot.net – [קישור](#).

"גם דקדוקים חסרי הקשר הם שפה פורמלית", וגם הם אינם שפת תכנות. גם ביטויים וגולרים, ממשימים ככלី עוז להגדירה הפורמלית של שפות תכנות. הם אינם משמשים כדי להנדר את המשמעות של תכנות בשפה תכנות, אלא רק להגדרת דקדוק, ככלומר אלו תכנות הן חוקיות ואלו לא".

#### האם ביטויים וגולרים הם דקדוקי חסרי הקשר?<sup>5</sup>

"כל ביטוי רגולרי מגדיר מה שקרי "שפה רגולרית". בהקשר זה שפה היא אוסף של סדריות.

דקדוק חסר מגדיר "שפה חסרת הקשר". גם שפה חסרת הקשר היא אוסף של סדריות.

כלראות של שפה רגולרית היא שפה חסרת הקשר: בעבר כל ביטוי רגולרי, אפשר לבנות דקדוק חסר הקשר שמנדר את שפה רגולרית.

- חוק כמו , `<A><B><C>` שיכל להופיע ב- EBNF, אינו חוק אפשרי בדקדוק רגולרי.

- ניתן לדבר על המונח "דקדוק רגולרי":

גם דקדוק רגולרי מגדיר שפה רגולרית. לא ברכנו בהרצאה על דקדוקים רגולרים. בדקדוק רגולרי חוקי הגזירה מוגבלים יותר מאשר חוקי הגזירה של דקדוק חסר הקשר. אפשר לומר של דקדוק רגולרי הינו דקדוק חסר הקשר אבל לא להיפך".

#### מבנה הבניין של שפות התכנות:

- **Nameable** – סוג של יישות, כמו פונקציות, טיפוסים, קבועים ומשתנים, שהמתכנת יכול לתת להם שם.

:Nameable types in C -

```
//Type named "struct Date"
struct Date {
    int Month,day,year
}
```

:Nameable values in Pascal -

```
CONST
Pi = 3.14159
```

• **מזהים (Identifiers)**: הם שמות של יישויות בשפה – לדוגמה שם של משתנה, שם של קבוע, שם של טיפוס, שם של ישוות אחרות המופיעות בשפה. השמות הללו מזהים ישות ומאפשרים לפנות אליה אחרת שנוצרה.

- הצורך במזהים נובע מהיות התכנות מודולרי. יש צורך לתת שמות לאוסף תהליכי חישובים שהתוכנית מבצעת ואו להשתמש באוסף זה בטור אוסף גודל יותר.

- כל שפה מגדירה מאייה תווים יורכב מזהה.

#### סדריות (סדריות): String.

- **מילולון**: דבר שמצוין את עצמו. כמו למשל מספר. נבחין כי מילולון הוא ערך. כלומר: סדרית, Integers, Characters וכו'.

○ לדוגמה ב- `x=3` המילולון הוא 3 – מילולון של Integer. כלומר, מילולון הוא לאו דווקא סדרית.

○ פונקציה אונומית מהצורה: `x = fn` היא מילולון – ישות שמצוינת את עצמה ואין לה שם.

▪ בשפת C היבים לחת שם לפונקציה (כלומר מזהה), בעוד שהפונקציה עצמה תהיה מילולון.

#### הערות

#### סימני פיסוק ( )

- **밀לים שמורות** המשמשות כפקודות, הערות ועוד.

• פעמים רבות ישנה הבלטה ויזואלית של מרכיבי דקדוק בשפה.

#### סוגי האסימונים ( Terminals - Tokens

<sup>5</sup> נלקח מהאתר Safot.net – קישור.

<sup>6</sup> נלקח מהאתר Safot.net – קישור.

Kind	Example	Denotes?
Identifier	<code>main, i, printf, argv</code>	a "nameable"
Literal	<code>"132", "Hello, World\n"</code>	itself
Operator	<code>*, +, *, /</code>	a builtin function
Punctuation	<code>;, , (, )</code>	nothing (reading and parsing aide)
Reserved word	<code>if, class, int</code>	...
Reserved identifier	<code>int, class, int</code>	primitive type
Predefined identifier	<code>Integer, &amp;primitive type</code>	
Comments	<code>/* fubar */</code>	Nothing!

שאלה בנושא:

- **איזה Tokens מציינים ישות תכונתית ואילו לא?**<sup>7</sup>

פתרונות:

אסימונים שמצוים ישות תכונתית:

- **מזהים** – מתייחסים לדברים שנמצאים בזיכרון כמו פונקציות, ערכיהם של משתנים וכו'.
- **אופרטורים** – בשפה המאפשרת פולימורפיזם ובפרט פולימורפיזם אד הוק עם העמסה, כמו `+ C++`, אפשר להעMISS על האופרטורים ולהתייחס אליהן כאל פונקציות הנמצאות בזיכרון לכל דבר ועניין. לכן יש בהם התייחסות לשיטת תכונתית.

אסימונים שלא מזהים ישות תכונתית:

- ליטרלים
- סימני פיסוק
- מילים שמורות
- הערות

**המילים שמורות ( Reserved Words|Keywords ) :** הם מילים אשר למרות שהן עומדות בכללים הקובעים מהו מזהה, הם שמורות למטרות אחרות ועל כן נוכל להשתמש בהן למתחם שם לשימוש שיצרנו (המחנכים). לדוגמה בשפת C המילה `int` –

עומדת בכללים המגדירים מזהה חוקי בשפה, אך היא אינה מזהה. לא ניתן לכתוב פונקציה בשפת C אשר שמה הוא `.int`.

- מילים שמורות משמשות לעיתים כמו `סימני פיסוק` – לדוגמה בפסקל המילים: `program, begin, end`:

#### • **סוגי המילים שמורות:**

- **סימני פיסוק.**
- לדוגמה בפסקל המילים: `program, begin, end`.

#### • **פקודות.**

- לדוגמה בשפת C המילה `.return`.

#### • **מבנה.**

- לדוגמה בשפת C המילה `.struct`.

#### • **типовים אוטומטיים.**

- לדוגמה בשפת C הטיפוס `.int`.

#### • **מזהים שמורות ( Reserved Identifiers ) :** מילים שמורות המשמשות כמזהים.

לדוגמה מילים שמורות המשמשות לזיהוי ישויות "אוטומיות".

▪ שמוזהה את הטיפוס האוטומי של מספר שלם.

▪ לא כל המילים שמורות הן מזהים שמורות.

לדוגמה:

▪ Atomic command – C – **Return**

▪ סימני פיסוק – **begin, end, program**

▪ אופרטור לייצור טיפוסים מורכבים מטיפוסים אחרים. – C – **struct**

**מזהה מוגדר מראש ( Pre-defined Identifier ) :**

- זהו מזהה המקשר לישות כלשיי במשפט התוכנות (כמו טיפוס, פונקציה, פרוצדורה או ערך).
- בזהה מוגדר מראש המתכנת יכול להשתמש לפי בחירותו לצרכים אחרים – ולקשר אותו לישות אחרת.
- ההבדל בין מזהה מוגדר מראש הוא שמדובר בהשתחמת בשמו לצרכים אחרים (בזהה שלו) כמו לדוגמה `Integer` כמשמעותו אחר בפסקל. אבל מזהה שמור – לא ניתן להגדיר בדרך אחרת.
- מסבך את הגדרת השפה שלא לצורך וכובל אותה מפני שינויים עתידיים.

**הערות נוספות בהקשר זה:**

- return היא פעולה אוטומטית שלה הוקדשהמילה שומרה בשפת C.
- שפת קובל מעניקה להרבה פעולות מזהים שמורות - יש מיללים שמורות יהודיות בעבר חיבור, חיסור, כפל וחילוק.
- `WriteLn` זוהי שגורה הבנויה בשפת התוכנות פסקל ואשר למתכנת אין גישה אליה. משמעותה של השגורה הוא חלק בלתי נפרד מהגדרת שפת פסקל. אך זו אינה מילה שומרה, אלא מזהה מוגדר מראש – המתכנת יכול לקשור את המילה לשותה אחרת.
- האפן שבו שפת C תומכת בקלט ופלט שונה מהותית מזו של שפת פסקל. ב- C הפונקציה `printf` הוגדרה בספריה. ספריה זו ניתנת להחלפה – והמתכנת יכול למשה מחדש.
- למעשה בשפת C אין מזהים מוגדרים מראש כלל.

**דוגמאות נוספות:**

סוג	דוגמא	שם מונע מפני שימושים נוספים	מימוש מפני למתכנת ובר	מימוש מפני למתכנת ידו	העבורה	מבנה בסביבה
מזהה שמור	<code>print</code> בשפת AWK	מזהה שמור	כן	לא	כן	
מזהה מוגדר מראש	<code>WriteLn</code> בשפת פסקל	מזהה מוגדר מראש	כן	לא	כן	
מזהה ספריה	<code>printf</code> בשפת C בתבוניה 1.2	מזהה ספריה	כן	כן	כן	
מזהה אחר	<code>HelloWorld</code>	מזהה אחר	לא	לא	לא	

**ספריה:**<sup>8</sup>

אוסף של תחכום המשמש לפיתוח תוכנה. אוסף זה כולל מידע, הגדרות או קוד אשר מספקים שירות לתוכניות הייצוגיות.

על מנת להשתמש בתוכן של ספריה יש צורך בד"כ לטעון אותה ע"י פקודה הנינתנת ל- Interpreter או לקומפיאילר (לדוגמה `include` בשפת C).

**מחולקת לשניים:**

**הספריה הסטנדרטית, Standard Library** – ספריה הממשת לוגיקה או תוכנה אשר דרושה בחלק גדול של התוכנות (למשל קלט-פלט). מתכני השפה רואים בספריה זו ישות נפרדת מהשפה, אך ראו צורך לספק שירותים מסוימים למתכנים המשתמשים בשפה, באמצעות ספריה זו. כדי להשתמש בספריה זו ובתוכנה יש לייבא אותה לתוכנית.

**הספריה המובנת, Built-in library** – ספריה זו גם מספקת שירותים למתכנת, אך מתכני השפה מתייחסים לשפה זו כחלק משפט התוכנות על מנת להשתמש בישיות מספריה זו, אין צורך ליבא את כל הספרייה אלא לנקוב במזהה הישות.

**הבדלים העיקריים בין ספריה מובנת לספריה סטנדרטית:**

- אוף הייבוא של המזהים מהספריה – בספריה הסטנדרטית יש צורך ליבא את הספריה לפני השימוש במזהה המצויה בה. בספריה המובנת – נקייה במזהה ללא צורך לציין את שם הספריה ממנה הואלקח ואו לבצע טעינה של ספריה זו.
- התיהסות מתכני השפה בספריה – ספריה סטנדרטית נחשבת יישות נפרדת מהשפה, כדי עוז למתכנת. בעוד ספריה מובנת – נחשבת כחלק מהשפה.

**מזהה ספריה:**

- אוסף של רוטינות (או מודולים) שנכתבו מראש ומיינים למתכנת:

- ניתנת להחלפה, כמו ב-C.
- לא ניתנת להחלפה, כמו בפסקל או AWK.

### Replaceable vs. Built-in library

What's Better?

Replaceable	Built-in
• Troublesome for programmer	• Less work for the programmer
• Small language manual	• Large language manual
• Adaptive	• Not adaptive
• Makes language design more modular	• Library is typically small
• Library can be very large	• Excellent for PL designed for beginners and for one-liner/scripting PLs.
• Most modern languages	

○

סוגי רוטינות:

○

▪ Low level – כאלה שלא ניתן למש בשפה.

▪

▪ Essential – שאין טעם שככל מתכנת יכתוב אותם מחדש.

▪

▪ Tiresome – אין טעם להטריד את המתכנת בミומש.

▪

- כאשר נעשה שימוש בשפה במוחה ספרייה ישנו צורך ליבא מזהים אלו באחת מהדריכים: באמצעות מילה שמורה, עיבוד מקדים (טרנספורמציות טקסטואליות לקובץ טרם ההידור) או ייבוא אוטומטי (ע"י המהדור, כמו ב-Java).

### סיכום סוגים מוחאים:

#### - החזקים ביותר - מזהים שמורים, מוגדרים מראש:

- זמינים תמיד.
- לא ניתן למתכנת לבחון את השימוש.
- נאשר על המתכנת לשנות את השימוש.
- נאשר על המתכנת להשתמש במזהים אלו לצורך אחר.

#### - החלשים יותר - מזהים שמקורם ממקור,

- מזהה שמקורו לשוט כתשי בשפת התוכנות.
- הותר למתכנת לקשר אותו לשם שימוש אחר.

#### - הći חלשים - מזהה ספריה:

- המתכנת יכול לבחון את השימוש שלהם ולשנותו.
- המתכנת יכול להשתמש בהם לצרכים אחרים.

### הגישות השונות לנקודת התחלה וסיום של תוכנית:

ישנה הגישה האוטרכית (השפה קובעת את נק' התחלה) שבה צריך לציין את תחילת התוכנית בפירוש. אחת המילים השמורות קובעת את קבוצת הפקודות שתבוצע ראשונה. דוגמאות לכך: בפסקל - המילה program. ב-AWKBegin. ב-CBegin.

- מה שורשם בסיכון של איריס קלקה – נקודת התחלה יכולה להיות גם השורה הראשונה. כשציינתי לפני פרופ' גיל את הנושא הוא אמר שהוא שchar שחייב שפה אחרת המשמשת המושג. לא קיבלתי תשובה חד משמעית.
- היקף התוכנית – התוכנית תמיד תהיה בקובץ בודד.
- השימוש בשיטה זו בפסקל לא מפסיק אותנו לאור העובדה שככל התוכנית תהיה כתובה בקובץ אחד.
- התווויות הגבולות של התוכנית בפסקל או ב-AWK היא פשוטה – אין בתכנית דבר מלבד הכתוב בקובץ היחיד שבו היא מצויה וכל מה שבקובץ זה הוא חלק מהתוכנית.

ואילו ישנה הגישה המטאפיסית (סביבת העבודה קובעת את נק' התחלה) שבה הקומפיילר מסיק בעצמו את נק' התחלה והסיום של התוכנית. בגישה זו הביצוע יתחיל לרוב מפונקציה בעלי שם מסוים, יש לציין שם הפונקי' הווא לא מילה שמורה ויתריה מכך לא מוגדר בשפה. סביבת העבודה היא זו שלבסוף מכירעה.

- דוגמא לכך היא שפת C.

- רוב השימושים שלמה משתמשים בשם "main" כדי להגדיר את נקודת התחלה.

- המימוש של C ב-windows משתמש בשם ".WinMain".
- **סביבה העבודה - הקומפילר** – קובעת איזה קבצים ירכיבו את התכנית.
- הגישה השילשית והאחרונה היא הגישה הholistic (המתכונת קובע את נק' ההתחלת), שאלת מטריה לסבירות היפות להגדר את נק' תחילת הריצה, אלא מעניקה למתכונת את השליטה על כך. למשל בשפת AiPIL המתכונת יכתוב קובץ מאגד שיגדר את סביבת העבודה. שפת Java היא דוגמא נוספת לכך.
- השפה מגדרה איך לזהות את הקבצים שמרכיבים את התכנית ואת נקודות ההתחלה. הגדרות אלה תלויות בסביבת התכנית.

#### **חישוב אינטראקטיבי:**

- תחילת החישוב היא בפקודה הראשונה אותה מקלט המשמש אל הזרז. בחישוב אינטראקטיבי יש עדין צורך להעתות את גבולות התכנית, כולל לקבוע מה כלול בתכנית ומה מחוץ לה. הקביעה נעשית על ידי סביבת העבודה (התכנית) שבדוגמה שבעמ' 21 ב"צדדים ראשונים"). והוא יכולה להיות אוטרקטית, מטאיפיסית, או הוליסטית.

#### **בצד מובלילות זו מזו הפקודות בתחום הבלוק, סוג הדקדוק השונים:**

- **דקדוק ספראטיסטי:**
- הפקודות בתחום הבלוק מופרדות על ידי מפריד, סימן מיוחד (כל שתי פקודות מופרדות ע"י מפריד). אם בבלוק יש מ פקודות או יש בו 1-מ מפרידים.
- אם בבלוק יש פקודה אחת בלבד – אז אין בתחום הבלוק אף מפריד.
- הקושי בדקדוק הספראטיסטי הוא בכך שבמהלך כתיבת התכנית יש שינויים תדריים בפקודות אשר בבלוק. לא ניתן להעיר פקודה מבלוק אחד למשנהו מבלתי לבדוק כי מספר המפרידים הן במקור ובמטרה ממשך להיות קטן באחד מספר הפקודות.
- נבחין כי סימן הנקודה פסיק אינו(!) חלק מהפקודה – בנויגוד לדקדוק הטרמיניסטי.
- דקדוק ספראטיסטי מקל (פסקל) – מתיר גם פקודה ריקה:**

```
begin
    writeln('hello,world')
end.
begin
    writeln('hello, world'); // between this line and the "end."
        there is an empty command
end.
```

- **דקדוק טרמיניסטי:** (נהוג במעט כל שפות ה: {})
- סימן הנקודה פסיק אינו מפריד בין פקודות, אלא מסיים אותן – (Terminate).
- סימן הנקודה פסיק הוא חלק בלתי נפרד מהפקודה.
- **דקדוק ספראטיסטי-טרמיניסטי:** הפקודות מופרדות ע"י סימן הנקודה-פסיק וניתן, לפי בחירת המתכונת, להוסיף סימן הנקודה-פסיק בסוף סדרת הפקודות. מיזוג של הדקדוק הספראטיסטי הרגיל והטרמיניסטי.
- **דקדוק לייברלי:** (שפת Eiffel, Go, AWK)
- סימן הנקודה פסיק הוא אופציונלי לא רק בתום הסדרה של הפעולות אלא בתום כל פקודה.
- המהדר מסתמך על ההנחה שבעל שורה יש פקודה אחת לכל היוטר – וכך הוא מסיק סיום פקודה (הנחה שאינה חיונית).
- שפת Go מגדילה לעשוות ומוחקת באורה אוטומטי מקובץ התכנית סימני נקודה פסיק אשר אותן יכול המהדר להסיק בעצמו.

**גדר** – מציין סדרה שלתו אחד או יותר, שמשמשת גם כפתחה וגם כבריח (של בלוק). גדר מפרידה בין שני קטועי טקסט הגובלים זה זהה.

**בעיית גידור הגדר** – היכולת הגדר עצמה בתחום תחת הסדרית המסומנת ע"י הגדר.

#### **פתרונות בעיית גידור הגדר:**

- **AMILOTS** – שימוש בתו מיוחד שנותן משמעות מיוחדת לתו שמוופיע אחריו.

- מאחד המבחןים<sup>9</sup>: "밀וט הוא טכניקה לחתת לתו או לסדרת תוים משמעות השונה מהמשמעות הרגילה שלהם".
- יתרון של שיטה זו היא אפשרות לכלול בתחום המילולון תוים מיוחדים אחרים – אשר להם אין ייצוג גרפי או שהייצוג הגרפי שלהם יכול להיות מטעה. לדוגמה: "מ'" המייצג מעבר לשורה הבאה.
- מה משמעותתו מילוט שבא לפניתו שאין צורך במלוט? מתכוון של השפה צריך להחליט על ההתנגדות. לכל בחירה של מתכוון השפה יש צד אחר:
  - התעלומות מילוט שלתו שבסה יכולה להביא להタルחות מסווגות אמיתיות של המתכוון.
  - התרעעה על מילוט שכזה יכולה להטייל מעסמה נוספת על המתכוון אשר י策ר להיזהר מילוט תוים שאוותם אין למלוט.
- מילוט המילוט – מתן משמעותו לתו מילוט כתו מן המניין.
- הכפלת הגדר.
- שיטה "נקיה" יותר מילוט, אך בהעדרתו מילוט – יש להשתמש באמצעות אחד כדי לכלול במילולון תוים נעדרי ייצוג גרפי.
- בנוסוף יש קושי בבנייה ביוטיים רגולריים בשיטה זו.
- ריבוי אפשרויות מסגור – יותר מגדיר אחת אפשרית.
- לדוגמה בהן ניתן למסגר מילולון הנק בין זוג של גרשימים בודדים והן בין זוג של גרשימים כפולים – שיטה זו חלשה יותר משיטת המילוט מסוימת שהיא לא תומכת במילולון המכיל הנקתו של גרשימים בודדים והן הנקתו של גרשימים כפולים. ולכן יש שפות המאפשרות את האופציה "לקבוע כל תוך כדי נגיד למילולון".
- אפשרות לקבוע כל תוך כדי נגיד למילולון (שפת פרל).
- הקושי היחיד בשפה זו היא בכתיבת מילולון המכיל את כל התווים כולם.
- כתיבה בפתחה של המילולון את אורכה של התת סדרית.
- יש שפות המאפשרות לכותב להגדיר בפתחה של העטרה את הבריה שלה (מנגנון נדר).

#### שאלות הקשורות לנושא:

##### - מדווקא אין מילוט בתחום הערות שורה?<sup>10</sup>

"אין מילוט בתחום הערות שורה ממשום שהעטרה שורה מתייחסת לכל השורה הנוכחית ולכן אין משמעותו לקינון הערות שורה מכשאם נראה בתחום הערטה שורה את הפתיחה של הערטה שורה נדע שהקשרו הוא כפирושתו רגיל ולא פתיחת שורה, עברו הבריה גם אין צורך במילוט מסוום שהבריה הואתו ירידת שורה והתו הנ"ל לא יכול להופיע בהקשרו הרגיל בתחום הערטה שורה ממשום שפירוש הדבר היה שהעטרה שורה מתרפית על יותר משורה אחת בניגוד להגדלה. עברו כלתו שאינו הבריה או פתיחת אין סיבה למילוט ממשום משמעותו ברורה."

##### - הסבר מדוע בשפה שיש בה הערות מקוננות יתכן צורך בamilot הפתיחה כמו גם מילוט הבריה?<sup>11</sup>

"שפה בה יש הערות מקוננות לא נוכל להבדיל בין שימושה בתחום הפתיחה כפתיחה של העטרה מקוננת או בשימושו כתו מן המניין כנ"ל לגבי בריה שלא נדע אם סוגר את העטרה או גם הוא בפירושו הרגיל. אם העורות אינן מקוננות, נעלם הצורך במילוט הפתיחה. הסבר." כאשר מובטח שאין הערות מקוננות כל מופע של תו הפתיחה בתחום מתייחס לפירושו כתו רגיל ללא משמעות מזוהה לגבי הבריה הבועה לא נפתחה ממשום שעדין לא נוכל לדעת אם משתמשים בבריה בפירושו הרגיל או בסוגר של העטרה".

##### - ומה בדבר מילולוני סדרית?

"בamiloloni סדרית יתכן שהפתיחה יהיה שונה מהבריה, אבל אין צורך במילוט, ממשום שאין קינון של מילולוני סדרית".

##### - תן סיבה מדוע שפה ירצה להוסיפה מילוט בתחום הערות?<sup>12</sup>

"צריך מילוט בתחום הערטה כדי לאפשר מצב של העטרה בתחום הערטה, ולמה שנרצה העטרה בתחום הערטה? אם אתה רוצה לשים קוד שלם בתחום הערטה ובתוך הקוד זהה יש גם הערטה אז יוצר מצב של העטרה בתחום הערטה ואם אין מילוט של הערות אז הסוגר של הערטה הפנימית ביותר תסגור גם את הערטה החיצונית".

##### - מדווקא יש להשתמש במילוט בסדריות?<sup>13</sup>

<sup>9</sup> מבחן חורף 2013-2014 – מועד א' – קישור לפתרון המבחן – [כאן](#).

<sup>10</sup> נלקח מהאתר [Safot.net](#) – [קישור](#).

<sup>11</sup> נלקח מהאתר [Safot.net](#) – [קישור](#).

<sup>12</sup> נלקח מהאתר [Safot.net](#) – [קישור](#).

<sup>13</sup> נלקח מהאתר [Safot.net](#) – [קישור](#).

AMILLOT MESHAUOT MATHON MESHAUOT ACHERET LETO AO LAASIMON (token) SHL SHFAT HATCNOOT, ASHER SHONA MAHAMSHMEOOT HARIGLAH SHL OTTO TO (ASIMON).  
HAPLUT OTTO MIYICROT HATCNOOT MIYAZG BDERAK KELL CAOSOF SHL SDORIOT BTOUR HATCNOOT. BSHFOT HATCNOOT SHMUYADOT LISHIMOSH KELLYI (general purpose) NDRESHAT HAMICA BHTCNOOT SHICOLOT LIYIZR FPLT CALSHAO. MISIBA ZO, OMISIVOT ACHROT SHL SHLOMOT HAGDORT HESHPA, NDRESHAT YKOLAH LEGDOR SDORIOT SHMCILOT KLL TOCON SHAHO, KLOMER KLL SDORAH SHL TOVIM.

HAMILLOT BSDORIOT UL CN NDRASH MISHTI SIVOBOT:  
HARASHIT, YSH ZORAK BMANGNON LYIZR SDORIOT SHATHIYNA MSOGLOT LEHCIL AT SIMAN HAGDOR SHL HSDORIT, OTTO YSH LMLAT. SHNAYI, YSH ZORAK BMANGNON LYIZR SDORIOT SHMCILOT TOVIM ASHER AIMIM MZOIVIM BMKLOTH, AO APILO CALO SH SHAINON LHM YIZOG GRIFI. LISHM CK, MSHTHMSIM BEURD HONOMRI SHL HTO (LMASHL LEPI ASCII AO (Unicode). YSH ZORAK UL CN BAMILLOT SHL HSPOROT BTOKH SDORIOT. HAMSHUOT HARGLAH SHL HSPOROT HIA TOI HSPOROT.  
HAMSHUOT ALIYA NUSAHA HAMILLOT, HIA YIZOG NOMARI SHL TO CALSHO BMEURCAT HATOVIM."

#### HSBVR MDOUZ AIN ZORAK BAMILLOT BAMILLOLONIM MASFPIIM.<sup>14</sup>

"SHNI MERCAVIM LHTSHOVA HNGCNOH:

1. BAMILLOLONIM MASFPIIM AIN FTIH, BRICH AO GDR.
2. AOSOF HTOTOIM SHBHM COTBIM MYLLOLONIM MASFPIIM HOA MOGVEL: HSPOROT (OLLA H"MASFPIIM" CPI SHACHDIM KRAO LHM), SIMAN HNKODA HUASHRONITA, FLOS VMINOS, VGM HAOT E AO E. CDI LYIZG MASFPIIM, AIN ZORAK LTHT MSHUOT ACHROT LAFF AHD MAALO.

ZOHI HTSHOVA HRSIMIYAH SHPOROSMA BMACHON - BUBRIT PSHOTAH: HTPKID SHL MYLLOT BAMILLOLONIM HOA LAAPFSR LHCNNS TOVIM SHLA KIYIMIM BMKLOTH, LMASHL YRIDAT SHORA (a), TAB (t), AO TOVIM SHISH LHM TPKID SHL FTIH, BRICH AO GDR BAMILLOLONIM (SHBHALUT HEM HOA SIMAN HGRSHIM, VLCN CDI LCTHOB BMFORSH GRSHIM BTOUR MYLLOLON NCHTOV"). HITYOT SHAINON SHOM TCHLIT BHTCNOOT HALLA BAMILLOLONIM SHHM MASFPIIM BLBD, CYOUN SHISH YHSHTI MATT TOVIM SHWSIM BHM SHIMOSH BAMILLOLONIM MASFPIIM, AIN SHOM ZORAK BAMILLOT".

#### TAR AT HAMILLOT BAMILLOLONI SDORIOT (String) BSHFAT C BAMILIM SHL<sup>15</sup>

"MYLLOT NUSAHA BSHFAT C BTOKH SDORIOT VBTOKH MYLLOLONIM MASFPIIM. HAMILLOT NUSAHA AMTZUOT TOH HAKU HNTOI HHPFK (SHRBIM CHTBO BTUTOT CD /), SHMSH CDI LHCNNS TOVIM CGON: \n, \r, \t, \b, \", \'  
VGM CDI LHCNNS TOVIM BAMTZUOT KOD H ASCII SHLHAM".

CHTIBAH "HURROT" BHTCNOIT:

- BD'C MUBED HESHPA MATHULIM LCHLOTIN MAHAUROT VHN MOSHLCOT BSHLB MKDM SHL HUIVOD.
- BSHFOT MODERNIOT YOTHR YSH MAGMA SHL MATHN MSHUOT HURROT - HURROT YKOLOT LHIZOT MUOBODOT UL MNT LYIZR TIUOD SHL HATCNOIT.
- LDOGMA: BSHFAT Java HURROT HMTCHILOT BFTIH \*\*/ MUOBODOT LCDI YZIRAH TIUOD HATCNOIT.

#### HMRCVIM SHL HAGDORT KBVZAHA RKKRSVIBAH:

- ATOMIM: LDOGMA HMMHROZOT HRICKA.
- BNAIM: CIZZ LYIZR AIBRIM MORCAVIM M- MEMBERS SHBNNO KODIM LCN VATOMIM. CLLL HYZIRAH NOTL AHD AO YOTHR MAIBRIM HKBVZAHA, VBVNA AIBR AHR.
- MNGIMLIOT: HAGDORT RKKRSVIA UYI KBVZAHA MINIMALIT SHL ATOMIM VPEULOT BNYA SHICSO AT CL HBNIA.
- MHSHKPFIM:

<sup>14</sup> NLKCH MHATR Safot.net - KISHP.

<sup>15</sup> NLKCH MHATR Safot.net - KISHP.

- Atomic command יכול להיות  $a=s*\sin(b+1)$  כי הפעולות המתמטיות מהצורה:  $s=a$
- Arithmetical Expressions – שיכוח לתחום ה-
- Atomic reference Expressions – הימן הינוreference – השם ו-
- Java – בה הטיפוסים מוגדים בזורה רקורסיבית: Type Constructors class, enum.array.
- הטיפוסים האטומיים הם מילים שמורות בשפה.

- *Arithmetical expressions.*

**Atoms** literals, references to named entities....

**Constructors** mathematical operators, user-defined functions,...

- *Executable statements (commands) in C.*

**Atoms** assignment, `return`...

**Constructors** `if`, `for`, `{...}`...

- *Types in C.*

**Atoms** `int`, `char`...

**Constructors** aka *type constructors*

- “points to”,
- “array of”,
- “record with fields”, *and*,
- “function taking type  $\tau$  and returning type  $\sigma$ ”.
- ...

שאלה שקשורה לנושא:  
**תיאולוגיה יהודית רקורסיבית – מי הם היהודים האטומיים?**<sup>16</sup>  
**כידוע, יהודי הוא כל מי שאמו יהודי או שהתגיר.**

- א. כתוב תיאור זה באמצעות שלושת התנאים לתייאור קבוצה מוגדרת רקורסיבית.
- ב. לאחר שעשית זאת, חזר ועינך במה שכתבת, והסביר מדוע יש שני ואրיאנטים להגדלה – האחד שבו יש אטומים במספר לא חסום, והאחר שבו יש בנאי בעל 0-ארגומנטים.
- ג. אטומים: שרה – האם היהודי הראונה.  
מבנה: א. לידא לאם היהודי. ב. גיור.
- ד. מינימליות: כל אדם שאינו שרה, לא נולד לאם יהודי או לא עבר גיור – אינו יהודי.
- ה. ההבדל בין שני הוריאנטים הוא כיצד אנו מתייחסים לאדם שגוי: האפשרות הראשונה היא להתייחס אליו בהתאם לבניין (שלא מקבל ארוגומנטים), או בתור אוטם חדש, המגדיר את קבוצת האטומים כקבוצה לא חסומה המכילה את כל המgoירים".

### :Compound vs Atomic Members

- אטומים: לא ניתנים לפירוק למרכיבים שהם members עצמם בעצם.
- פקודה אטומית (Atomic command )**: פקודה שלא ניתן להזותה בתמי פקודות<sup>17</sup>. היא יכולה להכיל בתוכה expression מרכיב, זה לא גורם לפקודה לא להיות אטומית.
- ביטוי (expression) לא מכיל commands (לפחות לא בפסקל).
- Compound members**: לרוב ניתנים לפירוק לצורה של עץ עפ"י החוקים של הבניה.
- Constructors**: מצוינים ע"י מילוט מפתח, עליהם ניתן לחשב כעל דגשים\שמות של הבנים.

נתבונן כיצד בהגדלה הרקורסיבית של אוסף הפקודות בפסקל:

#### אטומים – הפקודות האטומיות:

- הפקודה הריקה.
- פקודת הצבנה.
- קראיה לפורצתורה.

בנייה – כללי הייצור העיקריים של הפקודות הם:

<sup>16</sup> נלקח מהאתר Safot.net – קישור.

<sup>17</sup> מה-56 שאלות ותשובות של אריה.

- שרשור של פקודות המופרדות בעזרת סימן הנקודה ופסיק (;) הוא פקודת.
- פקודת תנאי, כפי שתוארה לעיל, היא פקודת, אשר מכילה בתוכה פקודת אחת או שתיים.
- פקודת תנאי רבת ראים המוגדרת באמצעות המילה השמורה case.
- לולאות המתארות ביצוע איטרטיבי של פקודת (אוטומית או מורכבת), אף הן פקודות. יש בפסקל שלושה סוגים של פקודות לולאה מורכבות:

for      o  
while    o  
repeat until    o

שאלה:

**?Pascal – בין ההשמה  $a:=b$  לבין  $b:=C$  – מה ההבדל העיקרי בין ההשמה?**

פתרון (לא אוישר ע"י פרופ' גיל):  
הראשון ביטוי והשני פקודת.

ההשמה הראשונה גם תשוערך בסופו של דבר לערך החדש של a (במילים של הקורס: יחוור הערך של ה-L-Value (R-Value), בעוד שבספקל ההשמה לא תשוערך כלל)<sup>19</sup>.

בשפת C אופרטור הצבה גם מחזיר את הערך שהציב ולכן ההשמה בשפת C היא ביטוי (ביטוי מהזיר ערך, פקודות לא).  
בפסקל לעומת זאת, ההשמה אינה מהזירה ערך, ולכן זהה פקודת.

נזכיר כי בשפת C קיימות שתי פקודות אוטומיות בלבד:

א. פקודת הריקה.

ב. Command expression

במקרה הזה השימוש באופרטור = יוצר Command expression אשר יוצר פקודת מביטוי הצבה, אך אין הדבר שווה לפקודת הצבה שקיימת בשפות אחרות כפקודות אוטומיות.

<sup>18</sup> נלקח מהאתר Safot.net – קישור.

<sup>19</sup> נלקח מהאתר Safot.net – קישור.

## 2. ערכים וטיפוסים

### ערכים

ערך הוא ישות שקיימת **במהלך** חישוב וריצת התוכנית. ערכים הם **לא משתנים!**

- בפסקל ערך הוא כל מה שניתן לעבור ארגומנט לפונקציית פרוצדורה.
- ניתן להציג ערך למשנה.
- נבחן כי "טיפוס" (Type) הוא קבוצה של ערכים – ומגדיר סט פעולהות אותן ניתן להפעיל על הערכים של אותו הטיפוס.
- ישן שפות אשר בהן אין משתנים (פרדיגמה פונקציונלית) אולם מבון שיש בהם ערכים.
- קבוצת הערכים בשפה היא קבוצה מוגדרת רקורסיבית.

**משתנה (Variable)**: **ישות** **שיכולה להכיל ערך** ומספקת פעולות **"בדיקה"** ו-**"עדכן"** על המידע שבו.

- איזור בזיכרון אשר יכול להכיל ערך.
- משתנה יכול להיות לא מוגדר.
- ממושך בעזרת זיכרון כלשהו, כמו ראמ/דיסק קשיח.
- סוגים שונים של משתנים:
  - קבועים אבל הערכים שלהם יכולים להיות לא ידועים.
  - Mathematical variables ○
  - Mathematical variables ○
  - Logic programming variables ○
  - Imperative variables ○
  - יכולים להשנות עם הזמן, לדוגמה:  $n+1$ .
  - יכול להיות שלא יהיה להם ערך כלל.

נתבונן במבנה הערכים בשפת LISP (האםו של כל שפות התכנות הפונקציונליות):

An **S-Expression**<sup>6</sup> is

- ① An Atom
- ②  $(S_1 . S_2)$ , where  $S_1$  and  $S_2$  are S-expressions

An Atom is

- ① A string of any length (Many LISP implementations ignore letter case)
- ② The special value **NIL**.
- ③ An integer (non-essential)
- ④ A real number (non-essential)

Examples:

- *hello*
- *(a.NIL)*
- *(NIL.a)*
- *(Hello.(war.lord))*

- הביטוי שבודגמא יכול להיות מיוצג  
כע"ז בינהאי כאשר לחוליה-Ano  
קוראים cons, למצבי השמאלי car  
ולמצבי הימני cdr.  
- הערכים הם s-expressions!  
- לערכים אין טיפוס!  
- יש חוקיות לפירוק אחר בנית ערך:  
כל s-expression הוא או אטום או  
ביטוי שיש לו חוקיות בניה – כפי  
שנלמד בהמשך.

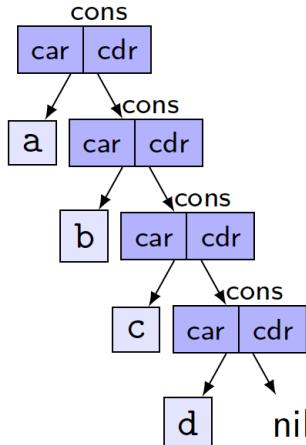
The list

(a b c d)

is shorthand (syntactic sugar) for

$$\left( a . \left( b . \left( c . \left( d . \text{NIL} \right) \right) \right) \right)$$

In binary tree representation:



**ביטויים (Expressions)** הם חלק מהתוכנית שמתורגם לערך בזמן ביצוע התוכנית. גם הם קב' מוגדרת רקורסיבית.  
הבנייה של ביטויים הם קריאות לפונקציות.

## טיפוסים – Types

**טיפוס (Type)** הוא קבוצה של ערכים.

כל טיפוס הוא תת קבוצה של קבוצת עולם הערכים של השפה.

כਮון שגם טיפוסים הינם קבוצה מוגדרת רקורסיבית, כאשר יש לנו בשפה טיפוסים אוטומטיים וכן בניאי טיפוסים.  
כל טיפוס T מגדיר את  $T(S)$ , קבוצת הערכים מסווג T.

"כל טיפוס הוא סט ערכים, אולם לא כל סט ערכים הוא טיפוס". טיפוס מגדיר לא רק סט ערכים, אלא גם סט פעולות אותן ניתן לבצע על אותו הטיפוס

לפעמים ערך יהיה שיך ליותר מטיפוס אחד, ולעתים יהיה שיך לאינסוף. לדוגמה "0" הוא ערך ששיך לכל טיפוסי המצביעים בספט C.

השימוש בטיפוסים נעשה למשל כאשר נפעיל אופרציה על ערך מסוים כלשהו, נבדוק האם האופרציה יכולה בכלל לפעול על ערך זה, נבדוק מה המשמעות של הפעלה כזו וכן ביטוי מסוימת טיפוס יתקבל מהפעלת האופרציה על הערך.

**בניאי טיפוסים** מאפשרים לנו ליצור טיפוסים מורכבים, ניתן לחושב עליהם **אופרטורים על קבוצות**.

### בנייה טיפוסים:

1. מכפלה קרטזית (Cartesian Product) :

$$T \times S = \{ (x, y) | x \in T, y \in S \}$$

לדוגמה טאפל. Record 2.

כasher  $i_j$  ליבלים (מוזהים) ו-  $T_j$  שם טיפוס.

3. העלאה בחזקה (Integral Exponentiation) :

כאשר כל איברי הטאפל הם מאותו הטיפוס:  $T^n$ .

לדוגמה טאפל שכל האיברים מאותו טיפוס, מערך struct של n אלמנטים מאותו טיפוס.

4. האיחוד הזר (Choice Type\Disjoint Union) :

$$T + S = \{ (\text{Left}, x) \mid x \in T \} \cup \{ (\text{Right}, y) \mid y \in S \}$$

לדוגמה ב-ML datatype

### פירות:

1. **מכפלה קרטזית** היא בנאי טיפוסים, כאשר גודל קבוצת הטעינאים שתחזק במכפלה קרטזית של שתי קבוצות טיפוסים היא מכפלת הגודלים של שתי קבוצות אלו. ניתן לבצע גם מכפלה קרטזית על יותר מקבוצה אחת, כאשר היא לעולם לא תהיה קומוטטיבית.

### Definition (Cartesian Product Type Constructor)

If  $T$  and  $S$  are types, their **Cartesian product** is a type denoted by  $T \times S$ , where  $T \times S$  is the set of tuples

$$T \times S = \{\langle x, y \rangle \mid x \in T; y \in S\}.$$

ע"י הגדלת תווית (מזהה) לכל איבר בתחום ערך מטיפוס מורכב (ע"י מכפלה קרטזית), נקבל טיפוס חדש והוא ה-Record.

### Definition (Records Type Constructor)

Let  $i_1, \dots, i_n$ ,  $n \geq 0$  be a set of unique labels drawn from  $\mathbb{I}$ , and let  $T_1, \dots, T_n$  be types, then,

$$\{i_1 : T_1, \dots, i_n : T_n\}$$

is the **record type** induced by  $i_1, \dots, i_n$  and  $T_1, \dots, T_n$

3. **העלאה בחזקה**: ע"י ביצוע מכפלה קרטזית כאשר כל איברי ה-tuple הם מאותו טיפוס נקבל בנאי טיפוסים שהוא בעצם **העלאה בחזקה** של טיפוס מסוים.

### Definition (Integral Exponentiation Type Constructor)

For a type  $T$  and a natural integer  $n$ , the **integral exponentiation** of  $T$  to the power of  $n$ ,  $T^n$ , is defined by

$$T^n = \overbrace{T \times \cdots \times T}^{n \text{ times}}$$

(מספר הסברים ובהמשך נפרט על בנאי טיפוסים נוספים – איחוד זור\ $\setminus$ Choice Type\Disjoint union)

#### **Unit**

- עברו העלה בחזקת 0 של כל טיפוס – נקבל טיפוס Unit.
- Unit הוא המכפלה הkartezית של 0 טיפוסים.
- ניתן לחשב עליו כעל:
- o ערך מורכב – נוצר מרכיב שרירותי.

- טיפוס אטומי.
- Unit הוא לא הקבוצה הרכיקה!
- ל- Unit יש ערך אחד ויחיד – ה- Unit = {()}.
- משותנה מטיפוס Unit הוא לא באמת משתנה, מכיוון שהוא יכול להכיל רק ערך אחד שלא ניתן לשינוי (ה- tuple הריך).
- מספר הביטים שלו הוא 0.
- Unit :C -ב-

### 239. Emulating Unit type in C

Option I: singleton enum

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

Option II: empty struct

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

*This type's (only) value is {}; it can only be used for initialization*

Option III: zero sized array (only in C++)

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

Unit מתאימה לטיפוס ה- "void", Incomplete – "void" יש מילה שומרה.  
נשים לב שעבור פונקציות שמקבלות void – כלומר מקבלות כפרמטר את הטיפוס UNIT ואשר מחזירות ערך –  
אנו למעשה שולחים להם את הטעפוף הריך – לדוגמה:

```
Long time(void);
Long t = time();
```

בדיקת שם שב- fopen אנו שולחים טאפל של 2 סדריות:  
File \*f = fopen("data.txt", "r");  
לכן, בהקשר של פונקציות אנו שולחים למשזה את הטעפוף הריך.

:Type Branding – בהמשך יש הסבר על המונח Type Branding – זה ישלים את ההגדלה:  
**Branding :Labels = Identifiers**

### 181. Branding type constructor

Let  $\mathbb{I}$  be an infinite set of identifiers (often called *labels*)

**Definition 3.8** (Branding). If  $T \in \mathbb{T}$  is a type and  $\ell \in \mathbb{I}$  is label then  $\ell(T)$  is *the  $\ell$  brand of  $T$*  where,

:Branding – האופרטורים ל-

### 182. Operators for branding

**Creation** A value of type  $\ell(T)$  is created from a value  $v \in T$

$$v \in T \Rightarrow \ell(v) \in \ell(T) \quad (3.3.19)$$

**Extraction** A value  $v \in T$  can be extracted from type  $\ell(T)$

$$\ell(v) \in T \Rightarrow \ell(v)\#\ell \in T \quad (3.3.20)$$

### מערכיים – :Array values

- ברוב שפות התכנות **array values** הם לא ערכים מדרגה ראשונה (פרק 3 מוסבר מהו ערך מדרגה ראשונה) שכן לא ניתן ליצור אותם ללא משתנים, לא ניתן להחזירם כערך החזרה ועוד.
- שפת C מבילה את ה- **:array values**
  - אי אפשר להעביר מערכיים לפונקציה.
  - אי אפשר להחזיר מערךים.
- שפת Pascal מבילה את ה- **:array values**
  - אי אפשר ליצור **array values** בלי עזרת "משתנה".
  - אי אפשר להגיד **array value** באזור של ה- **.const**.
  - אפשר להעביר **array values** כפרמטרים לפונקציה ופירושו הוא אבל אי אפשר להחזיר אותם.

### - הוא ה-Unit בשפת C: void

- פונק' שמחזירה **void**: נאמר עליה שהיא מחזירה רק אחד והוא תמיד אותו דבר ולא נאמר עליה שהיא לא מחזירה כלום.
- אין טיפוס **void** ב-C שכן אין מערכיים מטיפוס **void**, אין שדות מטיפוס **void** במבנה וקן חסודות עוד תכונות רבות כדי שהוא יהיה טיפוס. זו רק מילה שומרה בשפה שנראית כמו טיפוס ומיצגת את הערך **Unit**.
- ציון "**void**" בתוך רשימת פרמטרים לפונקציה אומר שהפונקציה מפנה את הטיפוס **Unit** לערך מטיפוס החזרה שלה. זו הדרך לצין פונקציה חסרת פרמטרים בשפות פונקציונליות טהורות. בשפות אימפרטיביות, המשמעות היא שהפונקציה אינה מקבלת פרמטרים והיא מחזירה ערך, אך סביר להניח שהיא גם משנה את מצב התוכנית. כולל מדבר בפירושו הרבה לאל פרמטרים.

ציוו "..."(3 נקודות) אומר שהפונקציה מקבלת ארגומנט מטיפוס ANY או TOP. ליתר דיוק מספר כלשהו, כולל אפס, של ארגומנט מטיפוס זה.

שאלה שקשורה לנושא:

- מה ההבדל בין "... " ובין **void** בשפת C ?<sup>20</sup>

התיחס לשימוש שלהם בתחום רשימת הארגומנטים של פונקציה. נסה תשובך תוך שימוש במונחים שנלמדו בקורס. הסבר כיצד התשובה משתנה כשועברים לשפת C++.

פתרון:  
"ציון **void** בתחום רשימת פרמטרים אומר שהפונקציה מפנה את הטיפוס UNIT לערך מטיפוס החזרה שלה. זו הדרך לצין פונקציה חסרת פרמטרים בשפות פונקציונליות טהורות. בשפות אימפרטיביות, המשמעות היא שהפונקציה אינה מקבלת פרמטרים, והיא מחזירה ערך, אך סביר גם להניח שהיא משנה את מצב התוכנית, כולל מדבר בפירושו לא פרמטרים. ציון ... כולל שלוש נקודות אומר שהפונקציה מקבלת ארגומנט מטיפוס ANY או TOP. ליתר דיוק, מספר כלשהו, כולל אפס של ארגומנטים מסווג זה."

- הוא טיפוס שלא מספק מידע על הערכים שלו. לפעמים ניתן להשלים ערך כזה.
- **void** למשל הוא ערך לא מושלם ולא ניתן להשלים אותו.
- טיפוסים לא מושלמים לא מספקים מידע על הערכים שלהם, לדוגמה: **struct Person\* person**.

### 4. בני טיפוסים נוספים הוא האיחוד הזר (Choice Type|Disjoint Union)

שלשות הפעולות הבסיסיות הנוגעות לאיחוד הזר הן:

- ייצור של ערך מטיפוס איחוד זר של שני טיפוסים (Construction).
- בהינתן ערך מקובצת האיחוד, לקבוע לאיזו מן הקבוצות שאוחדו הוא שייך (Tag Testing).
- קבלת הערך הקודם טרם הצמדת התוויות (Projection).

<sup>20</sup> נלקח מהאתר Safot.net – קישורים.

## The 3 basic operations on disjoint union

**Construction** Create a value of  $T + S$  from

- value  $v$ ,  $v \in T$  or  $v \in S$
- an appropriate tag.

**Tag Testing** Given  $v \in T + S$ , determine whether it came from  $T$  or  $S$ , i.e., determine whether  $v$  is of the form

$\langle \ell_1, v_1 \rangle$

or

$\langle \ell_2, v_2 \rangle$

**Projection** Extract the value, by removing the tag, e.g., obtain  $v_2$  from

$\langle \ell_2, v_2 \rangle$



דוגמאות לבנאים של איחוד זר: variant record in Pascal, datatype in ML, Union in C

### Definition (Choice Type)

If  $T$  and  $S$  are types, then so is their **disjoint union**,  $T + S$ , defined by

$$T + S = \{ \langle \text{left}, x \rangle \mid x \in T \} \cup \{ \langle \text{right}, x \rangle \mid x \in S \}.$$

where **left** and **right** are tags used for identifying whether the value came from  $T$  or  $S$ .

אפשר להסתכל גם על פוינטרים כעשוים מטיפול זה:

Pointers can be thought of as a choice between **Unit** type and  $T$

### Operations on Pointers

#### Construction

- Create a **null** pointer
- Create a pointer to a "stored" value

**Tag Testing** Determine whether a pointer is **null** or not.

**Projection** If the pointer is not **null**, extract value.

זהו מנגנון לאחסון הבחירה שנעשתה באיחוד זר (Choice Type) לצד הערך שנבחר כחלק מהבחירה.

סוגי ה- Tags בשפות השונות:

:ML -

בנוי בתוך השפה ונעשה בצורה implicit.tag בצורה ישירה.

:C -

באחריות המשמש לטפל ולהגדיר.

o

על מנת להוציא tag ל- union علينا להוסיף את זה ב- struct. לדוגמה:

```

struct {
    enum { LONGINT, DOUBLE, UNSIGNEDCHAR } tag;
    union {
        long int i;
        double d;
        unsigned char chars[sizeof(double)];
    } u;
} tagged_u;

```

שאלה בהקשר זה:

### - מה השווה והשונה בין Union לבין Disjoint union בשפת C ?

תשובה:

Disjoint union הינו אופרטור מתורת הקבוצות, בהינתן שתי קבוצות T,S, ה-Disjoint union שלן מוגדר להיות:

$$T + S = \{ (Left, x) | x \in T \} \cup \{ (Right, y) | y \in S \}$$

כאשר Left, Right הם תתיות כדי לוזה אם הערך הגיא -T או -S.

כל איבר בקבוצת S או איבר מ T שמייל גם תיוג שמצוין מאיזה קבוצה הוא הגיא. כוכור, טיפוסים מוגדר ע"י קבוצת ערכיהם, וכך בנאי טיפוסים מוגדרים ע"י אופרטורים על קבוצות.

union בשפת C הינו ניסיון למשוך disjoint union באמצעות union בשפת C מכיל בכל רגע נתון ערך אחד השיקד לאחד

הטיפוסים שמהם הוא נובע.

אולם, ההבדל העיקרי ביניהם הוא שבשפה C אין תיוג, ולכן אין באמצעות דרך לדעת לאיזה טיפוס שייך הערך הנוכחי. מכאן, ש - union בשפה C לא עונה באמת על הדרישות של disjoint union ולכן ניתן לראותו כת- "מוחלש".

לשם השוואה, בשפת ML ניתן להגיד disjoint union אמייתי (datatype) (datatype disjoint union) (datatype disjoint union), כאשר התיוג הוא חלק ממבנה מאופן השימוש, ולכן ניתן לדעת בכל רגע באיזה ערך מדובר ולא ניתן להתייחס לערך שלא לפני התיוג הנוכחי.

### : Type Errors (שגיאות טיפוס)

התכנית מבצעת **type error** (שגיאת טיפוס) על ערך  $T \in \tau$  אם היא מנשה לביצוע:

- שינוי של הערך בזורה שלא עקבית עם הטיפוס T.
- פירוש של הייצוג של המconaה של הערך  $\tau$  בזורה שלא עקבית עם הטיפוס T.

טיפולים Type Errors הוא לא בטוח פעמים רבות ויש כמה מקרים שעבורם נקלט Type Errors בעקבות ביצוע פעולות לא חוקיות הנוגעות לטיפוס מסווג זה.

### : Type Errors (שגיאות טיפוס) על דיווחים:

- דינמית – כאשר התכנית מבצעת את שגיאת הטיפוס עצמה.
- סטטית – כאשר התכנית לא יכולה להבטיח שלא יבוצעו שגיאות טיפוס בזמן הריצה.

מנגנון tag שנובע מקיום טיפוסי Choice-Type קיים בשפות שונות ונאכף בנסיבות שונות:

ב-ML יש אכיפה חזקה, ואילו ב-C ניתן למתקנת חופש רב. כתוצאה לכך פועלות union ב- C מאוד לא בטוחה.

ב-ML נוכל לדמות enum C-choice type של Units:

```

datatype suit =
  diamond of unit
| heart of unit
| spade of unit
| club of unit;

```

```

datatype suit =
  diamond
| heart
| spade
| club;

```

שאלה שקשורה לנושא:

- נתה את בניי האיחוד הזר (DISJOINT UNION) בכל אחת מהשפות הבאות, תוך התייחסות לקריטריונים הבאים:<sup>22</sup>

- האם קיים איחוד זר בשפה?
- האם קיימת תגית בשפה?
- מה מידת האכיפה של השימוש בתגית?
- האם איחוד זר בשפה מאפשר TYPE PUNNING?

א. פסקל

ב. C

ג. ML

ד. BASH

מקור: מועד א' אביב תשע"ה

פתרון:

- "פסקל": קיים בניי איחוד זר והוא ממומש ע"י variant record. קיימת תגית בשפה כאשר בגרסת המקורית השימוש היה חובה ובגרסת הסטנדרטية השימוש הוא אופציוני. השפה לא אוכפת שימוש נכוון בתגית בכל הגרסאות. הייצוג של איחוד זר הוא לא ידוע ולכן השפה לא מאפשרת type punning.
  - "שפה C": לא קיים איחוד זר אבל מספר עריכים על טיפוסים שונים באותו רצף ביטים לענן union. הגדרת התוиг הוא אופציוני ולכן השפה לא אוכפת תיויג.
  - "ML": קיים איחוד זר שנitin להגדיר אותו באמצעות datatype. הוא מובנה בשפה ואין אפשרות לגשת אליו שירות. קיימת אכיפה של שימוש נכוון בתויג. השפה לא מאפשרת type punning כי לא ניתן לשנות או להסיק כי ערך הוא מטיפוס אחר.
  - "Bash": אין כלל איחוד זר.
- תוספת לתשובה מהאחד המבחןים:
- ML תוצאה שגיאת ריצה על שימוש שגוי בתגית.
  - פסקל בגרסת הסטנדרטית מאפשרת (אך לא מחייבת) בדיקה בזמן ריצה.
  - C לא מספקת מנגנון לבדיקה בזמן ריצה – המתכונת צריך להוסיף שדה עבור-tag באופן יני.

#### נעמוד על ההגדרות השונות:

- Type aliasing (משתמש בשקלות מבנית<sup>23</sup>) – מtran שם נוסף לטייפוס קיימים. לאחר מתן שם נוסף, ניתן להשתמש בשני הפעולות, במקרה היו אותו הטיפוס בדיק.

- דוגמא ל- Type aliasing הוא המילה typedef בשפה C.
- Type branding (משתמש בשקלות שמית<sup>24</sup>) – יצירת טיפוס חדש על סמך טיפוסים אחרים קיימים. Type Branding משתמש בשקלות שמית ולא בשקלות מבנית ולכן גם אם ניצור שני טיפוסים זהים מבחינה מבנית, הם לא ייחשבו זהים מבחינת השפה. ניתן לחוש על טיפוס מסווג זה כאיחוד זר של טיפוס אחד.
- דוגמא ל- Type Branding הוא המילה struct ב-C.
- בשפות מונחות עצמים ניתן לבצע זאת בעוזרת עצמים.

דוגמה לתוכנית שבה ההבדל הזה חשוב:  
דוגמה TYPE במשפט פסקל ממציעים branding לטיפוס. לכן, לא ניתן לעשות:

<sup>22</sup> נלקח מהאתר Safot.net – קישוט.

<sup>23</sup> שקלות מבנית – אם המבנה הפנימי של שני טיפוסים הוא זהה – קלומר אם התחלנו מהטיפוסים האוטומטיים ובנינו שני טיפוסים בדיק באותו האופן – אז גם הטיפוסים נחשבים שקולים.

<sup>24</sup> שקלות שמית – שני טיפוסים שקולים רק אם יש להם את "אותו השם" והם הוגדרו באותו מקום.  
(שקלות שמית ומبنית מוגדרים באופן מפורט יותר בהמשך)

```

TYPE
    Meters      = Real;
    Seconds     = Real;
VAR
    m:Meters;
    s :Seconds;
    Procedure P_S(s:Seconds)
    |   Begin Write(s,"sec");
    end;
    Begin
        P_S(s);
        P_S(m); //ERROR
    end.

```

אם לא היינו יודעים ש- TYPE בשפת פסקל מבצעים לטיפוס, היינו חושבים שהתכוונית הייתה עובדת, משום ש- Real – מטרים (Meters) ו- שניות (Seconds).

כאמור, משום ש- TYPE בשפת פסקל מבצעים לטיפוס, לא ניתן לבצע זאת.

נשים לב כי ב- C הוא אין Alias, Brandingtypedef

### Some typedefs

```
typedef double meters;
```

### More typedefs

```
typedef double seconds;
typedef double coulombs;
```

```

void print_seconds(seconds s) { // Trying the above typedefs
    printf("%gsecs",s);
}
meters m; kgs k; seconds s; coulombs c;
main() {
    print_seconds(s); //✓
    print_seconds(m); //✓
    print_seconds(k); //✓
    print_seconds(c); //✓
}

```

### פירות על שקולות (טיפוסיות) שמיות ושקילות (טיפוסיות) אבנית:

**שקילות שמיות – Nominal Typing, Name Equivalence** – שני טיפוסים שקולים רק אם יש להם את "אותו השם" והם הוגדרו ב"אותו מקום". באופן יותר מדויק – שקולות שמיות בין שתי ישויות מתקיימת אם הם הוגדרו ע"י אותו טיפוס ממש. דוגמאות:

בפסקל יש שקולות שמיות אדוקה ולכנן עברו:

```

procedure sort(var a: array [1..50] of integer)
begin ... end;

```

- הערה: var מציין שהשגרה אינה פועלת על העתק של הפרמטר כי אם על הפרמטר עצמו. לכן, שינויים של הפרוצדורה בפרמטר ישתקפו מידית במשתנה שהועבר לה כפרמטר.

לא נוכל לשולח לפrozדורה זו שם דבר. מכיוון שלטיפוס זהה אין שם (וככל טיפוס שנשלחה אליו בוודאי לא הוגדר באותו מקום). ולכן גם אם נבנה טיפוס ממנה – קיבל אייר מתיפוס בעל שם אחר.

כדי לפתור זאת נגדיר טיפוס חדש, ניתן לו שם ונשתמש בשם זה בהגדרת השגרה:

```

type T = array [1..50] of integer;
procedure sort (var a: T);
...
var b : T;
sort(b);

```

- כוונתו של מתכון השפה הייתה טובה וחשובה – אילוץ המתכון מראש את טיפוסי הנתונים שעליהם התכוון שולו בעבוד, ולמנוע את המצב שבו שני טיפוסים שונים מטרות שונות בתכלית יתערבבו במקרה.

**שקליות מבנית – Structural Typing** – אם המבנה הפנימי של שני טיפוסים הוא זהה – קלומר אם התחלנו מהטיפוסים האוטומטיים ובנינו שני טיפוסים בדיקו אותו האופן – אז גם הטיפוסים נחשבו שקולים.

קשה להגדיר ולבדק עבור טיפוסים רקורסיביים (רשומות נניה).

שאלות הקשורות לשקליות מבנית ועל מתכון השפה לשים עליהם את הדעת:

○ האם רשומות דרישות שונות בין המשמות כדי שתהייה שקליות מבנית?

○ האם מערכים דרישים שוויין בין הטוחחים כדי להיות שקולים מבנית?

○ האם מונה "חלש יותר" מאשר "שקליות מבנית": Subtyping

○ פעולה שמצופה לקבל אופרנד מטיפוס  $T_{tag}$  ומקלט אופרנד מטיפוס  $T$  היא חוקית אם:

▪  $T$  ו-  $T_{tag}$  שקולים לגמרי.

▪ אם  $T_{tag}$  הוא subtype של  $T$ .

:height subtype age של subtype age – לדוגמה בפסוק `:height`

**type**

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

#### נבחין בין השקליות השונות:

- בפסוק יש שקליות שמיית הדזקה.

- ב-C לעומת זאת, יש גישה מעורבת:

○ שקליות שמיות ב: enum, struct, union

○ שקליות מבנית – עבור כל שאר הבנים, נהוג טיפוסיות מבנית ופרט ב:

▪ .typedef

▪ \* מצביע.

▪ & רפנס.

▪ – בנאי ההופך את הארגומנט שלו לטיפוס שאינו ניתן לשינוי אחריו שאתחול.

▪ volatile – בנאי ההופך את הארגומנט שלו לטיפוס "נדיף" ככל טיפוס שההדר אינו יכול להוניה

לגביו שימושים מסווגו משניים את רכם רק באמצעות שינויים שהתכנית מבצעת.

▪ (\*) מצביע לפונקציה.

▪ [] מערך.

○ דוגמאות שייעבדו בעקבות השקליות המבנית:

```
const volatile int & f(char *a, double b[]) {
    return * new int;
}

const volatile int & (*g)(char *a, double b[]) = f;
const volatile int & (*h)(char *a, double b[]) = g;
```

```
typedef
    const volatile int & (*T)(char *a, double b[]);

const volatile int & f(char *a, double b[]) {
    return * new int;
}

T g = f;

const volatile int & (*h)(char *a, double b[]) = g;
```

```

int f(double p[10000]) {
    return printf("p[5]=%g\n", p[5]);
}

static double a[] = {4, 5, 6};
static double b[] = {7, 8, 9};

int main() {
    return f(a);
}

```

- מזה שהדוגמה האחורונה תעבור – נוכל לומר כי שפת C היא **weakly typed**, מכיוון שבזמן ריצה אין בדיקה של חירגה מגבלות של מערך.
- בנוסף, ב- C אין הבדל בין מצביע ושם של מערך.
- בשפת C++ המצביע דומה, אלא שם הבנאי **class** יוצר טיפוסות שמות.
- בשפת Java יש שני בנאי טיפוסים בלבד: **מערך** (array) ו**מחלקה** (class). באופן דומה לשפת C, בשפת Java **מתקיימת טיפוסות שמות** לגביהם **מחלקות וטיפוסות מבנית** לגבי מערכיהם, וגם בשפה זו גודל המערך אינו חלק מהטיפוס. אולם, בשונה משפת C, ב- Java מטבחה בדיקה דינמית (בזמן ריצה) של חירגה מגדלי מערכים, ובהתאם לכך, ערך מטיפוס מערך בשפה זו נושא עמו בזמן ריצה את גודלו.

#### איזו שקולות עדיפה ולמה:

- ב-D"כ שקולות שמות עדיפה משיקולים של הנדסת תוכנה.
- שקולות שמות מתגנשת עם שני דברים:
  - קומפילציה נפרדת – נסביר ע"י דוגמא – נניה ויש שני קבועים שרצוים להשתמש באותו הטיפוס T. לכן, חייבים להגדיר אותו בשני הקבצים. הקומפילר של C עצם עיניים ומניח שההגדירות של הטיפוסים samaota זהות.
  - לא משנה היכן נשים את ההגדירה (גם אם היא תהיה בקובץ Header ונעשה לו משני הקבצים) – הקומפילר יכול לגשת לgresאות שונות של אותו Header או שפות ניתן לקמפל קובץ אחד, לשנות את ה-Header ואז לקמפל את הקובץ השני.

#### הסבירים נוספים<sup>25</sup>

##### שפת C פותרת את הבעיה בצורה הבא:

- אין צורך להשתמש ב"קדם מעבד" – ניתן להגדיר את "אותו הטיפוס" ב-2 הקבצים והשווין יהיה שווון מבני ולאשמי. שיטה זו עובדת בגלל העבודה שב-C weak typing יש.
- בדיקה ממשית של טיפוס ארגומנט לפונקציה.
- למעשה גם אם שמות הטיפוסים של הארגומנטים היו שונים וגם אם המבנה היה שונה לגמרי – עדין אפשר היה להדר שני קבועים המשתמשים בהגדירות שונות ולהריין את הקובץ.
- יותר מזה – הלינקר אינו בודק כי הטיפוס של הגדרת הפונקציה בקובץ אחד הוא הטיפוס אשר אליו מתיחס הקובץ الآخر.
- בנוסף, אין בדיקה של הטיפוסים של רשותות בין הלקוי תכניות שונות בשפת C.
- המסקנה היא ששוויון טיפוס רשותות (טיפוסות שמות) בין קבועים של שפת C אינו נבדק בזמן הידור.
- באופן דומה, שווין טיפוס פונקציות (טיפוסות מבנית) בין קבועים אף היא לא נבדקת.
- הוקי ששוויון הטיפוס לא נאכפים, ושגיאות טיפוס בין קבועים שונים יכולות לקרות. שגיאות יכולות לגרום לבעיות זמני ריצה מוזרות, שתగורמנה לעיצירת התוכנית בזמן ביצוע, ללא דיוקה על שגיאות טיפוס – או יותר גרוע, לפלט לא נכון.
- הבעיה המהותית היא העבודה שהלינקר אינו מכיר את הטיפוסים של השפה העילית.
- שמירה וקריאה של נתונים מקובץ – נתונים נוצרים ע"י מישחו אחר ולכן אין דרך לוודא שהטיפוס של הנתונים הם אותם טיפוסים, גם אם זהים לחולטין וגם אם נציגם לכל נתון את שם הטיפוס שלו. עדין לא תהיה שקולות

<sup>25</sup> More explanations in file: "06-StructuralandNominalTyping.pdf" pages 13-14

שנית. יתרה מזו, גם אם התוכנה שمرة את הנתונים שלה, היא לא יכולה לקרוא אותם ללא שבירת של שיקולות שמית, כי אין הבטהה שבאמת אלו הנתונים שהיא כתבה.

▪ דוגמא טובה לכך היא שתי תכניות פסקל שאחת כתבת נתונים והשנייה קוראת נתונים של טיפוס מסוים.

מכיוון שני הטיפוסים מוגדרים במקום אחר – הטיפוסים לא יהיו זהים. לכן הדרך היחידה להעbara המידע הוא באמצעות קבועים, אך לא ניתן לכתוב לקובץ שהטיפוס שלו הוא `File Of Integer`, מכיוון שהוא עדין טיפוס שונה בתכניות. לכן, הוגדו בפסקל טיפוסים מוגדרים מראש – `text` המציין טיפוס של קובץ של נתונים והטיפוס הזה מוכר בשתי התכניות (למעשה הטיפוסים `File Of Integer` והטיפוס `text` הם שונים בלבד). לדוגמה:

<pre>program p1(f) type   T = file of Integer; var   f:T; begin   ...   writeln(f,...);   ... end;</pre>	<pre>program p2(f) type   T = file of Integer; var   f:T; begin   ...   read(f,...); (* Type Error *)   ... end;</pre>
--	--

החיסרון בשיטה זו הוא המאמץ הרב בכתיבה שגורות הקידוד והפענוח, והנטל לשנותן בכל פעם שישנו שינוי בטיפוס.

▪ לפיה ההגדרה שתி תכניות פסקל לא יכולות לתקשר דרך קבצים בעזרת טיפוסים מוגדרים מראש (בשביל זה יש את `(text)` אבל בפועל, רוב המימושים של פסקל לא בודקים את הטיפוסים של קבצים – מה שהוחרת תחת "בティוחות הטיפוסים" בפסקל).

▪ בשפות בהן יש טיפוסיות מבנית – ההתאמה בין טיפוס הנתונים בשתי התכניות היא התאמה מבנית. במקרה זה, שימרת הנתונים תכיל גם את המבנה שלהם ולא ניתן יהיה לפתח את הקובץ אם הנתונים בו יהיו במבנה שונה מהמבנה לו מצפה התכנית הקוראת.

▪ הקשיי הזה שיש בטיפוסות שמיות, הוא לבדוק הסיבה שהשפות העוסקות בסיסית נתונים, כגון SQL יש טיפוסיות מבנית.

הסבירים נוספים על בעיות בשיקולות שמיות – **לא חובה לקרוא** – רק אם לא מובן מה שרשותי למלחה. ההסבר נלקח מ([קישוט](#)) ולא בהכרה הци מדויק, אך כן נבחר כתשובה נכונה ל-"מה הבעיות בשיקולות שמיות?".

**חיבור בין חלקו תוכנית כתובים באותו מקום**  
יש בעיה בחיבור בין חלקו תוכנית הכתובים באותו מקום, שכן כל אלמנט שניצור גם אם הוא שקול מבנית לאלמנט אחר, הוא בעצם אלמנט מטיפוס אחר. דוגמה שתבהיר את העניין: בשפת פסקל הטיפוסים של `a` ו `b` המצוינים להלן שונים:

`a: array of [1..4] of integer`  
`b: array of [1..4] of integer`

הם שקולים מבנית, אולם `a` הוא אלמנט של טיפוס ללא שם מסווג אחד, ו `b` הוא אלמנט של טיפוס ללא שם מסווג אחר.

אם אנחנו ריצים לגדיר טיפוסים `a,b` מטיפוס מערך וגם שהם יהיו מאותו טיפוס אנחנו צריכים לעשות את הדבר הבא:

```
type Num = array of [1..4] of integer
a: Num
b: Num
```

**קלט/פלט וקשר עם העולם החיצון**  
כפי שאמרנו, טיפוס מוגדר על פי השם שלו והמיקום שבו הוא הוגדר, לכן אם ננסה **לייצא** (פלט) או ביציקט מתוכנית `A` מטיפוס, `T`, ולאחר מכן **לייבא** (קלט) אותו לתוכנית `B`, או אפילו לאותה תוכנית `A`, או הדבר יהיה לא חוקי. הסיבה לכך היא שבטיפוסות שמיות כל היזור יוצר טיפוס חדש. כתעת הטיפוס `T` בתוכנית (`A` שהרצאה בפעם השנייה) כבר אינו זהה לטיפוס שהייתה בהרצאה הראשונה, ההיזור גורם לכך להיחשב כיצילו שהטיפוס לא הוגדר באותו מקום. מוזר, אבל הגיוני אם חשבים על זה. השפה לא יכולה לדעת הבדל בין הרצה של תוכנית `B` והרצאה של תוכנית `A`. היא פשוט בודקת אם האובייקט המוצג הוא מאותה ריצה של התוכנית, ואם לא אז הוא פשוט ייחס כיצילו הוא הגיע מתוכנית אחרת – ככלומר הוגדר במקום אחר.

### כתיבה תוכנית המתחלקת על כמה קבצים

שוב, טיפוס מוגדר על פי השם שלו ועל פי המיקום שבו הוא הוגדר, וכך גם אם תהיה אותו הגדלה של טיפוס בדיק בשני קבצים שונים, הוא פשוט ייחס כטיפוס שונה, בעקבות העובדה שהוא לא הוגדר באותו מקום. למשל מתכני פסקל פשוט התעלמו מה הצורך של מתכוון לכתוב תוכניות המתחלקות על כמה קבצים, והגידו אותה להיות שפה אוטרקטית. אם תרצו להשתמש בטיפוסים שיוכרים גם ע"י תוכנית A וגם ע"י תוכנית B תצטרכו:

לחבר את שני הקבצים לקובץ אחד (שזה כמובן פרטן לא מעשי, הרי כל המטרה של כתיבת תוכנית על כמה קבצים היא החלוקה הזאת).

בפסקל יש פרטן חלקי לבעה זו - קיים הטיפוס `text` (או הגדרת טיפוס מעט שונה - `File of Characters`), אשר יוכל לקרוא קובצי טקסט, ואו נעביר מידע בין תוכניות שונות על ידי קבצי טקסט, ופירוש חדש של הקבצים בתוך התוכנית. שוב שיטה מסורבלת אך זו הדרך החוקית היחידה בפסקל.

**טיפוס None** (ידוע גם ב-Bottom): הערך הריק של Choice Type Constructor. זהו בעצם קבוצה ריקה.

- **כפי ש-Unit** הוא הערך הריק של Cartesian-Product.
- **מיומש לדוגמא ב-C** (לא בהכרח הקומפיילרים יאשרו את זה):

#### 245. Emulating None in C

Challenge: define a type with no legal values

##### Option I: empty enum

```
typedef enum {
    // empty!
} none;
```



##### Option II: empty union

```
typedef union {
    // empty!
} none;
```



The author of these slides is not so sure your C compiler will like these!

**טיפוס Any** (ידוע גם C-Top): סט של כל הערכים בשפה. תמיינה קטנה של C++ Any בא להידי ביטוי עם "...". כמו למשל ב-`catch()`.

נקודות חשובות בנושא זה:  
- ב-Eiffel: כל הטיפוסים הם מחלקות, כולל יורשים מ-ANY. NONE יורש מכל המחלקות. אף מחלוקת לא יורשת מ-None.

##### ניתן להשוו עלי Unit, Any, None כ:

- Any – איחוד זר של כל הטיפוסים בשפה.
- None – טיפוס מורכב – איחוד זר של 0 אלמנטים.
- o הערך הריק של איחוד זר – Choice type.
- o טיפוס מורכב – התוצאה של 0 אלמנטים. Unit
- o הערך הריק של Cartesian product.

- בניית טיפוסים המעביר ערכיהם מקובצת S לקובצת T, ידוע גם בשם פונקציה.

## Definition (Mapping Type Constructor)

If  $T$  and  $S$  are types, the *mapping from  $S$  to  $T$* , denoted  $S \rightarrow T$  (sometimes also  $T^S$ ) is

$$S \rightarrow T = \{m \mid m \text{ is a mapping from } S \text{ to } T\}.$$

מס' הערכים שנקלבל בטיפוס המורכב הוא כמס' הערכים ב-S בגובה מס' הערכים ב-T. נשים לב שעבור None של Mapping כלשהי T קיבל Unit.

בנוסף, נבחין כי הקרדינליות של פונקציות Currying  $(s_2 \rightarrow T) : s_1 \rightarrow (T^{s_2})^{s_1}$  הוא:

- <sup>26</sup> דוגמאות למייפוי  $T \rightarrow S$ :
- פונקציה שמקבלת איבר מטיפוס S ומחזירה איבר מטיפוס T.
  - מערכיים אסוציאטיביים מ-S->T.

- בניי טיפוסים לייצרת תת קבוצה של קבוצת ערכים.  
אפשר להסתכל על בניי זה כפונקציה  $\text{Boolean} \rightarrow \text{Power Sets}$ .

בניי עבור טיפוסים אורדינליים הוא "...". המגדיר טיפוס שהוא תת חום של טיפוס אורדינלי כלשהו.

הבנייה נוספת היא הגדרה רקורסיבית של טיפוס ונסתפק בדוגמה קטנה עבורה.  
זהו הגדירה רקורסיבית של עץ:

```
datatype T = leaf of int | branch of int*T*T
```

**טיפוסים אטומיים (Atomic Type)** – טיפוס שלא ניתן לפרק את ערכיו לאבני יסוד קטנות יותר. בפרט, אין ערכים מטיפוס אחר המוכלים בתחום ערכי הטיפוס האטומי. טיפוסים שהוגדרו ע"י המתכנת (למשל Type בפסקל) או שהוגדרו ע"י השפה.

**טיפוס פרימיטיבי (Primitive Type)** – טיפוס אטומי שהוגדר מראש ע"י שפת התכנות. בד"כ שלו הוא מילה שמורה. לטיפוסים אלה יכולות להיות תכונות כגון: Ordinal, Ordered, Numeric.

טיפוסים פרימיטיביים שעבורם קיים יחס סדר. יכולים למפות ל- subrange טבעי (כמו int, bool).

**טיפוסים Non-Ordinal:** נקל צפה, String וכו'.

הערה בנושא זה – במספר מקומות, כמו באתר Safot.net ראייתי הגדרות שונות ל-"טיפוס פרימיטיבי". לעיתים טיפוס זה לא הוגדר כתיפוס אטומי ולעיתים הוא כן הוגדר כתיפוס אטומי. בסMASTER שלו – הוא הוגדר כתיפוס אטומי ואני משתמש בהגדרה זו. הערה נוספת זהה היא ש- "טיפוסים שהוגדרו ע"י השפה נקרא טיפוסים פרימיטיביים שכן הם לא תמיד אטומיים ("string") – זה לא בהכרח נכון ותלוי בהגדרה.

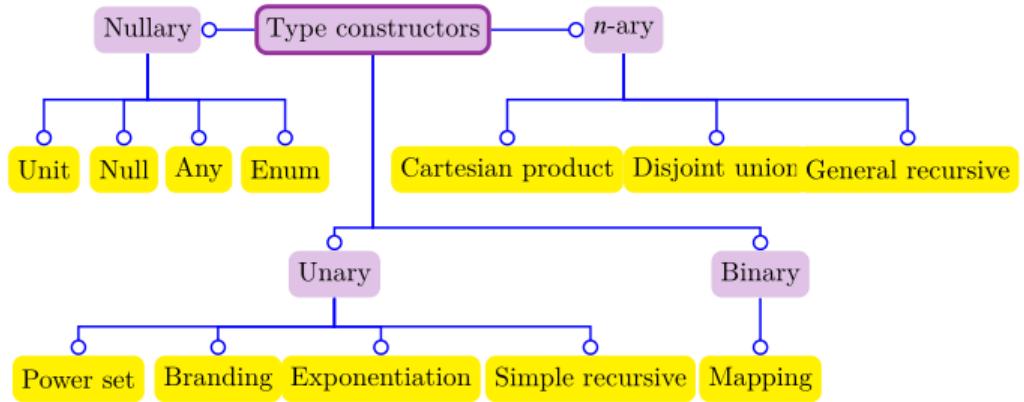
דוגמאות לטיפוסים:

**Int, char, double** – **טיפוסים אטומיים** וגם **טיפוסים פרימיטיביים** ב- Java, C, C++  
**Boolean** – הוא טיפוס פרימיטיבי ב- C++ ו- Java אבל אינו פרימיטיבי ב- C כיון שהוא טיפוס שנדרשת עבורו ספריה חיונית (stdbool.h), ועדיין בכל השפות הטיפוס הוא אטומי.

**String** – אינו פרימיטיבי ואני אטומי בכך אף אחד משלהש השפות הנ"ל. בעוד שבספת C "נתפר" פתרון השימוש במערך תווים (Char\*), ב- C++ וב- Java מיומש המחרוזות מובנה בספריות הסטנדרטיות בצורה מלאה.

## סיכום בנאים – Type Constructors

<sup>26</sup> נלקח מהפתרונות ל מבחן אביב 2013 מועד א'.



- הערכה בנושא:

- רשימת האופרטורים ליצירת טיפוסים חדשים בשפת C הם: struct, union, מצבי, מערך, פונקציה, כמו גם שתי המילים השמרות ו- .const volatile

#### הבדלים הסינטקטיים בין פונקציות לאופרטורים:

<u>נושאים</u>	<u>Operators</u>	<u>Functions</u>
מיקום הארגומנטים:	Prefix, Infix, Postfix	Prefix
חוקי קידימות:	קיימות.	אין.
חובת סוגרים:	לא חובה.	חובה.
איך מרכיב השם?	- סימני פיסוק: -, +, << וכוכ'.	מזהים. מזהים שמורים: new, sizeof - מזהים שמורים וגם סימני פיסוק: [][], delete [] וכוכ'.
מספר הארגומנטים:	1 עבר אוופרטורים שהם Prefix Postfix 2 עבר אוופרטורים שהם infix. 3 עבר " : ? " - ב- C	0,1,2,3,4
ניתנים להעמסה ע"י המתכנת?	ב C++, כנ.	ב- C++ ו- Java – כן.
	ב- C, Pascal – לא.	ב- Java, Pascal – לא.

שאלה:

<sup>27</sup> האם קיים ערך ששייך למספר סופי של טיפוסים כלומר יותר מטיפוס 1 אך לא מספר אינסופי?

תשובה:

ניקח את המשתנה v כך ש- v הוא טאפל: ()=v

עבור v מתקיים כי הטיפוסים היחידים המכילים אותו הם unit ו- any ולכן מתקיים עבורו כי: #types(v)=2

תשובה נוספת:

בפסקל הערך 3 שיך לטיפוס Int וכלל תת תחום שלו. יש די הרבה תת תחומיים כאלה, אבל בכלל זאת – מספרם סופי.

### 3. מערכות טיפוסים

נסוג מערכות טיפוסים בעזרת מס' קרייטריונים.

החסhti בושא זה את ה"קרייטריונים לאפיון PL".

קיום של **מערכת הטיפוסים**- האם קיימת בכלל מערכת טיפוסים בשפה? ב- Lisp למשל אין.

רמת תחכום- העשור של מערכת הטיפוסים בשפה, שמתבטה במספר הטיפוסים הפרימיטיביים, מספר בניין הטיפוסים ועוד. שפות סקריפט כמו Awk ו Bash מושך לרוב למתוחכם. מידת התחכום נמדדת בסולם:

- ① no typing
- ② degenerate typing
- ③ non-recursive type systems  
as in FORTRAN
- ④ recursive type systems  
as in PASCAL
- ⑤ functions as first class values  
as in ML
- ⑥ highly advanced type constructors  
e.g., "monads" of HASKELL

#### **:Degenerate Typing**

- מעט מאוד טיפוסים פרימיטיביים (ב"כ אחד או שניים).

- מעט מאוד בניין טיפוסים (Type Constructors). (

**אורתוגונליות** - במערכת טיפוסים המוגדרת רקורסיבית, כדי ליצור טיפוס חדש, יש לבחור בניין טיפוסים, ולבחור את הטיפוסים עליהם יופעל הבנייה. **המערכת היא אורתוגונלית אם שתי הבחירה יכולות להיות ב敞开 בלתי תלוי – كلומר ניתן להפעיל כל בניין על כל טיפוס.**

- במלילים אחרים, האם הטיפוסים שוים או שמא יש ביניהם **אפליה**, הבאה לידי ביטוי למשל במבנה טיפוסים אשר לא מקבלים את כל הטיפוסים.

○ נאמר ש-Type Constructor הוא מפללה אם הוא לא מופעל על כל type.

○ דוגמאות לאפליה:

- ב- C++ בניין הטיפוסים "רפרנס ל-" הוא מפללה כלפי עצמו שכן יש רפרנס כמעט לכל דבר אך אין רפרנס לרפרנס.

בשפת Pascal ניתן להעביר משתנים לפונקציה בעזרת רפרנס אבל לא ניתן להחזיר רפרנס מפונקציה.

ב- Pascal הטיפוס Integer הוא מופלה מכיוון שניתן ליצור set of Boolean \ set of char אבל אין אפשרות ליצור set of Integer. לכן, Integer הוא טיפוס מופלה. והמבנה טיפוסים set הוא טיפוס מפללה.

- דוגמא למערכת טיפוסים אורתוגונלית – ML, מכיוון שכל הבנאים יכולים לפעול על כל הטיפוסים.
- דוגמא למערכת טיפוסים שהיא רחוקה מלהיות אורתוגונלית – מערכת הטיפוסים של פסקל:

- פונקציות מופלות לרעה.
- אי אפשר ליצור קבוצות של כל טיפוס.
- אין אפשרות (בגלל שקיולות שמיות) ליצור קבצים של טיפוסים שאינם אוטומטיים.

#### **:Typed vs Untyped Languages**

**Typed** - ניתן לאוסף הערפדים לקבוצות עם פחות או יותר התחנוגות איחודת תחת אותן פעולות על ערכים בכל קבוצה.

לדוגמה: C, Pascal, ML, ADA, Java ורוב שפות התוכנות האחרות.

**Untyped** - לכל ערך יש אוסף פעולות שימושית לו והסמנטיקה שלהם ייחודית לערך.

לדוגמה: Lisp, Prolog, Mathematica וצדומה.

האם Lisp הוא Typed או Untyped?

Lisp היא Untyped מהסיבות הבאות:

- כל הערכדים הם S-Expressions.

- כל S-Expression הוא עץ ביןאיילם, שהעלים שלו הם אטומים.

- הפעולות הבסיסיות: Car, Cdr, Cons, Null.

**טיפוס מרמה ראשונה (First Class Type)** – בד"כ זה אומר שהטיפוס יכול<sup>28</sup>.

- להזור פונקציה.

- להיוות מעבר לפונקציה.

- להיבנות בזמן ריצה.

- ב- ML פונקציות הן First Class.

- פונקציות בשפת C הן לא First Class Type מכיוון שאין להיבנות בזמן ריצה.

**טיפוס מרמה שנייה (Second Class Type)** – הוא טיפוס אשר מופלה ע"י רובו או כל בניית הטיפוסים.

- בפסקל פונקציות הן second class type מכיוון שככל ה-type Constructors דוחים אותן.

### Case Study Pascal:

#### Orthogonality in PASCAL

##### Non-discriminatory type constructor

you can create arrays of anything,

##### Discriminatory type constructor

you can create sets of Booleans and of Char, but not of Integer or of "set of Boolean".

##### Second class type

there are no "arrays of functions", no "records of functions", no "sets of functions", no "pointers to functions", etc., so we must "conclude" that functions are second class types.

#### First & second class values in PASCAL

##### First-class values

Only simple, atomic values: truth values, characters, enumerands, integers, reals, and also pointers.

##### Lower-class values

can be passed as arguments, but cannot be stored, or returned, or used as components in other values

- composite values (records, arrays, sets and files): cannot be returned!
- procedure and function abstractions
- references to variables (unless disguised as pointers)

**ערך מרמה ראשונה (First Class Value)** – מוגדר כערך שעליו ניתן לבצע פעולות בסיסיות מסוימות כמו העברה כפרמטר לפונק' או החזרתו לפונק'.

בגدول – ערך שנייה לבצע עליו את כל מה שהשפה מאפשרת.

:First Class Value כל ערך שנייה לבצע עליו את הפעולות הבאות נחשב First Class Value.

- להעביר אותו כארוגומנטים לפונקצייה\פרוצדורה\פונקציה.

- להחזיר אותו מפונקציה\פרוצדורה\פונקציה.

- לבצע השמה לתוך משתנה.

- ליצור\לחשב אותו ע"י חישוב ביטוי.

בשפת פסקל רק הטיפוסים הבסיסיים, האטומים הם בעצם ערך מרמה ראשונה – ערכים אטומים, ערכי אמת, chars, enum, integers, reals ועוד.

שאר הערכדים המורכבים יותר כמו records ו-tuples הם ערך מרמה נמוכה יותר שכן לא ניתן למשתמש בהם להזירם כערך החזרה של פונק'.

**ערך מרמה שנייה (Second Class Value)** – לא מותר לעשות עליו את כל מה שהשפה מאפשרת. לדוגמה אי אפשר לתת לו שם,

אי אפשר לשים אותו במשתנה, אי אפשר להעביר אותו כפרמטר לפונקציה, אי אפשר להחזיר אותו כערך חזרה של פונקציה וכו'.

דוגמאות:

- בפסקל ניתן לתת שם לערך שהוא const a=3, אבל אי אפשר לתת שם לתוכן של מערך.

<sup>28</sup>נלקח מ- Stack Overflow – קישור.

- בפסקל ניתן להعبرיר ערך שהוא תוכן של מערך כפרמטר לפונקציה אבל לא ניתן להחזיר ערך זה.
  - ב- C אין אפשרות להعبرיר מערך כפרמטר לפונקציה, וגם אין אפשרות לכתוב פונקציה שמחזירה פונקציה או לשים מערך בשמשנה – לכן, **ערך ב- C הוא Second Class Value**.

שאלת שקופה לנושא:

הסביר את הביטוי: **טיפוסי Reference** דומים יותר ל-**First class citizens** מאשר ב-**C++** :

כתרון

בכל זאת, REFERENCE בשתת פונקציה מוגדר לדוגמה. אין אפשרות ליצור OBJECT REFERENCE, אבל, אין אפשרות שפונקציה תחזיר REFERENCE, או אפשר ליצור REFERENCE כshedah ברשותה, משתנה בפונקציה וקיים בזאת. כל אלו אפשרים בשפת C++.

**טיפוסיות חזקה (בטוחה) וחלשה** – רמת החזק של מערכת טיפוסים נמדדת ע"פ החופש הנitin למתכנת באשר לטיפוס של ערך. ככלומר האם מותר למתכנת להתעלם/לשנות טיפוס של ערך, האם פונק' תאפשר לקבל ערך שהיא לא הוגדרה עבورو ועוד.

**Type Punning** – טיפוסיות חלשה מתחייבת לפעמים ע"י שימוש ב-

<b>Weak Typing</b>	<b>Strong Typing</b>
<ul style="list-style-type: none"> <li>- עריכים משוכרים לטיפוסים אבל המתכונת יכול "לשבור" או להתעלם מהמשמעות זהה.</li> <li>- Type Punning</li> </ul>	<ul style="list-style-type: none"> <li>- בלתי אפשרי "לשבור" את הקישור בין הערך ובין הטיפוס שלו בשפה.</li> <li>- בלתי אפשרי להשתמש בפעולת עם ערך שלא מתאים לטיפוס של הפעולה.</li> </ul>
דוגמאות: assembly, C, C++, some variants of Pascal	דוגמאות: ML, Eiffel, Modula, Java

- Type Punning - הוא בעצם חשיפת המتنכנת למימוש של ערכיהם כרץ' של ביטים.
- "להתיחס לערך נוסף של ביטים ולהתיחס לביטים בלבד קשור ל'טיפוס'."

- ב-C בא לידי ביטוי ב-Casting של מצביעים ל-int וכן ב-union בו ידוע למתקנת כי כל ערך מהטיפוס יחזק אותו מס' של ביטים בזיכרון.

ל- Type Punning – ניסיון להשיג את הכתובת 1, למורota שזה אסור בנסיבות הטעינה כי המתכונת לא מתחס באופן שמייאי וישיר עם כבורותם בזיכרונו. אלא מערכם והפעלה היא זו שאהראים על זה – ולפניהם Type Punning<sup>30</sup>

- **ל- Type Punning יש את הכוח לבצע:**

```
long i, j;
int *p = &i, *q = &j;
long L1 = ((long)p);
long L2 = ((long)q);
long L = L1^L2;
```

8 : Union Type "לחתךל" בערך ע"י שימוש בו במקומות שלא מתחאים לערך שלו. לדוגמה בעזרת

```
union {
    double foo;
    long bar;
} baz;
baz.foo = 3.7;
printf("%ld\n", baz.bar);
```

**פסקל חזקה מ-C#, ג'אווה חזקה מ-**

**31 :Type Errors ו- Type Punning** הקשר בין Type Errors הוא סוג של Type Punning -

<sup>29</sup> נלקח מהאתר Safot.net – קישור.

<sup>30</sup> נלקח מהאתר Safot.net – קישור.

נלקח מהאתר – Safot.net – קישור.<sup>31</sup>

## בדיקות טיפוסים:

האם ערך שנשלח לפונק' הוא מהטיפוס שלו הוא מהכח  
הזמן שבו מתחבצאות בדיקות הטיפוסים מהו קритריון  
נוסף לאפין מערכת טיפוסים:

### Definition (Type Checking)

Language implementation applies **type checking** to ensure that no nonsensical operations occur.

#### Statically Typed PLs

- type rules are enforced at **compile time**.
- every variable and every formal parameter have an associated type.

C, PASCAL, EIFFEL, ML, ...

#### Dynamically Typed PLs

- type rules are enforced at **run-time**.
- variables, and more generally – expressions, have no associated type.
- only values have fixed types.

SMALLTALK, PROLOG, SNOBOL, APL,  
AWK, REXX, ...

**בטיפוסיות דינמית (Dynamic Typing )**, בזמן ריצה למשתנים וביטויים אין tag טיפוס עד שהם משוערכים לערך מטיפוס כלשהו. לכל ערך יש tag מזוהה שלו בזמן ריצה. חוקי טיפוסים נבדקים בזמן ריצה.

- רק לערכים יש fixed types.

- למשתנים וביטויים (variables and expressions) אין ערכים בזמן קומפליציה.

#### יתרונות:

- גמישות - מערכים יכולים להיות מכמה טיפוסים.
- אפשר להריץ תכנית חלקית.
- זמן פיתוח מהיר יותר.

#### חרובנות:

- Space overhead – כי לכל ערך יש אינפורמציה על הטיפוס בזמן הריצה.
- Time overhead – יש צורך לבדוק את ה-Tag בזמן ריצה.
- לא בטוח – הרבה שגיאות טיפוס שיכלו לעלות עליו בזמן קומפליציה.
- פחות בהירות נונגע לקוד. מהו הטיפוס? הרבה יותר קל להבין שיש מידע עבור הטיפוס שלהם.

**טיפוסיות סטטית (Static Typing )** מתחפינת בכך שברור איזה טיפוס משתיך לכל ישות בקוד, בד"כ תהיה הצהרה מפורשת על כך. לעומת זאת יופעל פונק' על ערכים שהן לא הותאמו לקלט, משתנים לא יקבלו ערכים שלא מטיפוסם ובנוסף כל טיפוס מגיע עם "חbillת מידע" על הטיפוס. חוקי טיפוסים נבדקים בזמן קומפליציה.

#### מאפיינים של טיפוסיות סטטית:

- לכל משתנה, פרמטר, פונקציה או פרוצדורה – רשם גם הטיפוס שלו.
- הצהרה לפני שימוש – כל המזהים צricsים להיות מוגדרים לפני השימוש.
- שום ערך לא יהיה כפוף לפעולות שהוא לא מכיר.
- <sup>32</sup> ערכים לא צricsים לשאת עימם Tag טיפוס – לכל ערך משוקט טיפוס ע"י המתכנת, וכך אין צורך שהערך ישא עימיו פיסת מידע המסמלת את הטיפוס שלו. מצב זה מביא לחסוך בזיכרון וליעילול תהליכי קריאת וכנתיבת ערכים לזכרון.
- משתנה יכול להכיל ורק ערכים שקשורים לטיפוס שלו.
- אין פעולה, גם ככל הימור גדרות על ידי המשתמש כמו פונקציות או פרוצדורות, שבה היא תפעל על ערך מטיפוס שהוא לא מzystפה לו.
- המגבילות של טיפוסיות סטטית באות ליידי ביטוי בעיות שלא ניתן לאיתור טرم בזמן הריצה כגון: חלוקה ב-0, גישה למצביע null, שורש של מספרים שליליים, בדיקה שאנו ניגשים בטוויה הנכונה של המערך וכו'.
- לעיתים יש חריגים – לדוגמה בפסקל הטיפוס text הוא חריג ולא מתקיים עבורו.type check

<sup>32</sup> נלקח מהאתר Safot.net – קיישוב.

- C היא בעל טיפוסיות שטחית. \*כל\* בדיקות הטיפוסים בשפת C נעשות בזמן הידור. אין כל בדיקה של טיפוסים בזמן ריצה.
- יתרונות של טיפוסיות שטחית חזקה:
  - בטיחות – פחות באגים.
  - יעילות – פחות בדיקות זמן ריצה, ושימוש יעיל יותר בזיכרון – במקרה לדינמי, בטיפוסיות זו אין צורך לבדוק בכל שלב מה הערך.
  - בהירות – הטיפוסיות הופכת את הקוד לברור יותר.
- חסרונות של טיפוסיות שטחית חזקה:
  - חוסר גמישות – לדוגמה בפסקל מספר האינדקסים של מערך חייב שנכתב את כל המספרים עד אותו גודל המערך.
  - בעיית כפל קוד – במקומות שנשתמש שוב באותו הקוד אנחנו צריכים ל כתוב מחדש – דבר שמקשה על שינויים ותחזקה – צריך לגשת לכל המקומיות ולשנות.

כל שפה נמצאת בספקטרום הבא שבין טיפוסיות "חזקת הלהקה" ובין "שטחית לדינמית":



תיאוריות ריס טוונת כי לא ניתן לקבוע אלגוריתם כללי שיחלית עבור תוכנית כלשהי שהוא יקבל כקלט, האם היא מחזיקה בתוכנה מסוימת כגון: האם היא עצור, האם היא נכונה ועוד. טיפוסיות סטטית וקיים של טיפוסים עוזרות להתחמק מהתיאוריה הזו – הסבר מפורט יותר [כאן](#).

- **Postmortem Typing** שם אחר ל-טיפוסיות חלשה, טיפוסיות המטילה על המשתמש את האחריות לאתרו ומניעת שגיאות טיפוסים. לדוגמה גליישה מגבלות מערך ב - C וועוד. כתוצאה מדיניות זו של הטיפוסיות, התוכנית לרוב תמצא את מורה לאחר זמן כלשהו (או שתתנתק באופן מוזר) ואז תבצע נתיחה שלאחר המות ובה יאותו שגיאות הטיפוסים (מכאן שמה של הטיפוסיות).

- **Mixed Typing** השילוב בין טיפוסיות סטטית ודינמית. שפת התכנות דוגלת **הן** בטיפוסית סטטית והן בטיפוסית דינמית. קלומר חלק בבדיקות הטיפוסים מבוצעות בזמן קומpileציה וחלק מבוצעות בזמן ריצה.

נוכל לראות דוגמא לכך בשפת ג'אווה:

## Mixed typing in JAVA

Error	Compile Time	Load Time	Runtime
$V = E$	✓	✓	
$f(E)$	✓	✓	
null pointer access			✓
array access			✓
uninitialized variable	✓	✓	

קלומר Java היא איפה שהוא בין Static Typing ל- Dynamic Typing מכיוון ש:   
- אלמנטים של Java – Dynamic Typing – דברים שנבדקים זמן ריצה:  

- גישה של מצביעים ל- Null.
- גישה מגבלות המערך – המערך בזמן ריצה נושא את גודלו וכך מתאפשרת הבדיקה.

- בדיקות תקינות Reference לעצמים.<sup>33</sup>
- אלמנטים של Java Static Typing – דברים שנבדקים בזמן קומpileציה\טעינה:
- בדיקת שגיאות של השמות למשתנים.
- העברת פרמטרים לפונקציה.
- שימוש במשתנה לא מואתחל.
- בדיקות של טיפוסים שלא אוחלו.<sup>34</sup>

דוגמא נוספת: עבור פסקל רוב הבדיקות מתבצעות בזמן קומpileציה אך חריגה מעורך נבדקת בזמן ריצה. יש שגיאות טיפוס שלא נבדקות כלל.

שאלה בהקשר זהה:

### **?Dynamic typing או Static typing** <sup>35</sup> האם ב JVM יש

תשובה:

אכיפה כללית הטיפוסים נעשית בעיקרה בזמן טעינה. ככלمر לא בכל פעם שתתבצע פקודה מסוימת תהיה בדיקה של הטיפוסים, אלא רק פעם אחת. אם הפקודה קוראת לפונקציה בקובץ אחר, תהיה טעינה של הקובץ الآخر, ובדיקה של הקראאה בפועל לעומת הגדירה בקובץ الآخر.. אם הקובץ כבר טוען, לא תהיה בדיקה נוספת. אם לעומת זאת, הפקודה אינה משתמשת במידע המצויה בקובץ אחר, אז הבדיקה שלא נעשית בזמן שטענו את הקובץ שמכיל את הפקודה.

הבדיקה זו נמצאת בעצם, בין static typing וdynamic typing.

**Gradual Typing** – תוספות מודרניות המאפשרות לתוכנת לכתוב תוכנית בטיפוסות דינמית אך עם התפתחות התוכנית וכיתהה להוסיף שיוך של טיפוסים למשתנים, ערכי החזרה ועוד. שפת התוכנה תשתח פעולה ותודיע על דברים שהם ברורים מאליהם ואין צורך בהם וכן על סטיירות שעשוות לגרום לשגיאות טיפוסים בזמן ריצה. וכך, באופן כללי, יורדו "הוואצאות" של זמן הריצה. בוגדול יותר <sup>36</sup> – שפת תכנות הדוגלת בטיפוסות דינמית (Dynamic typing), ככלמר הטיפוסים נאכפים בזמן ריצה, אך אפשררת, דרך Annotations, לבצע אכיפה סטטית בחלקם מסוימים מאוד בתוכנית. שפות כאלה הן למשל ActionScript או TypeScript אין שפות שלמדנו לכתוב בהן בקורס העשויות שימוש בטיפוסות זו).

חידוד קל – ההבדל בין Gradual Typing ל- Mixed Typing – מילה במילה מפרופ' גיל – מהקישור האחרון ():

- ב- Mixed Typing חלק מהבדיקות מתבצעות בזמן ריצה, והאחרות, בזמן הידור. יתכן שבבדיקות מסוימות תתבצענה גם וגם.
- ב- Gradual Typing, השפה תומכת בכך שבעור ישיות מסוימות הבדיקה היא בזמן ריצה, ואחרות יהיו בזמן הידור, וזאת כדי לעודד את המתוכנת לעבור מהאחד לשני.

**Duck Typing** – גרסה של טיפוסות דינמיות. ברגיל בודקים לפי הטיפוסים שאתה הפונק', יכולה לקבל כערך, ואילו ב-duck בודקים לפי הפונקציות שיכולה לפעול על הערך והאם הפונק' הנתונה היא ביניהן. מיטה יכולה כל ערך שהוא לו את ה-"טיפוס" משלו, שמגדיר קב' פונקציות כלשהי יכולה לפעול עליו. Duck

#### Dynamic Typing at run time

- ① Determine type  $T$  for which  $op$  is defined.
- ② Determine type  $T'$  the type of  $v$
- ③ If  $T' \leq T$ , execute  $op$  on  $v$

#### Duck Typing at run time

- ① Determine  $O(v)$ , the set of operations recognized by  $v$ , by either
  - determining  $T$ , the type of  $v$
  - reading the list  $O(v)$  as attached to  $v$
- ② If  $op \in O(v)$ , execute  $op$  on  $v$

<sup>33</sup> נלקח מהאתר Safot.net – [קישור](#).

<sup>34</sup> נלקח מהאתר Safot.net – [קישור](#).

<sup>35</sup> נלקח מהאתר Safot.net – [קישור](#).

<sup>36</sup> נלקח מהאתר Safot.net – [קישור](#).

- (Wikipedia) In computer programming with object-oriented programming languages, duck typing is a layer of programming language and design rules on top of typing. Typing is concerned with assigning a type to any object. Duck typing is concerned with establishing the suitability of an object for some purpose. With normal typing, suitability is assumed to be determined by an object's type only. In duck typing, an object's suitability is determined by the presence of certain methods and properties (with appropriate meaning), rather than the actual type of the object.

"... ב- **Duck Typing** התאמה של אובייקט נקבעת ע"י קיום מתחדות ותכונות מסוימות, במקומם לפי הטיפוס האמייתי של האובייקט".

- For example, in a non-duck-typed language, one would create a function that requires that the object passed into it be of type Duck, in order to ensure that that function can then use the object's walk and quack methods. In a duck-typed language, the function would take an object of any type and simply call its walk and quack methods, producing a run-time error if they are not defined. Instead of specifying types formally, duck typing practices rely on documentation, clear code, and testing to ensure correct use.

"... במקום לציין טיפוסים בצורה פורמלית (כמו בד"כ), ב- **Duck Typing** משתמשים על תיעוד, קוד קרייא ובדיוקן על מנת לוודא שימוש נכון".

ב- Duck – (סוג של dynamic typing) – נעשית בדיקה בזמן ריצעה בצורה קצת שונה. במקום לבוא לאופרטור ולבדק למה הוא מוגדר בשפה, אני אבודק את הטיפוס ואיזה פעולות מוגדרות עליו (נניח אם מנשים לעשות חיבור). כל פונקציה מגדרה סט של הgebenות לפי מה שקרה בתחום הפונקציה.

דוגמאות:

: Dogma 1 -

```
fun foo x {
    x.foo();
    x.foo2();
}
```

בזמן ריצעה והעברה של פונקציה ל- x.

אם יש בפונקציה משאו בסגנון: y+x.

או ב- statically typing רגיל כשהוא מגיע לשורה הזאת הוא יסתכל על האופרטור x ויבדק שהטיפוסים משמאלי ומימין מתאימים. ב- Duck Typing כשהוא מגיע לשורה הזאת הוא יסתכל על x ויבדק אם האופרטור + מוגדר עליו. זה ההבדל בגודול.

: Dogma 2 – ב- Scala -

```
class Dog { def speak() { println("woof") } }
class Klingon { def speak() { println("Qapla!") } }

object DuckTyping extends App {

  def callSpeak[A <: { def speak(): Unit }](obj: A) {
    obj.speak()
  }

  callSpeak(new Dog)
  callSpeak(new Klingon)

}
```

Running this code prints the following output:

```
woof
Qapla!
```

The class of the instance that's passed in doesn't matter at all. The only requirement for the parameter obj is that it's an instance of a class that has a speak() method.

## מַהוּ ?Duck Typing - Type

- Duck Typing – מאפשר שיהיה לכל ערך "טיפוס משל עצמו".
- טיפוס זמן הריצה – לכל פונקציה, לכל פרמטר ולכל קריאה מוגדרים סט של פעולות שאפשר לבצע בהאותה הפונקציה, לאותו הפרמטר ולאותה הקריאה הספציפית זו.
- קורה אם הטיפוס של הערך לא מתאים ל"טיפוס זמן הריצה" בזמן פעולה התכנית.

**Implicit|Explicit Typing** – האם הטיפוס של ישות נקבע אוטומטית ע"י המפרש (Implicit) או שמא המתכונת מכריז את טיפוס הישות בעצמו (Explicit), ככלומר על מי מותלת לאחריות. ב-**Implicit** נעשית הכרזת ערך ואת הטיפוס הקומפיילר יסיק לבדו.

הצורות השונות להצהרה על טיפוסים:

1. **Manifest typing**: באחריות המתכונת להציג על הטיפוס של המזהה.

שפות: פסקל, C.

2. **Implicit Typing ,Inferred typing**: רוב שפות התכונות יכולות להסיק, לפחות חלקית, את הטיפוס.

שפות: ML

הסכנות:

○ יצירה רשלנית של משתנים בעקבות טעויות הקלדה.

▪ הפטרון של ML – אין משתנים.

○ הודעות שגיאת טיפוס מבלבלות.

▪ הפטרון של ML – המתכונת יכול להוסיף הגדרה מפורשת של טיפוסים.

○ הסקות הטיפוס של חלק מהטיפוסים המורכבים (רקורסיביים וגורניים) יכולים להיות בעייתיים ולא ניתן היה להחליט ביניהם.

▪ הפטרון של ML:

- השפה מבצעת ניתוח עמוק של הטיפוסים בשפה כדי לגלוות מתי הבעייה הזאת יכולה להתתרחש.

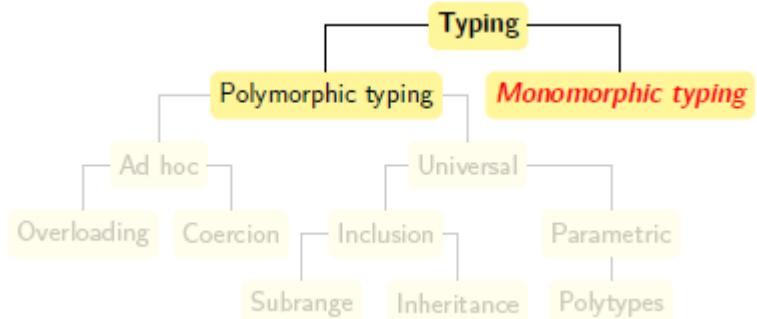
- ניתן להוסיף הגדרה מפורשת של טיפוסים.

## 3. :Semi-Implicit

- FORTRAN – משתנים שמתחלים באחד מששת האותיות הבאות הן :integers "i", "j", "k", "l", "m", ":".
- כל השאר הם ממשיים.
- – הגרסאות הישנות. סיומות כמו % ו-\$ קובעות את הטיפוס של המשתנה.

**טיפוסות שמייה ומبنית – Structural and Nominal typing** – הסביר לפני כן לעומק.

## טיפוסיות:



### מוניומורפיזם – Monomorphic Typing

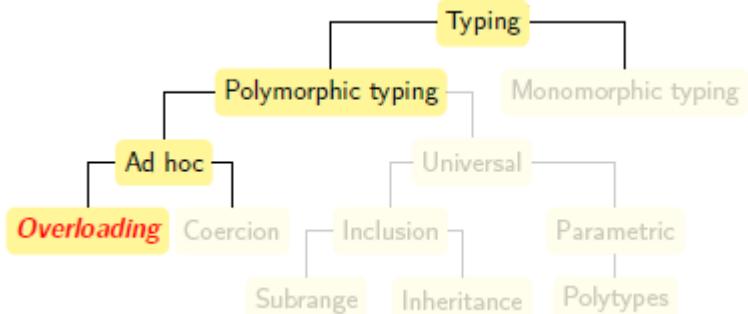
- מונומורפיות = בעלת צורה אחת.
  - לפונקציות ושאר הישויות בשפה יש רק טיפוס אחד וייחיד.
  - לדוגמה בפסקל – הפונקציות והפרוצדורות שמדובר בהן מונומורפיות.
  - לדוגמה בשפת C – הפונקציות שמדובר בהן מונומורפיות.
  - בדיקת שגיאות נעשית בצורה ישירה.
  - בעיתי כשרוצים להשתמש בקודים גנריים כדוגמת מין.
  - פולימורפיזם = בעל הרבה צורות.
  - לשויות יכולות להיות הרבה טיפוסים.
  - ניתן להשתמש בקוד שוב, בזכות פולימורפיזם אוניברסלי.
  - חומכת בפונקציות גנריות טיפוסים גנריים.
- מתחלק לשני סוגים: אד-הוק ואוניברסלי:

	Universal	Ad Hoc
No. Shapes	Unbounded	Finite and Few (often very few)
Shape Generation	Automatic	Manual
Shape Uniformity	Systematic	Coincidental

## :Polymorphic Typing – פירוט פולימורפיזם

- זהו פולימורפיזם** עבור מספר סופי של טיפוסים שנוצר בזיכרון ידנית ע"י \*המתכונת\* או \*מתקנן השפה\* שמספר הצורות בו הוא סופי ואין בהכרח מכנה משותף ביניהם (בין הצורות), מלבד הדמיון שהמתכונת יוצר עבורם. הערכה של פרופ' גיל – עדיף לא להתייחס ליצטר: "**זהו פולימורפיזם** עבור מספר סופי של טיפוסים שנוצר בזיכרון ידנית, **שמספר הצורות בו הוא סופי** ואין בהכרח מכנה משותף ביניהם".

## 1. העמסה ( Overloading )



מונח שיש לו כמה משמעותות, לאו דווקא קשורות אחת לשנייה. נוכל למצוא העמסה בשפות הדיבור שלנו, כאשר המשמעות של המונח מובנת מתוך הקשר.

### :Identifier\Operator" overloading

- לאופרטור "+" בפסקול וב-C יש שימושיות מוגנות:

- חיבור של מספרים שלמים.
  - חיבור של מספרים ממשיים.

- נאמר ר' פונזאייה מושגיהם אמרו: מזהים גם יכולם להיות מועמסים.

- לפוקציה יש יותר מטיפות אחד

- איז מאננו אוטומטי ליצור טיפוסים שונים לפונקציה

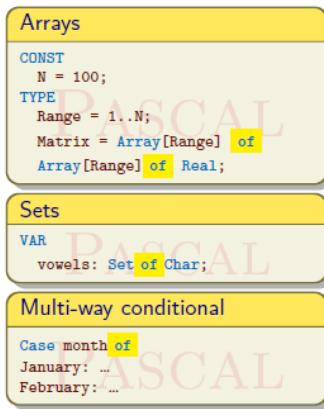
- לדוגמה מספר פרויזדורות או פונקציות שיש להן את אותו המזהה

- ובחיזי כי חלק מהפונקציות יכולות להיות מוטמאות בשפה וחלק בשליטת המשתמש

- כמו הפקודת Write בפקול - שモוצמת בינו לרשום.

```
WriteLn(0);
WriteLn(0.0);
WriteLn(false);
WriteLn(Sunday); // Sunday is defined:
.....           // "Days =(Sunday,Monday);"
```

- נבחן כי לא כל העמסה של מילוט מפתח (Keywords) היא העמסה פולימורפית!
    - "Plain" Overloading
    - "of" – יש לו ממשמעות אחרת בהקשרים שונים.



ו- **ישות**, כמו משתנה, פונקציה או פרוצדורה.

### - מילה מפתח המועמסה בשפה C – static

Meaning	Example	Comments
Scope	<code>static char buff [1000];</code>	When applied to definitions made at the outer most level of a file <sup>37</sup> Antonym of <code>extern</code> ; global in file, but inaccessible from other files
Storage class	<code>int counter(void) { static int val = 0; return val++; }</code>	Do not place on the stack. Shared by all invocations of the function. Antonym of <code>auto</code> ; value persists between different invocations.
Not an instance member	<code>class Book { static int n; public: Book() { ++n; } ~Book() { --n; } };</code>	Shared by all instances of a <code>struct</code> or a <code>class</code> .

Table 4.4.1: Overloading of keyword `static` in C

### - ההעמסה של מילוט מפתח היא לא הטעמה פולימורפית מכיוון שהיא לא מצינית משחו Nameable

- הערות נוספות בנוגע להעמסה:
- שפת C אוסרת על העמסת פונקציות בשפה, אך שפת C++ מאפשרת זאת.
- ב- C\C++ אין פונקי' מועמסות המבנה בשפה שכן אל לנו לשכוח שאין בהן כלל פונקי' המובנות בשפה (אלא פונק' ספרייה). לנוכח יש העמסה של פונקציות בשפה המשמש או בתוך הספריות עצמן בשפה.
- שפת C ושפת Pascal לא מאפשרת למשמש להגדיר העמסת אופרטורים בשפה, אך יש בשפה אופרטורים Built-in שםועסים.

שפות שונות בוחרות בפתרונות שונים עבור דו המשמעות של פונק' מועמסות:

(Context Dependent vs Context Independent)

## Ambiguity resolution

Consider the call `Id(E)` where `Id` denotes both:

- a function  $f_1$  of type  $S_1 \rightarrow T_1$
- a function  $f_2$  of type  $S_2 \rightarrow T_2$



```
//If written in C++:
    class T1;
    class T2;
    class S1;
    class S2;
//In general:      Id(E)
/*f1:*/          T1 Id(S1 element) { ... }
/*f2:*/          T2 Id(S2 element) { ... }
```

### Context Independent (C++)

- Either  $f_1$  or  $f_2$  is selected depending *solely* on the type of `E` (class  $S_1$  or class  $S_2$ )
- We must have  $S_1 \neq S_2$
- May lead to ambiguities in the presence of coercion

### Context Dependent (ADA)

- Either  $f_1$  or  $f_2$  is selected depending *on both* on the type of `E` or how `Id(E)` is used.
- Either  $S_1 \neq S_2$  or  $T_1 \neq T_2$  (or both).
- Ambiguity is not always resolved:

```
x : Float := (7/2)/(5/2);
```

Has at least two ambiguous interpretations:

- $3/2 = 1.5$ ,
- $3.5/2.5 = 1.4$ .

<sup>37</sup> כלומר:

- בחירת הפונקציה נעה **Context Independent** – שמיובלת בטיבוס שמקבלת הפקחיה המועמסת ולא תלוות בערך חזרה.  
- בחירת הפונקציה נעה **Context Dependent** כתלות ב:

- טיבוס שמקבלת הפקחיה המועמסת.
- או בטיפוס שמחזירה הפקחיה המועמסת.
- או גם וגם.

- **הסתירה (Hiding)** – היא מצב שבו הגדרה חדשה של מזהה בשפה מסתירה את ההגדרה הקודמת עבורה אותו מזהה ולכן יהיה שימוש בהגדירה החדשה. במקרה אחרות, הסתרה נוצרת ב- Lexical Scoping כאשר מזהה המוגדר בסביבה החיצונית, מוגדר מחדש בסביבה הפנימית לה. ההגדרה הפנימית יותר מסתירה את הגדרה החיצונית.

דוגמאות:

דוגמא להסתירה בשפת C++

דוגמא להסתירה בשפת C

```
namespace p1{
    void func(){
        cout << "p1 - func - void" << endl;
    }
    void func(int a) {
        cout << "p1 - func - int" << endl;
    }
    void func_not_overidden() {
        cout << "Not overridden" << endl;
    }
    void func_not_overidden(int a) {
        cout << "Not overloaded int" << endl;
    }
}
// second name space
namespace p2{
    void func(){
        cout << "p2 - func - void" << endl;
    }
    void s() {
        func(); // Will call p1::p2::func();
    }
}
```

**Hiding in C**

```
static long tail;
...
int main(int ac, char **av) {
    // hides outer tail
    const char **tail = av + ac - 1;
}
```

<sup>37</sup> נשאל באתר SAFOT.NET – [קישור](#). לטענת המרצה ההגדרה שלי מבלבלת. לא התקבלה תשובה על מה ההגדרה המדוייקת.

```

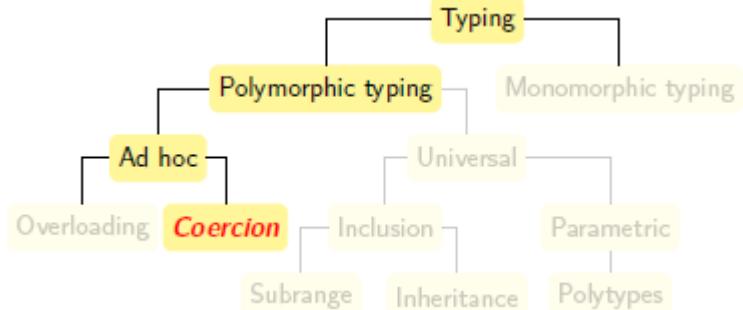
    //func(2); //Won't work since func(int) is hidden (because we override func(void))
    func_not_overidden(); //<-- However if the function is not overridden
    func_not_overidden(2); //<-- this will also work
}
}
}

```

חשיבות לשים לב להבדל בין Hiding & Overloading בשפה:

- שימושות חדשה מסתירה משמעות שונה.
- מספר שימושיות יכולות להיות ביחד.

הمرة לא מפורשת של ערכיהם מטיפוס אחד לערכים מטיפוס אחר. **Coercion .2**



ב C++ בזמנן ריצה כאשר נקראת פונק' אשר עברורה יש העמסה, יישנו "טורניר" בין כל הפונק' שעשויות להתחייב והמנצחת נבחרת אם בה יש כמה שפחות המרומות. אם אין מנצחת בודדת אז זו שגיאיה.

דוגמא:

- פסקל מאפשר המرة לא מפורשת מ- Integer ל- Real – פונקציות שמוגדרות לקבל Real אבל יכולות לקבל גם Integer, בזכות ההמרה.
- בשפת C++ :

## Builtin coercion in C++

```

int pi = 3.14159; // Builtin coercion from double to int
float x = '\0'; // Builtin coercion from char to float
extern double sqrt(float);
x = sqrt(pi); // Builtin coercion from int to double
               // and then
               // Builtin coercion from double to float

```

Coercion is sometimes called, especially in C++, type casting and type conversion, without particular distinction between implicit and explicit applications.

- בשפת C ניתן לבצע המرة לא מפורשת (ומפורשת) של int, double, float אחד לתוך השני.
- חיסרנו - חוסר בהירות בקשר לדרך הهماה של טיפוס אחד לティיפוס אחר.

## פולימורפיזם אוניברסלי ( Parametric & Inclusion ) :

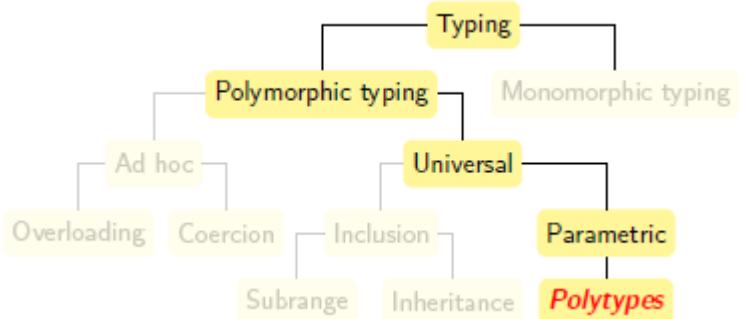
ניבור לפולימורפיזם אוניברסלי, המתחלק לפולימורפיזם פרמטרי והכללה.

- זה פולימורפיזם עבור מספר לא חסום של טיפוסים, שנוצר כתוצאה אוטומטית ויש מבנה משותף בין הצורות.
- פולימורפיזם הוא אוניברסלי אם אופרטור פועל בצורה זהה עבור כל הערכים שאוחתם הוא יכול לקבל.
- פולימורפיזם אוניברסלי כגון פונק' ומחלקות גנריים מעידים על היות מערכת הטיפוסים פולימורפית.
- יתרונות של פולימורפיזם אוניברסלי:
  - o לפונקציה יחידה (או טיפוס) יש משפחה גדולה של טיפוסים קשורים אליה.
  - o הפונקציה פועלת בצורה אחידה על הארגומנטים שלה, ללא קשר לטיפוס.
  - o אפשרותה כוח הבעה גדול יותר של השפה, מכיוון שפונקציה אחת יכולה לקחת ארגומנטים ממספר אינסופי של טיפוסים.

### 1. פולימורפיזם פרמטרי (Parametric)

הוא למשל פונק' פולימורפית שהן פונק' שעובדות על מגוון של טיפוסים.

דוגמא לפונקציות כאלה הן הפונקציות הגנריות ב- C++, אותן ניתן לצור עם Templates.



ב - C++ הפונק' הפולימורפית נבדקת רק בעת השימוש בה ואילו ב- ML היא נבדקת בזמן הגדرتה. ML מבעת הסקט טיפוסים מתחכמת יותר ויכולת להסיק טיפוס הפונק' לפי ערך ההחזרה. יש לציין כי העמסה מסבכת את תהליך הסקט הטיפוסים ולכן ML לא מאפשרת העמסה ע"י המתכנת, וכן כאשר היא עוסקת בהסקה של טיפוס היא מתעלמת מההעמסות שモוגדרות בשפה.

	ML	C++
<i>Declaration of Type Parameters</i>	Optional	Obligatory
<i>Passing Type Arguments</i>	Optional	No
<i>Checking</i>	On Declaration	On Instantiation

### - 38 Polytypes

טיפוסים שיש להם פרמטר טיפוס, כמו למשל טיפוסים גנריים.

זהו טיפוס "משותף" לכל המופעים ( Instances ) של טיפוס פרמטרי ספציפי.

טיפוס שבגדרה שלו יש אחד או יותר Type Variables – שזה בעצם משתנה שהערכים שלו הם טיפוסים (לדוגמאות ב- ml).

<sup>38</sup> More explanations on polytypes in "\_beamer-4-+-advanced-typing.pdf" pages 48 and above.

## No programmer-defined polytypes in PASCAL!

The type of the predefined function `eof` is `File of σ`. If PASCAL had user defined polytypes, we could have written

```
Pascal
TYPE
  Pair(σ) = Record
    first, second: σ;
  end;
  IntPair = Pair(Integer);

```

```
Pascal
TYPE
  RealPair = Pair(Real);
  list(σ) = ...;
  VAR
    line: list(Char);

```

Unfortunately, this would not work in PASCAL. All we can write is something of the sort of

```
Pascal
TYPE IntPair = Record
  first, second: Integer;
end;
VAR line: CharList;
```

נבחן כי Monotype הוא טיפוס שבהגדרה שלו אין אין Type Variables.

### מהם הערכיהם של Polytypes?

- ב- C++ לפונקציה אין ערכים, רק אם מחליפים את הטיפוס ב- `type variable` או נקבל טיפוס אמיתי.
- נבחן כי Templates in C++ הם חסרי ערכים! רק אם נקבע ערך ממש או נקבל ערך ממש (בניגוד ל- ML לדוגמה).
- ב- ML ניתן בקלות להגדיר ערכים בעלי ייצוג polytypes, המיצגים פונקציות פולימורפיות – לדוגמה:  $'a \ x \ 'a \rightarrow 'a$
- סט הערכיהם של polytypes הוא החיתוך בין כל הטיפוסים שיכולים להיות מ- `Polytype` זה. בМИלים אחרות – כל הערכיהם (יכולים להיות גם פונקציות) השיכים לו ללא קשר לטיפוס שימושיו לו לאחר מכן.

### דוגמאות בעבר Polytype

1. נתבונן ב- `List(γ)` :Polytype שנובעים מטיפוס זה: Monotypes
  - – כל הרשימות הסופיות של Integers כולל הרשימה הריקה.
  - – כל הרשימות הסופיות של ערכי אמת או שקר כולל הרשימה הריקה
  - ...
  - נבחן כי הרשימה הריקה משותפת לכל הקבוצות:
    - הרשימה הריקה היא ערך של כל Monotype שנבע מהתיפוס  $\text{List}(\gamma)$ .
    - התיפוס של הרשימה הריקה הוא:  $\text{List}(\gamma)$
    - אין עוד ערכים מהטיפוס:  $\text{List}(\gamma)$
2. נתבונן ב-  $\gamma \rightarrow \gamma$  :Polytype שנובעים מטיפוס זה: Monotypes
  - – Integer → Integer – פונקציה שבמיהה את המספר הבא ברשימה וכו'.
  - – String → String – פונקציה הזהות (מ- `String` ל- `String`), פונקציה שמוסיפה תוו למחרוזת, מוחקת רוחזים מהמחרוזת וכו'.
  - ...
  - נבחן כי פונקציית הזהות משותפת לכל הטיפוסים שעבורם:  $\gamma \rightarrow \gamma$ .
3. דוגמא של Polytype ריק – כלומר Polytype שהחיתוך של כל הטיפוסים שיכולים להיות מ- Polytype זה הוא ריק:
  - τ
  - τ x τ

שאלה בנושא:<sup>39</sup>

כמה ערכים, אם בכלל, יש לכל אחד מהטיפוסים הפולימורפיים שלහן?

הנicho שהטיפוסים  $\tau$ - $\theta$  מוגדרים עם שוויון:

$$\text{א. } (\tau \times \tau) \rightarrow \tau$$

$$\tau \rightarrow (\tau \times \tau)$$

$$\tau \rightarrow \text{bool}$$

$$(\tau \times \theta) \rightarrow \tau . \tau$$

$$\tau \rightarrow (\tau \times \theta)$$

$$(\tau \times \tau) \rightarrow \text{bool}$$

$$\tau \rightarrow (\tau \rightarrow \text{bool})$$

פתרונות:

א. 2 ערכיים: פונקציה המחזיר את האיבר הראשון, פונקציה המחזיר את האיבר השני.

ב. ערך אחד: פונקציה המחזיר Pair המורכב מהערך שהועבר

ג. 2 ערכיים: פונקציה המ>Returns True לכל ערך, פונקציה המ>Returns False למל' ערך

ד. ערך אחד: פונקציה המ>Returns את האיבר הראשון

ה. 0 ערכיים:  $\theta$  מועבר כารוגמנט אך לא בחרו איך להוציאו כפלט ערך שלו מבלי להכין מראש.

ו. 4 ערכיים: פונקציה המ>Returns True לכל ערך, פונקציה המ>Returns False לכל ערך אם הערכים שוים (תזכורת: נתון

שמוגדר על  $\tau$  פעולה השוואתית -  $\tau = \tau$  אחרת, והפונקציה ההפוכה לכך ( $\tau \neq \tau$  אם הם שוויים)

ז. 4 ערכיים: בדוגמה לסעיף ז'

#### שאלה נוספת:

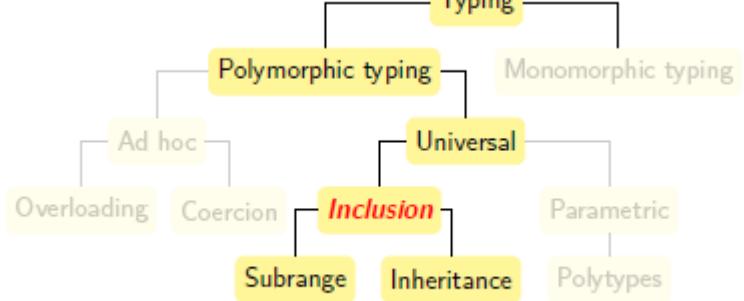
- האם הקידינליות של:  $\tau \rightarrow (\tau \rightarrow \tau)$  גדולה מ-1? התשובה היא כן. ניתן שתי דוגמאות:

```
fn (x::xs) = x;
fn (x::y::xs) = y;
```

#### 2. פולימורפיזם של הכללה (Inclusion)

נובע מקשר של הכללה בין שני טיפוסים או שני קבוצות ערכים.

בפסוק יש ביטוי לכך עם בנאי הטייפוסים של תחת קבוצה "...".



מחולק לשניים:

- נגידר תחילתה מהו **Subtyping**

- גראסא 1: טיפוס A הוא subtype של טיפוס B אם  $A \subseteq B$
- גראסא 2: טיפוס B הוא subtype של טיפוס A אם לכל ערך מטיפוס A ניתן לבצע המרה (מפורשת או לא) לערך מטיפוס B.
- דוגמאות:

- בפסקל הערך Nil – שijk לכל טיפוסי המצביעים.
- בשפת C הערך 0 הוא פולימורפי – שijk לכל טיפוסי המצביעים.
- בשפת C++ הטיפוס void\* הוא super-type של כל טיפוסי המצביעים.

#### נבחין כי: Inclusion מורכב מ:

לדוגמא בפסקל: Subrange

#### Type

Index = 1..100;

Digit = '0'..'9';

- בפסקל טיפוס subrange "ירוש" את כל הפעולות של הטיפוס אב שלו.

:Subtype Subclass ( Inheritance )

class manager: public Employee { ... } -

- המרה ב-C של long\* void היא לא Ad-Hoc אלא פולימורפיזם פרמטרי שכן היא אוניברסלית וקיימת עבור כל טיפוסי המצביעים.

### הבדל בין פולימורפיזם אוניברסלי ופולימורפיזם Ad-Hoc

<u>Ad-Hoc</u>	<u>Universal</u>	:polymorphism
מספר הצורות השונות הוא סופי, חסום, ובד"כ קטן.	מספר הצורות השונות לא חסום.	מספר הצורות השונות:
כל צורה וצורה מוגדרת בנפרד.	נעשית ע"י תבנית אחת. מתבנית זו נוצרות כל הצורות השונות באמצעות מכני.	הגדרת הצורות השונות:
כיון שב ad hoc polymorphism גדרת הצורות השונות נעשית על ידי תבנית אחת, הצורות השונות אחידות.	כיון שב universal polymorphism גדרת הצורות השונות נעשית על ידי תבנית אחת, הצורות השונות אחידות.	אחדות הצורות הנוצרות:
<b>polimorfism coercion</b> 3+7.5; // "3" is Coerced to "double" in order to do the addition  <b>polimorfism העמלה:</b> //The operator "<" in the next template: template <typename T> T max(T a, T b) { return a<b? b: a; } //Now the operator "<" uses different meanings according to the type it gets. //For example for "int" and "double" it will have different meanings	<b>polimorfism פרמטרי:</b> ב- C++ פונקציה טמפלטית: template <typename T> T max(T a, T b) { return a<b? b: a; }  <b>polimorfism הכללה:</b> C++-ירושה ב- class A{}; class B: public A{}; <b>subrange .2</b> TYPE Index = 1..100; Digit = '0'..'9';	דוגמאות:

### הערה לגבי אחדות הצורות הנוצרות בפולימורפיזם Ad-Hoc

כיוון שב ad hoc polymorphism כל צורה וצורה מוגדרת בנפרד, הדמיון בין הצורות השונות הוא לא כואורה יד המקירה בלבד. בפועל, המתכונת ו/או יוצר השפה, שבנה את הצורות השונות בדרך כלל משתדל שייהה דמיון בין הצורות השונות. כך למשל, ישנו פולימורפיים אד הוק של אופרטור החיבור בשפת C, והגדרת שימוש החיבור של מספרים שלמים שונה בתכלית מהחיבור של מספרים ממשיים, הרי ברור שמתכוון השפה רצה שהצורות השונות של אופרטור החיבור תהיה דומות במובן זה, שכן שתיהן מגישות את האידיאל של החיבור המתמטי, שהוא זהה למספרים שלמים ממשיים.

#### סיכום נוסף על סוגי הפולימורפיום השוניים:

## Varieties of polymorphism

### Ad Hoc **Created by hand; caters for a limited number of types**

**Overloading** A single identifier denotes several functions is an ad hoc term simultaneously

- Reuse is limited to names, but there are is reusable code

**Coercion** A single function can serve several types thanks to implicit coercion between types

- Extending the utility of a single function, using implicit conversions

### Universal **Systematic, applies to many types**

**Parametric** Functions that operate *uniformly* on values of different types

**Inclusion** Subtypes inherit functions from their supertypes

## 4. אחסון

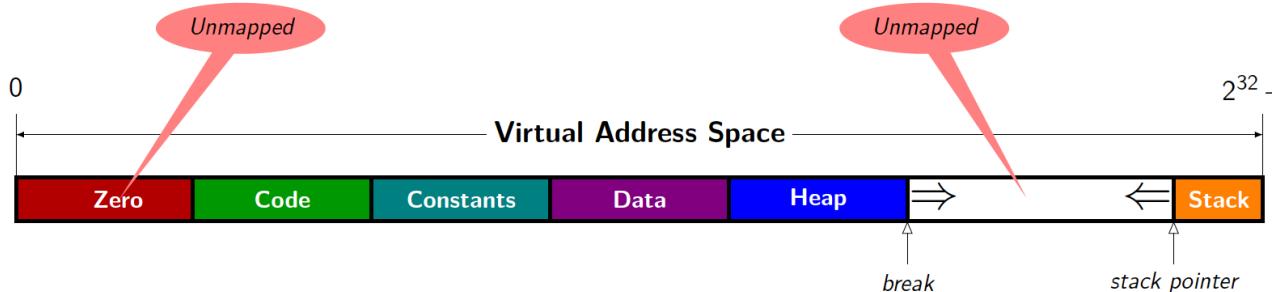
משתנה הוא ישות שעשוויה להכיל ערך ומספקת את האפשרות לבדוק מה הערך שמכילה, אם מכילה זהה, וכן האפשרות לעדכן ערך זה.

- "תא (Cell) מוקצה" הוא המימוש של "משתנה".
- בד"כ כשמיישו אומר "משתנה" הוא מתכוון "משתנה בעל שם".
- אך יש גם "משתנים ללא שם".
- בקורס נשתמש ב"משתנה" על מנת לתאר גם "משתנים בעלי שם" וגם "משתנים ללא שם".

האחסון הוא קבוצה של תאים שעשויים להיות בכל מיני מצבים שונים. עבור משתנים וערבים ניתן לרוב לבצע **בדיקה / עדכון סלקטיביים** (Selective update), אולם לעיתים ניתן לעשות רק **עדכון טוטאלי** (Total update). לדוגמה אם יש ערך מרכיב (struct) ומצביעים השמה לרכיב אחד שלו – זה Total update. לעומת זאת אם מעדכנים את כל הערך המורכב – זה Selective update.

מודל הזיכרון הקלاسي:

Segments:



Permissions:

	READ	WRITE	EXECUTE
Zero	X	X	X
Code	X	X	✓
Constants	✓	X	X
Data	✓	✓	X
Heap	✓	✓	X

:Selective Update

נניח שיש לנו תא בזיכרון (store) המכיל ערך מרכיב (כלומר לא ערך אטומי). תא בזיכרון זה הוא בדרך כלל המשתנה, אך לא בהכרח. Selective update הוא האפשרות לעדכן תת-ערך של הערך המורכב המוחסן בתא.

שאלה בהקשר:

- <sup>40</sup> הסבר מהו selective update ומדוע הוא מבקשת על מימוש שפות שבחן יש העברת פרמטרים **by value**.

פתרונות:

"נניח שיש לנו תא בזיכרון (store) המכיל ערך מרכיב (כלומר לא ערך אטומי). תא בזיכרון זה הוא בד"כ משתנה, אך לא בהכרח. Selective update הוא האפשרות לעדכן תת-ערך של הערך המורכב המוחסן בתא.

<sup>40</sup> מבחן חורף 2012 מועד א'.

ברור שערכים מורכבים גדולים נפה זכרון רב. בשפות שיש בהן העברת פרמטרים by value לכואורה היה צריך להעביר פרמטרים שמכילים ערכים מורכבים ע"י העתקה של הזכרון הזה. העתקה זו דורשת זמן רב, וגם מגדילה את משאבי הזכרון של התוכנית.

אבל, אם בשפה אין צורך לבצע selective update און קורס לבעצם את הפעתקה הzo כלל. במקרה זה, ניתן לבצע ביעילות ובפשטות ע"י העברת reference לערך שMOVPERBY value. הרותינה הנקראת, רשותה כMOVPERBY לעדכן את תוכן הפלט, מוביל לשינוי ישפיע על הערך של הפלט ברוטינה הקוראת. אבל, כמובן שהעדכון הוא טוטלי ולא חלקי, כל שיש צורך לעשות הוא לעדכן את ה- reference ברוטינה הנקראת כך שיצביע על הערך החדש.

השיטה זו לא תעדכן כموון אם בשפה יש selective update לבנות וריאנט שלה, הבניוי בשיטה העצלנית שבזה דוחים את הפעולה היקרה של העתקה הערוך המורכב עד לרגע האחרון. כלומר, אין מנגנונים העתקה בזמן הקריאה לרוטינה, אלא רק לרגע שבו יש selective update (זאת מותקן תקווה של עיתים קרובות, רגע התשלום הזה לא ייגע")

**חידוד לפתרון:** "אם אין Selective update אז טיפוסים מרכבים ה- immutable – אי אפשר לשנות אותם, אז הסמנטיקה שמתאפשרת ה- call by reference ו- call by value זהה. לכן, אם אין אפשרות למשcall by valueSelective update במערכות רפנס, ולחסוך העתקה".

### **:Definition vs Declaration<sup>41</sup>**

**נתבונן לדוגמא על שפת C\C++**

- להגדיר מ schovo בשפת C++ זה אומר לחת את כל המידע כדי ליצור מ schovo בשלמותו.
  - **Definition** להגדיר פונקציה אומר להגדיר גם את הגוף של הפונקציה.
  - להגדיר Class אומר להגדיר את כל המתודות והשדות של אותו ה- Class.
  - דוגמא:

```
int func()
{
    return 2;
}
```

**Declaration** – כמשמעותה\פונקציה Class כל מה שאנו מודים זה "יש משווה עם השם הזה והטייפוס הזה" הקומפיילר יוכל לנצל כמעט את כל השימושים בשם זה – בלי להגדיר אותו!

```
int func();  
  
int main()  
{  
    int x = func();  
}  
  
int func()  
{  
    return 2;  
}
```

### **Declaration vs Definition: In Summary**

A declaration provides basic attributes of a symbol: its type and its name. A definition provides all of the details of that symbol—if it's a function, what it does; if it's a class, what fields and methods it has; if it's a variable, where that variable is stored. Often, the compiler only needs to have a declaration for something in order to compile a file into an object file, expecting that the linker can find the definition from another file. If no source file ever defines a symbol, but it is declared, you will get errors at link time complaining about undefined symbols.

### **:Variable Definition**

- "אזור משנה וקשר (bind לו שפ"

**בשיטות אומרים לקומפיבילר (איפה) ליצור את המשנה**

**לדונמא רישרושמיה.**

```
int x;  
int main()  
{  
    x=3;  
}
```

<sup>41</sup> Definition\Declaration\Variable Declaration\Variable Definition is all taken from – [here](#).

השורה הראושונה גם מצהירה על המשתנה וגם מדירה אותו. מה שהיא אומרת זה "צור משתנה עם השם x מטיפוס int ושמור אותו באזור המשתנים הגלובליים".

### **:Variable Declaration**

- "צור קישור (bind) והמשתנה צריך להיווצר במקום אחר".
- פה אנחנו לא מדירים משתנה, אלא מצהירים שקיים מקום שבו המשתנה הזה موجود.
- לדוגמה:

```
extern int x;
```

**- Array Value** – הוא מיפוי של סט אינדקסים לסט ערכים.  
**- Array Variable** – זה השימוש של Array value בעזרת שימוש במשתנים, כך שכל "תמונה" של כל אינדקס יכולה להשתנות בזמן ריצה.

- אפשרים מיפוי יעיל לזכורם בעזרת המודל הקלסי.
- בסיס להרבה אלגוריתמים.
- בסיס להרבה מבני נתונים.

:Expressions<sup>42</sup> – חלקים של ה- L-Value, R-Value

- L-Value: כתובות של משתנה. מה שתוכו מבצעים השמה, הרפנס.
- R-Value: התוכן של משתנה.

### **ישנים סוגים שונים של מערכים:**

#### **- (Static)**

```
const char* days[] = {
    "Sun", "Mon", "Tue",
    "Wed", "Thu", "Fri", "Sat"
}; // An array literal
int main(...) {...}
```

- הגודל נקבע בזמן קומpileציה.
- מוקצה על ה-.data segment.
- סט האינדקסים קבוע ונקבע בזמן קומpileציה.

#### **- (Dynamic)**

```
int[] printPrimes(int n) {
    unsigned char sieve[n];
    ...
    int r[] = malloc(sum(sieve) * sizeof(int));
    ...
    return r;
}
```

- הגודל נקבע בזמן ריצה.
- הגודל לא יכול להשתנות לאחר הייצירה.
- מוקצה על ה-.heap segment.
- סט האינדקסים קבוע בזמן יצירתה של array variable.

#### **- (Stack)**

```
void fileCopy(FILE *from, FILE *to) {
    char buffer[1 << 12];
    ...
}
```

```
void printPrimes(int n) {
    unsigned char sieve[n];
    ...
}
```

- הגודל נקבע בזמן ריצה.
- הגודל לא יכול להשתנות לאחר הייצירה.
- מוקצה על ה-.Stack.
- סט האינדקסים קבוע בזמן יצירתה של array variable.
- הסוג היחיד של מערכים בפסקאל.

#### **- (flexible)**

```
0a = 1..6; # uninitialized; size 6
0a = (1,2,3); # initialized; size 3
0a[13] = 17; # size is now 13
0a[17] = 13; # size is now 17
delete 0a[17]; # size is now 13
delete 0a[13]; # size is now 3
```

- הגודל יכול להשתנות בזמן ריצה.
- הגודל יכול להשתנות לאחר הייצירה.
- נמצא ב-.Perl.
- סט האינדקסים לא קבוע. גבולות האינדקסים יכולים להשתנות בעת גישה לאינדקס כלשהו – התא נוצר ונוצרים גם התאים של כל האינדקסים שלוינו.

<sup>42</sup> Definition taken also from Wikipedia: [https://en.wikipedia.org/wiki/Value\\_\(computer\\_science\)](https://en.wikipedia.org/wiki/Value_(computer_science))

```

$wives["Adam"] = "Eve";
$wives["Lamech"] = "Adah, and Zillah";
$wives["Abraham"] = "Sarah";
$wives["Isaac"] = "Rebecca";
$wives["Jacob"] = "Leah, and Rachel";
:
:
echo $patriarch;
echo $wives[$patriarch];

```

### מערכות אסוציאטיביים (associative)

- האינדקסים יכולים להיות כל דבר, ובפרט מחרוזות.
- נפוץ בשפות סקריפטינג, כמו AWK, PHP, JavaScript.
- בד"כ ממומש בעורת Hash Table.
- סט האינדקסים הוא לא גבולות. הוא משתנה בצורה דינמית כמשמעותם או כשמורידים ערכים מהמערך.
- הערות<sup>43</sup>:
  - שימוש מסורבל.
  - נדרש הרבה זיכרון לשמר גם את השמות של האינדקסים.

נבחן כי ל- Static, Stack, Dynamic based arrays – ישimplementations יעילים במודול הזיכרון הקלאסי. עבור מערכים גמישים אסוציאטיביים – יש צורך ביצירת מבנה נתונים יותר מורכב כדי למפות את הנתונים במודול הזיכרון הקלאסי. Multiple Dereferencing. השיטה של שפת C ליצוג מערכים דו מימדיים נקראת

### הסרנות ותרונות של מערכים עם טיפוס אינטגרלי לאינדקסים:

#### תרונות:

- רק ערכים נשמרים – לא אינדקסים.
- יש תיאור פשוט של האינדקסים החוקיים.
- גישה פשוטה בעורת שימוש בחיבור:
- בזורה מפורשת: ב- C או C++ אריתמטיקה נעשית בצורה מפורשת:  $a[i] == (a + i) == (i + a) == i[a] = i[a]$ .
- בזורה לא מפורשת: לדוגמה ב- Java הגישה למערכים מתורגמת לפקודות מכונה פשוטות.

#### הסרנות:

- כאשר יש המון מידע שטוף, יש צורך בטכניקות לדחיסת המידע.
- תכנות לא גמיש.

למערך דו מימדי יכולים להיות כמה ייצוגים בזיכרון: מערך חד מימדי ובו מופיעים איברי המטריצה שורה אחר שורה, מערך חד מימדי ובו מופיעים איברי המטריצה עמודה אחר עמודה. שיטת ייצוג נוספת היא מערך של מערכים. כל תא במערך הראשי מצביע על מערך שמייצג שורה במטריצה.

### :<sup>44</sup>Row major order \Column major order

- צורות שונות לסידור מערכים דו מימדיים במקומות אכסון רציף, כמו הזיכרון.
- עבור המערך הדו מימדי,  $3 \times 3$  הבא:

11	12	13
21	22	23

- הסידור בזיכרון יראה עבור הצורות השונות כך:

כתובת: Column Major Order						כתובת: Row Major Order					
0	1	2	3	4	5	0	1	2	3	4	5
11	12	13	21	22	23	11	21	12	22	13	23

זמן חיים של משתנה הוא הזמן בין רגע הקצאתו של משתנה לבין רגע השחרור שלו.  
ונכל למצוא סוגים שונים עבור משתנים:

<sup>43</sup>手册 56 שאלות ותשובות של אריה לבב.

<sup>44</sup> [https://en.wikipedia.org/wiki/Row-major\\_order](https://en.wikipedia.org/wiki/Row-major_order)

## Main Varieties of Lifetime<sup>5</sup>

- Persistent/Permanent lifetime (continues after program terminates)
- Global/Program Activation lifetime (while program is running)
- Local/Block Activation lifetime (while declaring block is active)
  - Heap lifetime (arbitrary)
  - Garbage Collected lifetime (arbitrary)

### :Persistent \ Permanent variable -

- הגיון: טוב לשימוש בישיות כמו אכソン קבצים (קבצים על דיסק), מסדי נתונים וצדומה.
- ההילך: מhabצע דרך פעולות קלט/פלט.
  - ב- C לדוגמה: fopen(), fwrite() וכו'.
- Java Serialization
- אוריך חיים שאינו תלוי בתכנית.

### :Global/Program Activation/Static variable -

- חי כל זמן שהחכנית רצה.
- דוגמה:
  - בפסקל כל משתנה שהוגדר ביחד עם התכנית הראשית הינו Global.
  - ב- C כל המשתנים שהוגדרו מחוץ לבlokים הם גלובליים.
  - ניתן לגשת אליו מכל חלק בתכנית.

### :Local/Block Activation/Automatic -

- חיים בתחום הבלוק שלהם.
- קיים כל עוד הבלוק "פעיל".
- ושבים על ה-.stack

### :Heap variables -

- משתנים שזמן החיים שלהם הוא מרגע הקצאתם (לרוב ע"י המשתמש אום לעתים ע"י השפה כמו במקרה של Closure), ועד לרגע שהורום שמתבצע ע"י המשתמש או ע"י מגנון איסוף זבל.
- משתנים "לא שם".

### :Garbage Collected -

- קיים מרגע הקצאה עד לאיסוף ע"י ה- Garbage Collected האוטומטי.

### מהו בלוק?

- בפסקל – פונקציות או פרוצדורות.
- ב- ML מה שתחת: "let".
- ב- C או C++ - פונקציות וגם מה שתחת "בונה הפקודות" בונה הפקודות ("{..}" (Command constructor))

מתי בלוק הוא ?Active \ מתי יש Activation לבלוק? מתי שהפקודות בו מבוצעות.

ב- C++ נוכל למצוא מיללים מיוחדים שתפקידו לציין את ומן החיים של המשתנה:

### :Storage Class Specifiers

## Approximate meaning of C's storage specifiers

- **auto:** block activation  
*block variables with no storage-class specifier default to auto*
- **register:** same as **auto**, but with recommendation to place in a register
- **static:** program activation
- **extern:** program activation  
*but declaration must be done somewhere else*
- **typedef:** empty lifetime variables  
*exists during compilation, as a template for defining other variables*
- **thread\_local:** thread lifetime  
*not in the scope of this course*

נבחן במנוחים:

.auto – מצוין בעזרת **auto**. מוקצתה על **block activation variable**  
.static – מצוין בעזרת **static**. מוקצתה על ה- **program activation variable**

(לא מדובר ברפונס של C++)<sup>45</sup> **Pure reference** :-  
Pure reference הוא ערך שדרכו תכנית עשויה שימוש כדי לגשת בזיכרון לא ישירה למשתנה (בד"כ משתנה ערימה).  
- נבחן כי ה- "pure" דורשת שכל הרפונסים יספקו גישה למשתנה כלשהו. ולכן רפונסים שניתן להצביע עליהם לא נחכבים.  
- אנו אומרים ש- reference מתייחס למשתנה.  
- הפעולה של שימוש ברפונס כדי לגשת ל"משתנה" שמתיחס אליו. Dereference

---

<sup>45</sup> \_beamer-5-+-storage.pdf – page 21/149

## Many realizations of references

- Address** most commonly, references are nothing but memory addresses, in which case, they are called *pointers*
- Offset** references may be implemented as offsets from a fixed address.
- Array index** in a language that forbids manipulation of memory addresses, references may be realized as array indices
- Handle** Index into an array which contains the actual pointer.
- Smart pointer** an abstract data type that extends the notion of pointers, while providing services such as
  - computing frequency of use
  - reference counting
  - lazy copying
  - caching
  - legality of access checking
  - ...

## The C++ "references vs. pointers" confusion

C++	Java (and many other PLs)
<b>Pointer</b> , i.e., pointer variable	<b>Pointer?</b>
<ul style="list-style-type: none"> <li>• May be "0", or point to a variable</li> <li>• Can change</li> <li>• Must be explicitly "dereferenced"</li> </ul>	<ul style="list-style-type: none"> <li>• no such beast;</li> <li>• sometimes used as a synonym for "reference"</li> </ul>
<b>Reference</b> , i.e., reference variable	<b>Reference</b> , i.e., reference variable
<ul style="list-style-type: none"> <li>• May not be "0"</li> <li>• Must point to a variable</li> <li>• Cannot be changed</li> <li>• No "dereferencing" prior to use.</li> <li>• Is <i>pure reference</i></li> </ul>	<ul style="list-style-type: none"> <li>• Is disjoint sum of pure-reference and <i>Unit</i></li> <li>• May be "null", or point to a variable</li> <li>• Can change</li> <li>• No "dereferencing" prior to use.</li> <li>• Is <i>not</i> pure reference</li> </ul>

## The "null" pointer

- Strictly speaking, the "pure" definition requires that all references provide access to some variable.
- Still, it is useful to have references which "refer to nothing"; e.g., for designating the end of a linked list.
- It is possible to realize "refer to nothing" as reference to a special variable.
- It is more convenient to allow a special, illegal value of references instead. This value is known in different languages as `null`, `nil`, `void`, `nullptra`, `0`, etc.
- In JAVA and many other languages, references are disjoint sum of "pure references" and *Unit*.
- C++'s *references are nothing but immutable, pure references*.

<sup>a</sup>A new C++ keyword

## How are dangling references created?

### I. Freed memory

a deallocated heap variable:

```
char *p = malloc(100);
strcpy(p, "Hello,World!\n");
free(p);
// p is dangling
strcpy(p, p + 5); // X
```

### II. Reference to stack

reference to a "dead" automatic variable:

```
char *f() {
    char a[100];
    return &a;
}
char *s = f();
// s is dangling
strcpy(s, "Hello,World!\n"); // X
```

### III. Inner functions

Activating a function outside the enclosing block in which it was defined, in the case that the function uses variables local to the block

```
// Provide name for function type:
typedef void (*F)(void);
// Forward declaration
F f();
// of function returning a function
F h = f();
h();
// May access a dangling reference
```

```
F f() {
    char a[100];
    // only Gnu-C allows inner functions
    void g(void) {
        // a is dangling if g
        // is called from outside f
        strcpy(a, "Hello,World!\n"); // X
    }
    return g;
}
```

רפרנסים כאלה ייווצרו במקרה שיצבינו על:

1. משתנה שהוקצה על הירימה ושהחר.
  2. משתנה מהסנית שזמן החיים שלו נגמר.
  3. פונקציה שנמצאת בתוך בלוק זר, שלא רץ
- כרגע, ובה יש שימוש במשתנה מהסנית שוגדר בפונקציה החיצונית יותר.

	Freed memory	Stack reference	Inner functions
C	X <i>programmer's responsibility</i>	X <i>programmer's responsibility</i>	✓ <i>no inner functions</i>
Gnu-C	X <i>programmer's responsibility</i>	X <i>programmer's responsibility</i>	X <i>programmer's responsibility</i>
PASCAL	X <i>programmer's responsibility</i>	✓ <i>cannot take the address of stack variables</i>	✓ <i>inner functions cannot leak</i>
JAVA	✓ <i>with garbage collection, programmer never deallocates memory</i>	✓ <i>objects are always drawn from the heap; stack variables are scalars or references to objects; one cannot take the address of these</i>	✓ <i>functions are not 1<sup>st</sup> class; their address cannot be taken, and they cannot leak</i>

### טיפול ב- **dangling reference**

- במקרה הטוב – התכנית תתרסק מיד.
- במקרה הרע – התכנית תתרסק אבל לא מיד.
- במקרה הכי גרוע – התכנית לא תתרסק בזמן הבדיקות – יישאר בה באג.

### Enumeration – טיפוס אוטומי<sup>46</sup> לא פרימיטיבי<sup>47</sup>:

- בד"כ מומפה לערכיהם של integer.
- שפות התכנות בד"כ לא מראות איך הן משתמשות את זה.
- **Enum** והוא בניי טיפוס ( Type Constructor ) – הוא יוצר טיפוס חדש!
- o אפשר להסתכל על הטיפוס הזה בשתי דרכים שונות (פרופ' גיל):
  - g. טיפוס אוטומי – שכן אין בו טיפוסים אחרים.
  - 7. טיפוס מורכב – שנוצר ע"י בניי שאין לו ארגומנטים מסווג טיפוס.
- ב- C זה יצירה "קבוע" החדש – יצירה שם למספר מסוים. לנכון, גם לא ניתן לקחת את הכתובת שלו.
- ובפרט – הוא לא שוכן בזכרון בשום מקום!
- Java –
  - o enums הם ערכים אודידיינליים עם קבועה של פעולות מיוחדות בשביבם.
  - o אפשרות פונקציות שונות על אותם איברים של ה- enum שנמצאים תחת אותו enum.

### Enumerated Type (טיפוסים מנויים) – Disjoint Union

<sup>46</sup>טיפוס אוטומי – טיפוס שלא ניתן לפرك את ערכיו לבני יסוד קטנות יותר. בפרט, אין ערכים מティפוס אחר המוכלים בתחום ערכי הטיפוס האוטומי. טיפוסים שהוגדרו ע"י המוכן (למשל Type בפסקול) או שהוגדרו ע"י השפה (מודדר לפני כן).

<sup>47</sup>טיפוס פרימיטיבי – טיפוס שהוגדר מראש ע"י שפת התכנות ושמו הוא מילה שморה, בדרך כלל (גם כן הוגדר לפני כן).

<sup>48</sup>נלקח מ- StackOverflow – קישור.

<sup>49</sup>נלקח מ- StackOverflow – קישור.

**Definition 3.12** (Enumerated type constructor).  
If  $\ell_1, \ell_2, \dots, \ell_n \in \mathbb{I}$ ,  $n \geq 1$  are labels, then  $\{\ell_1, \ell_2, \dots, \ell_n\}$  is an **enumerated type**, whose values are  $\ell_1, \ell_2, \dots, \ell_n$ .

נתן להסתכל עליו כ- "איחוד זר" של <sup>50</sup>:  
Branded unit types

An enumerated type can be thought of as a disjoint union of branded **Unit** types:

$$\{\ell_1, \ell_2, \dots, \ell_n\} = \ell_1(\text{Unit}) + \ell_2(\text{Unit}) + \dots + \ell_n(\text{Unit}) \quad (3.3.26)$$

מהם המילולונים של טיפוס שכזה? <sup>51</sup>  
פתרונות: שמות התగיות שבאמצעות הוא נוצר.

דוגמא ל- enumerated Type אפשר למצוא בשאלת הבהא:

- **52 An enumerated type can be viewed as a choice between brands of the unit type! TRUE OR FALSE?**

הפעלה נאייה של המשפט לעיל בשפת C תביא למסקנה שהטענה אינה נכונה.  
אולי, בדיקת הפעלה זו, תגלה שיש בהบาง.  
א. מהי הפעלה הנאייה זו.  
ב. מדוע הפעלה זו מביאה למסקנה שהמשפט אינו נכון.  
ג. מהו הבאג בהפעלה של המשפט על השפה?

פתרונות:

"א. הפעלה נאייה של המשפט:

```
union suit {
    struct {} spades;
    struct {} hearts;
    struct {} clubs;
    struct {} diamonds;
};
```

עכשו כביכול נוכל ליצור משתנה מסוג suit שיכל להכיל כל אחד מהערכים (כלומר הוא Enumeration של הערכים) בעלות של 0 בתים בזיכרון (כמוון זהה לא עובד).

ב. הפעלה מביאה למסקנה שהמשפט אינו נכון. בשפת - C אין דרך קבועה למשתנה את אחד מערכי ה- Unit.

ג. הבאג בהפעלה המשפט על שפת C הוא שפה ב- Choice type (ב- Disjoint Union Choice type) אמיית כיון שב- C הבניי union לא מבצע Branding, ולכן המשתנה מסוג Suit יכול את הערך של unit אך לא יכול את המידע של "לאיזה מבחן-unit"ים הורם המשתנה שייך, ככלمر בפועל לא ניתן להבדיל בין הערכים של המשתנה.

שאלה נוספת שsspft שקשורה:

- **53 An enumerated type can be viewed as a choice between brands of the unit type!**

זהה את כל המונחים שנלמדו בקורס המופיעים במשפט לעיל. לגבי כל מונח, רשום את כל הסימונים/מונחים אלטרנטיביים בעברו. כן ציין את המונח/ים המקבילים בעברית.

(התשובה לא אושרה ע"י פרופ' גיל)

טיפוס המוגדר על ידי המשמש. המשמש מגדיר את קבוצת כל הערכים השיכנים לטיפוס על ידי Enumerated type. שימוש במבנה (לדוגמה ב-C, הבניי הוא המילה השמורה enum).

<sup>50</sup> Type Branding - יצירת טיפוס חדש על סמך טיפוטים אחרים קיימים. (הוגדר והסביר לפני כן).

<sup>51</sup> נלקחה מהאתר – quia.org.

<sup>52</sup> נלקחה מהאתר – Safot.net – quia.org.

<sup>53</sup> נלקחה מהאתר – Safot.net – quia.org.

- טיפוס שקבוצת הערכים שלו מכילה רק ערך אחד. לדוגמה, הטיפוס שנקרא unit ב-ML הוא בדיקת כזה.

(מייתוג) - לכל טיפוס יש קבוצת ערכים שימושה מהטיפוס זהה יכול לקבל. בנוסף, בשפות שתומכות ב- Branding, טיפוסים שונים מכילים "מייתוג" שונה, כך שימושיים מטיפוסים שונים (אפילו אם יש להם את אותה קבוצת ערכים) ייחסבו שונים. דוגמה: בפסקל ניתן כתוב Kilograms = IntegerTYPE Kilograms ואז משנתה מטיפוס לא Kilograms (מייתוג) ייחסו שניים. בנוון שאין Coercion אוטומטי ביניהם. בניסוח אחר, בשפה שתומכת ב- Branding השמור משנתה מטיפוס T מכיל גם ערך וגם את העובדה שהמשנתה הוא מטיפוס T. זאת בניגוד ל-aliasing Type (לדוגמה typedef ב-C) שמאפשר לתת שם חדש לטיפוס קיים, ואז משנים מני הטיפוסים האלה יהיו שווים.

- טיפוס מורכב שקבוצת הערכים שלו היא איחוד זר (Disjoint Union) בין קבוצות הערכים של מספר טיפוסים אחרים. בנוסף, ב-Choice Type שמשמעותו כהלה ניתן לבצע פעולה Tag Testing ולgelot לאיזה מטיפוסים המקוריים שייך הערך של המשנתה ברגע נתון.

משמעות המשפט כתוב בcourt: טיפוס שהוא Choice Type Enumerated אפשר לתאר גם על ידי שמקבל מספר פעמים טיפוס Unit (בכל פעם עם Tag אחר). משנתה מטיפוס כזה (של ה-Choice Type שתואר) יוכל לקבל בכל פעם רק את הערך Unit, אך בנוסף על ידי Tag Testing יוכל לדעת לאיזה מה-Brand-ים של Unit הכוונה. דוגמה בשפת ML שבה הדבר אפשרי מופיעה בשקפים (שוף 214):

```
datatype suit = diamond of unit
| heart of unit
| spade of unit
| club of unit;
```

כעת נוכל לכתוב:

```
val s = diamond;
```

ולכתוב פונקציה שמקבלת משנתה מסוג suit ובבודקת מאיזה מותג הוא על ידי pattern matching ".pattern matching

ברוב המקרים:

- ערכים של "טיפוסים אוטומטיים" הם "ערכים אוטומטיים".
- ערכים אוטומטיים שייכים לטיפוסים אוטומטיים.
- בעיה: טיפוסים מורכבים שהערכים שלהם אוטומטיים.
- מצביעים - טיפוסים אוטומטיים שהערכים שלהם מורכבים.
- הערכים של הטיפוס האוטומי string, כשהוא קיים, לא אוטומטיים.

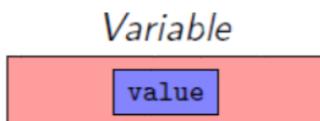
ייצוג טיפוסים בזיכרון:

- לרוב שפת התוכנות מחלטה על הייצוג הכי טוב לטיפוסים, ובכך מאפשרת לתוכנה שלא להתייחס למימוש של הייצוג ולהשתמש בטיפוסים על פי צורך.

## :Value Semantics vs Reference Semantics

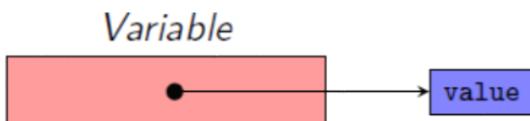
## Variables: reference vs. value semantics

### Value Semantics.



- Variable contains the actual value.
- C, C++
- JAVA for builtin, atomic types

### Reference Semantics.



- Variable contains a reference to a value which is stored elsewhere.
- C, C++, if pointers or references are used
- JAVA for all other types, including arrays
- Most modern languages

- סמנטיקת ערך משמעותה שהערך עצמו נשמר במשתנה בזיכרון.  
- סמנטיקת רפנס משמעותה שבמשתנה נשמר מצביע לערך.

### Values vs. reference semantics in JAVA

The basic type system of JAVA is defined by:

8 atomic types: byte, short, int, long, float, double, boolean, char

1 pseudo type: void

4 type constructors:

- array
- class
- interface
- enum

- Precisely 8 types in JAVA follow value semantics
- All the rest are reference semantics

### "Integer" vs. "int" in JAVA

- Each wrapper classes (except for Void) wraps a value of the corresponding primitive type.
- Wrapper types are almost fully interchangeable with their primitive equivalents:

```
int v = 3; // Primitive type
Integer r = new Integer(a); // Wrapper type
v = r.intValue(); // Explicit conversion
v = r; // auto un-boxing
r = v; // auto boxing
```

:Java - לדוגמא ב-

### Wrapper classes

- When generics were introduced to JAVA, it was discovered that the implementation was much simpler for reference types.
- The 8 value types did not justify extra machinery:
- Instead, the JAVA library introduced reference type equivalents

Integral types Byte, Short, Integer, Long

Floating point types Float, Double

Other types Boolean, Character

Unit types Void

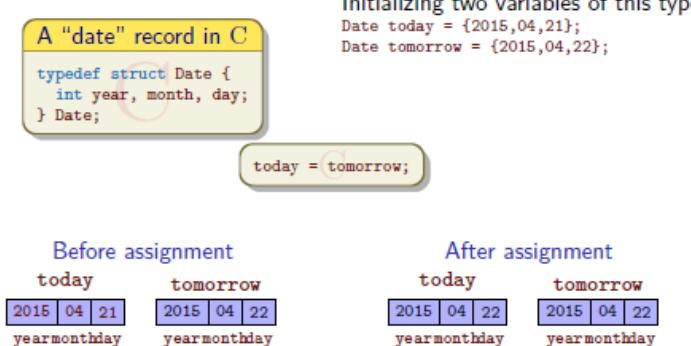
```
List<double> ds1; // X compilation error;
// type double is not
// a reference type

List<Double> ds2; // ✓ works fine;
// type Double is
// a reference type
```

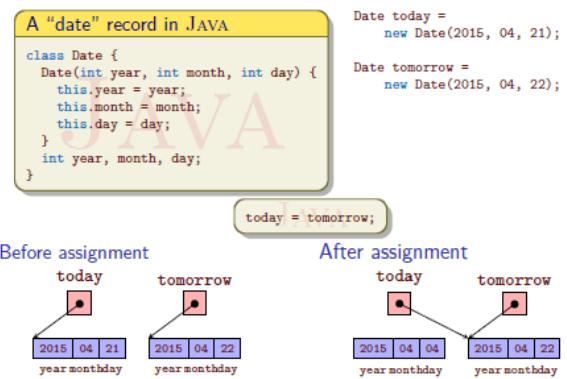
- הערכות נוספות על Integer ו- int ב- Java
- Integer מכיל גם את הערך null.
- לא מכיל את הערך null.
- נקבע שגייאת זמן ריצה עברו:

```
Double dd = null;  
double d = dd; //<-Runtime error
```

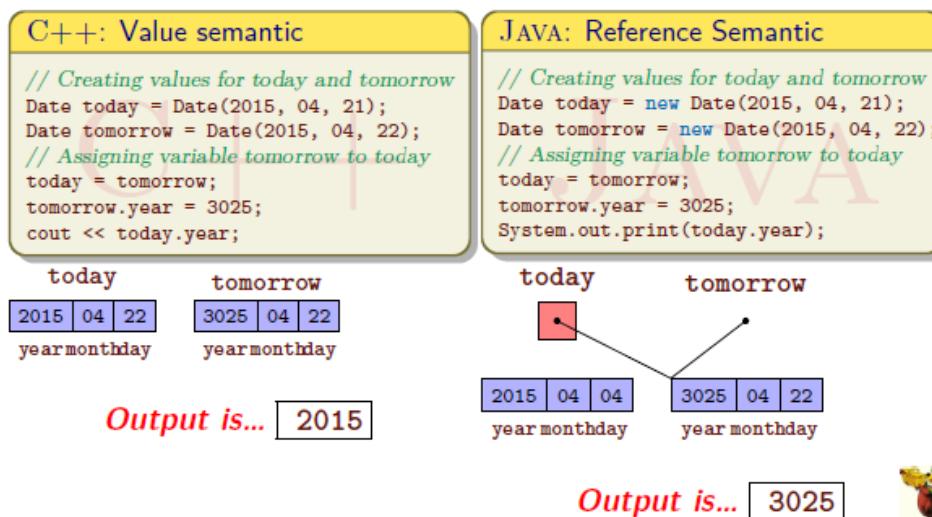
## C: value semantic of assignment



## Reference semantic of assignment in JAVA



נניח כי הטיפוסים הפרימיטיביים ב- Java מtab'זים ב- int, double etc. – "Value semantics" ב- Integer, Double etc. – "Reference Semantics" כל שאר הטיפוסים מtab'זים ב- genericity – "Reference Semantics" !First Class Type



הבדלים נוטפים בין Value Semantics של C++ ו- Java לבין reference semantics של

## C++ vs. JAVA<sup>13</sup>

### C++: Value semantics

```
class Date {public:
    int year,month,day;
    Date(int year, int month, int day) {
        this->year = year;
        this->month = month;
        this->day = day; }};
Date today(2015, 04, 21);
Date tomorrow(2015, 04, 22);

today = tomorrow;
tomorrow.year = 3025; operator overloading
cout << today.year;
```

*variable contains value*

### JAVA: Reference semantics

```
class Date {
    int year,month,day;
    Date(int year, int month, int day) {
        this.year = year;
        this.month = month;
        this.day = day; }}
Date today = new Date(2015, 04, 21);
Date tomorrow = new Date(2015, 04, 22);

today = tomorrow;
tomorrow.year = 3025;
System.out.print(today.year);
```

*variable refers to value*

*No operator overloading*

**Reference Semantics differences:**  
"=>" , "=="

באייזה סמנטיקה משתמשים ב- ?ML

- זה נראה כמו Value Semantics אבל במציאות ML וJAVA הרבה שפות אחרות משתמשות ב-
- באופן הזה שהמוכנת לא יכול "לראות" את ה-Reference בתכנית.
- המימוש מטבח בזרת רפנסים – מאחורי הקלעים.

### Java - ב- Value Semantics vs Reference Semantics

**Value Semantics**  
- ניתן לעשות שימוש ב-  
- בJAVA ע"י שימוש  
:Cloneable

#### Value semantic in JAVA?

Of course!  
*You just have to be explicit about it!*

```
class Date implements Cloneable {
    :
}
Date today = new Date(2015, 04, 21);
Date tomorrow = new Date(2015, 04, 22);

today = (Date) tomorrow.clone();
tomorrow.year = 3025;
System.out.println(today.year);
```

*Output is... 2015*

#### JAVA: Reference Semantic

```
// Creating values for today and tomorrow
Date today = new Date(2015, 04, 21);
Date tomorrow = new Date(2015, 04, 22);
// Assigning variable tomorrow to today
today = tomorrow;
tomorrow.year = 3025;
System.out.print(today.year);
```

*Output is... 3025*

### C++ - ב- Value Semantics vs Reference Semantics

```
// Creating values for today and tomorrow
Date today = Date(2015, 04, 21);
Date tomorrow = Date(2015, 04, 22);
// Assigning variable tomorrow to today
today = tomorrow;
tomorrow.year = 3025;
cout << today.year;
```

**Output is... 2015**

## Reference semantic in C++?

Of course!

You just have to be explicit about it!

```
class Date {...};

// Storing references to newly allocated values
// of today and tomorrow
Date *today = new Date(2015, 04, 21);
Date *tomorrow = new Date(2015, 04, 22);
```

```
today = tomorrow; // Leak!
tomorrow->year = 3025;
cout << today->year;
delete tomorrow;
delete today; // Heap corruption?
```

**Output is... 3025**

## :<sup>54</sup>Lazy Copying

- טכניקה הממשת "Value Semantics" ע"י כך שהעתק של עצם גדול נוצר ע"י יצירה רפרנס לעצם המקורי - אותו רוצים להעתיק. ההעתקה האמיתית מתחבצע רק אם וכאשר מנסים לשנות את ה- variables.
- קונספטואלית דומה למנגנון Copy On Write בקורס "מערכות הפעלה".
- טכניקה זו היא הכללה של הגישה של LISP.
- העתקה של מבנים קבועים (struct) למיניהם לדוגמה – יכולה לחתה המון זמן, لكن הומצא ה- Lazy Copying.

.Void Safety – גישה למצביים שהם null.

## מהו ?Void Safety

- בקורס למדנו כי כשאומרים על שפה שיש בה Null Dereference, הכוונה היא שלא ניתן לבצע Dereference .Pointers

ישנן הגדירות נוספות לפיהן המונח מתיחס, ראשית, לשפות מונחות עצמים בלבד, ויתריה מכך - הוא מתייחס לכך שלא ניתן ליצור או לאפשר Reference לאובייקט להיות Void/Null, וכך שעם השאלה האם ניתן לבצע Dereference כל אינה רלוונטית.

## או מהו בדיקת ?Void safety

ושובתו של פרופ' גיל:

בティוחות void יכולה להיות בכל שפה, גם אם היא לא מונחת עצמים. בדרך כלל, מדובר על שפות שיש בהן reference semantics. בשפות כאלה, יכולה להיות בעיה, והבעיה אכן קורית, של גישה למצביים שהוא NULL.

אם השפה היא STRONGLY TYPED, אז נדרש מנגנון של השפה כדי למנוע את הפעולה האסורה. ב Java זהה מופעל בזמן ריצה, וזאת בדומה לכל שגיאת טיפוס הנובעת מכך שפעולות מסוימות שבדרכן כל מותרות לגבי כל הערך של השפה, אין חוקיות לגבי ערכיהם מסוימים בטיפוס. דוגמאות נוספות לשגיאות מסווג זה הן חלוקה באפס, חירגה מגבולות מערך, הוצאה שורש מספר שלילי, וגם הפעלת פונקציה כגון ARCSIN שיכולה לקבל רק ערכים שערכם המוחלט אינו עולה על אחד.

בכל השגיאות מהסוג זה, שיטת הטיפול בשפה שיש בה טיפוסיות חזקה, היא באמצעות בדיקות בזמן ריצה. כך קורה ב JAVA. בשפות שיש בהן טיפוסיות חלשה, כמו C, ההנתנות במרקרים אלו אינם מוגדרת, והomonת הרלבנטי הוא קלומר, אין דרך לדעת בודאות מראש מה יקרה, ועל המתכנת לשבור את הראש. POSTMORTEM TYPING

הomonת VOID SAFETY מתייחס למאיצים למנוע שגיאה של גישה לערך שהוא NULL עוד בזמן היזור, תוך שימוש בבדיקה טיפוסים סטטיות. בשפת C++, יש שימוש ב REFERENCES של השפה, שאינם REFERENCES עליהם מדברים בהקשר הרחב יותר. מדברים C++ REFERENCES של REFERENCES נראים כך:

```
int i;
int &j=i;
```

שפה C++ מבטיחה, עוד בזמן היזור, כי משתנה שהוא REFERENCE לעולם לא יהיה NULL. וכך מתקיים באופן חלקי VOID SAFETY.

<sup>54</sup> \_beamer-5-+-storage.pdf page 232

<sup>55</sup> נלקח מהאתר Safot.net – קישורים.

בשפת JAVA לעומת זאת, ישנו ניסיון להרחבת שביהם יוגדר עיתור ANNOTATION למשתנים, שיקבע שם NonNull@.

כאשר ישנה השמה במשתנים עם Reference Semantics יש כמה דרכים אפשריות שבהן ההשמה מתחילה מאחורி הקלעים:

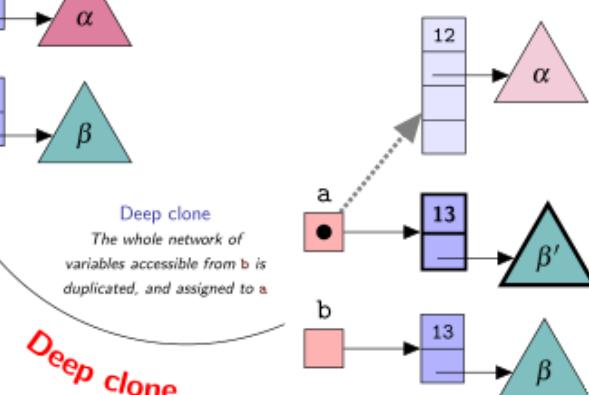
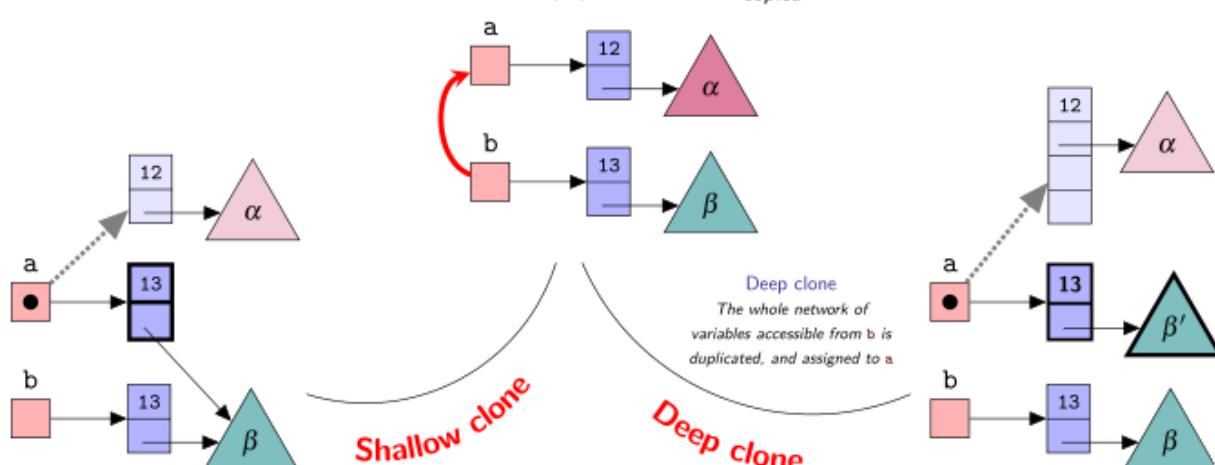
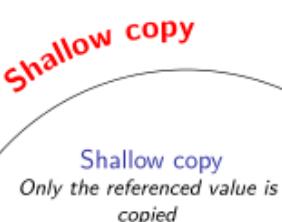
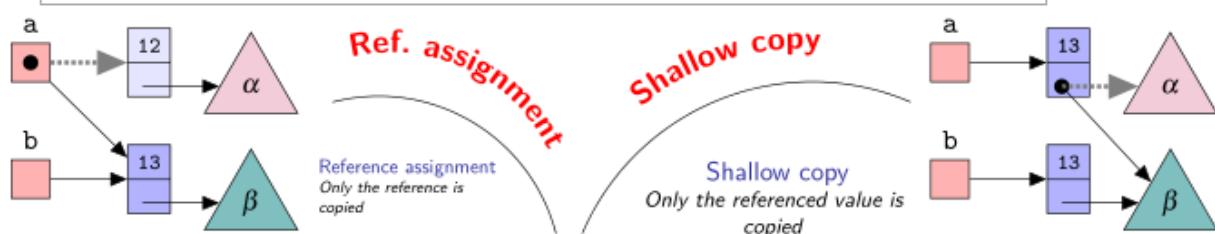
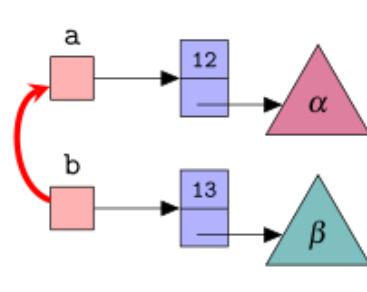
(Shallow copy, Deep copy, Reference assignment, Deep clone)

## Assignment strategies side by side

More generally, in any reference semantic programming language:

- Given two variables, **a** and **b**,
- each containing a reference to a value,
- which may include references to a network of variables,
- and an assignment command

`a := b;`



The variable itself is cloned, but all the references inside it are copied, rather than being cloned

ב- C++ אופרטור ההשמה מבצע באופן דיפולטיבי Shallow Copy

שאגיאת null pointer assignment רק עבור shallow copy

הסביר שלוי: הסיבה לכך null pointer assignment יכול להתרחש רק עבור shallow copy היא שלדוגמא אם a ו- b מצביעים לאותה כתובת זיכרון, או כשהנעשה  $a = *b$  נקבל שגיאת null.

ב- shallow clone וב- deep clone זה לא יכול לקרות משום ששם מובוצעת הקצאה של זיכרון עבור a לפני העתקה עצמה.

ב- Ref. Assignment זה גם לא יכול לקרות משום שהוא רק מעתקים את המצביע.

הסבירים נוספים מתחובות של פרופ' גיל (השאלות והתשובות רשומות בהמשך). לשם השלמות אוסף את הסבירים כאן:

- ב-value semantics, משתנה הוא שם של תא בזכרון. מהרגע שהמשתנה נוצר, התא קיים.
- ב-shallow clone, ערך מסוובט באופן הבא: הרמה הראשונה משובטת, ולאחר מכן המשובטים, יועתקו הערכים: ההעתקה היא גם של מצביעים, וגם של ערכים שאינם מצביעים.

הסבירים נוספים (אושרו ע"י פרופ' גיל):<sup>56</sup>

<sup>56</sup> נלקח מהאתר Safot.net – קישור.

"Shallow clone" יוצר עצם חדש שבו כל השדות הפשטוטים מועתקים וכל השדות הלא פשטוטים מועתק רק הרפרנס שלהם. Shallow copy לא יוצר עצם חדש, מעתיק את כל השדות הפשטוטים וכל השדות הלא פשטוטים מעתיק את הרפרנס שלהם. לעומת זאת, shallow copy מעתיק לתוך תא כלשהו שאנו לא יודעים אם הוא Null או לא, יכול מאוד להיות שבוצע Null אחר ו- assignment, קלומר שגנסה לשם לתוכו. לעומת זאת, shallow clone יוצר עצם חדש ואז נשים לתוכו, לא ניתן כי נשים לתוכו Null, לאחר והעצם הוא Null כי יש לנו שליטה עליו ושמנו אותו כרפרנס לחלק שהוא".

הבדל בין Deep Clone ו- Shallow Copy יתיק רק את הכתובת בעוד Shallow Copy יתיק גם את הכתובת וגם את התוכן.

ובאשר ל- "סכנות" הטമונות בשיטות ההעתקה השונות:

SEMANTIC ASSIGNMENT?	NULL POINTER ASSIGNMENT?	MEMORY ALLOCATION?
<b>Reference assignment</b>	<i>Never</i>	X
<b>Shallow copy</b>	<i>Maybe</i>	X
<b>Shallow clone</b>	<i>Never</i>	✓ (bounded)
<b>Deep clone</b>	<i>Never</i>	✓ (unbounded)

### What does JAVA `clone()` do?

- |                   |  |
|-------------------|--|
| Runtime Exception | • if the class <code>does not implement interface Cloneable</code>   |
| Shallow clone     | • if the class implements <code>interface Cloneable</code> , and<br>• the programmer <code>does not override</code> the default <code>clone()</code> method.   |
| Whatever          | • if the class implements <code>interface Cloneable</code> , and<br>• the programmer overrides the default <code>clone</code> method<br>• <code>in whatever way he likes</code> .                        |
| Deep clone        | • if the class implements <code>interface Cloneable</code> , and<br>• the programmer overrides the default <code>clone</code> method, and<br>• <code>correctly implements a "deep clone" semantic</code> |

### Overview: semantics of assignment

- Assignment semantics is defined by the language design:
  - C structures follows value semantics.
  - C used to place restrictions on passing structures by value.
  - Arrays cannot be assigned.
  - Pointers are used to implement reference semantics.
  - JAVA follows value semantics for primitive types.
- Value semantics may be slower
- Reference semantics may lead to sharing problems.
- Reference semantics is more expressive.

שאלות הקשורות לנושא –

<sup>57</sup> לא אושרה ע"י אף אחד ולכן יש לקחת בעברון מוגבל:

אם השפה משתמשת ב- **shallow copy** ייחד עם **reference semantics**, האם ניתן כי תהיה שגיאת **null pointer** בהצבה? הסבר מה هي שגיאת **null pointer**

תשובה – לא מאושירה ע"י אף אחד: "יכולת להיות שגיאת null pointer מכיון שהוא גורם לכך לגשת להעתיק לתא בזיכרון שיכל להיות null, ואז כמונן שתקבל שגיאת null pointer".

יש להבדיל למשל מ- **deep clone**, בו פשוט העצם מועתק ומוקצת מחדש עם הרפרנסים שלו, ושם אתה לא יכול לקבל שגיאת null pointer (כי \*אתה\* מוצא את התאים הדורשים). נ"ל לגבי null pointer

שגיאת null pointer מתרחשת כאשר אתה מנסה לגשת לחא בזיכרון שיש בו `NULL`.  
<sup>58</sup> צרכ' הנימה שתשכנע כי לא ניחסת את התשובה. אין להסביר מבלי לנמק. אפשר לנמק בקצרה. (בכל השאלות הנה שהקצתה זיכרון אינה יכולה להכשיל).

<sup>57</sup> נלקח מהאתר Safot.net – קישור.

<sup>58</sup> נלקח מהאתר Safot.net – קישור.

- אם השפה משתמשת ב- **value semantics**. האם יתכן כי תהיה הקצתה זכרון בהצבה? אם כן, האם ניתן לקובע בזמן הידור חסם על כמות הזיכרון המוקצת?
  - אם השפה משתמשת ב- **reference semantics** יחד עם **shallow clone**. האם יתכן כי תהיה הקצתה זכרון בהצבה? אם כן, האם ניתן לקובע בזמן הידור חסם על כמות הזיכרון המוקצת?
  - אם השפה משתמשת ב- **value semantics** יחד עם **null pointer**. האם יתכן כי תהיה רשימה מעגליות?
  - אם השפה משתמשת ב- **reference semantics** יחד עם **deep clone**. האם יתכן כי תהיה שגיאת **null pointer** בהצבה?
  - אם השפה משתמשת ב- **reference semantics** יחד עם **shallow clone**. האם יתכן כי תהיה שגיאת **null pointer** בהצבה?
- מקור:** סתו תשע"ה, מועד ב'.  
**פתרון ע"פ פרופ' גיל:**
- לא. לא תהיה כל הקצתה זיכרון בהצבה. ב- **value semantics**, משתנה הוא שם של תא בזכרון. מהרגע שהמשתנה נוצר, התא קיים. אין פעולה של העתקה יצירה של תאים חדשים.
  - כן, תהיה הקצתה זיכרון. כן, ניתן לקובע חסם על כמות הזיכרון אשר יוקצה בזמן הידור (בהתהה של טיפוסות סטטיות). ב- **shallow clone**, ערך מרכיב מסוובט באופן הבא: הרמה הראשונה משובטת, ועל התאים המשובטים, יועתקו הערכיהם; ההעתקה היא גם של מצביעים, וגם של ערכים שאינם מצביעים. גודל הרמה הראשונה בטיפוסות סטטיות ידוע.
  - לא. ב- **value semantics**, אין מצביעים. במקום מצביע, ישנו יחס של הכללה. לו היו רשימת מעגליות, היינו מקבלים ערך שמכיל את עצמו.
  - לא. לא ניתן. (המקרה היחיד שבו שגיאה כזו, הוא ב- **shallow copy**). ב- **Deep copy**, כל הערך כולו משוכפל, ובשאלה אנו מניחים שככל הקצתה הזכרון מצלוחות.
  - לא. ב- **shallow clone**, ערך מרכיב מסוובט באופן הבא: הרמה הראשונה משובטת, ועל התאים המשובטים, יועתקו הערכיהם. כיון שאנו מניחים שהקצתה הזכרון לא תחזיר **null**, הרי תמיד יהיו תאים זכרון להעתיק אליהם."

#### <sup>59</sup> **אייזו סכנה קיימת ב-Shallow Copy ומדוע?** **(לא אוישר ע"י פרופ' גיל – הסיבות נשמעות בסה"כ – נדרש לשימם לב שהבעיה העיקרית (לדעתי) היא null assignment**

**תשובה:**  
 "נתחילה בלהodd מהן השיטות להעתיק אובייקט, כאשר אובייקט הוא טיפוס מרכיב המכיל שדות פרימיטיביים, שדות מורכבים, מצביעים ועוד כל מיני ישות שלמנו עליין בקורס. ישנן 3 שיטות מרכזיות להעתיק אובייקט:  
 Shallow copy - העתקה "אחד לאחד" של הערכים המשמרים באובייקט. אם שדה הוא מסוג טיפוס פרימיטיבי, הערך שלו יועתק, ואם הוא מצביע (או Reference) לכמתובת בזכרון של שדה מרכיב, המצביע יועתק.  
 Deep copy - העתקה מלאה של האובייקט וכל הבלוקים בזכרון שהוא תלוי בקיומו. כדי להעתיק שדות מורכבים או מצביעים, יוצרו עותקים גם של הבלוקים בזכרון שלהם מצביע האובייקט, והאובייקט החדש יכיבע לעותקים החדשים.  
 Lazy copy - שיטה המשלבת בין שתי השיטות הראשונות. העתקה הראשונית של האובייקט תבוצע בשיטת Shallow copy וכך שרק ערכים פרימיטיביים ומצביעים יועתקו לאובייקט החדש, אבל לכל ערך מרכיב המועתק מוצמד "מנה" שמצוין כמה אובייקטים שונים מצביעים לאותו בלוק בזכרון. לאחר מכן, במידה והתוכנית תרצה לשנות את אחד הערכים המרכיבים באובייקט, החוכנית תבודוק האם הערך המוצבע משותף עם אובייקטים נוספים ותבצע Deep copy שלו מידת הצורך כדי לא לפגוע בתלות של אובייקטים אחרים בו.  
 Shallow copy יש, אם כן, יתרון ברור של מהירות וחיסכון בזכרון, אך גם חסרון ממשמעותי של יצירת תלות מושתפת של מספר אובייקטים באותו זיכרון משותף, ללא שליטה או בקרה מה קורה לזכרון זהה במהלך ריצת התוכנית. במילים פשוטות: האובייקטים "כайлוי" מייצגים ערכים שונים בזכרון, אך שינוי של ערך מסוובט ע"י אובייקט אחד יגרור שינוי עבור שאר האובייקטים, וזה לא תמיד כוונת המתכונת".

## ניהול זיכרון אוטומטי:

נבחין במושג: "Write Barrier" – "כמota העובודה שצרכית להתבצע בכל כתיבה" – לדוגמה אם מתחזקים בכל "משתנה" זה מהיביך כתיבה נוספת בכל שניי של "רפנס". יש את שיטת **ספרית הרפנסים** שלא משמשת הרבה מחר והיא לא מתמודדת עם מקרים של מבנים מעגליים.

ישנו מגנון **איסוף הזבל (Garbage Collector)** [GC] שפועל באופן אוטומטי ומשחרר את כל הזיכרון שלא נמצא בשימוש: -

שהරו הזיכרון הופך להיות חלק מהירויות מערכת ניהול זמן הריצה של השפה, במקום לאחריות המתכנתה. -

המתכנת לא משחרר זיכרון. -

מנגנון איסוף הזבל נמצא ב- Java, Smalltalk, Python, Lisp, ML, Haskell וברוב השפות הפונקציונליות או

מבוססות עצמאיות. -

יתרונות של איסוף זבל: -

- מונע Dangling References

- מונע Memory leak

- מונע Heap Corruption

- מונעHeap de-fragmentation

- מונע compacting collector

- מונע אפשר את קיומן של פונק' כערך מרמה ראשונה.

**:Mark & Sweep** -

### Summary: mark & sweep garbage collection

### Delicate issues of the marking process

**Mark** mark all cells as unused

- Do not visit an object more than once

**Sweep** unmark all cells in use (stack, global variables), and cells which can be accessed, directly or indirectly, from these

- Do not get stuck in a loop.

**Release** all cells which remain marked

- Typical implementations:

- Breadth-first search
- Depth-first search

- Marking:

- Can be done by "raising" a bit in each object

- More efficient procedure:

- Initially, all objects are "0"
- In first collection, marking is by changing the bit to "1"
- In second collection, marking is by changing the bit to "0"
- In third collection, marking is by changing the bit to "1"
- :

### <sup>60</sup>: **תיאור האלגוריתם Mark And Sweep** -

"באלגוריתם ארבעה שלבים":

1. בתחילת נמחק כל סימן מכל קטיעי הזיכרון שהוקצנו. זאת געשה תוך שימוש במבנה נתונים בערימה שמחזיר מצביע לכל קטיע שהוקצת אי פעם.

2. בשלב הבא, הקורי שלב ה- **Mark** נתחילה מקבוצת ה- **root set**, ומכל ערך המצויה בה, נבצע חיפוש בגרף הערכים הנגושים ממנו. אם באורה ישיר ואם באורה עקי, ונסמן את כל אלו.

3. בשלב ה- **sweep** נעבור שוב על כל קטיעי הזיכרון שהוקצנו ונאסוף את כל מי שאינו מסומן.

4. בשלב האחרון, שבדרכ' כל מטבח בד בד עם ה- **sweep**, נאחד לכדי בлок אחד קטיע זיכרון חופשיים שהם רצופים בזיכרון.

השלב הראשון אינו חוני, משום שנוכל לקבוע שתגית 0 תשמש לסימון בהפעלה אחת של האלגוריתם, ותגית 1 תשמש ליזכרון בשלב העוקב. אין צורך לפיקד לבטל סימונים. גם בשלב האחרון אינו מהותי לאלגוריתם."

**: אלגוריתם איסוף זבל נוסף הוא "Copy & Stop"** -

<sup>60</sup> נלקח מהאתר – [Safot.net – קיישור](#).

- חלוקת הערימה לשני חלקים: חלק להקצאות, חלק קופא. לאחר שהחלק א' מתמלא – העתק את כל המשתנים הפעילים לאזור הקפא והחלף תפיקודים ביניהם.

איך מתגברים על הבעיה העולות ב-?Garbage Collector [GC]

- Generational GC - אסוף תחילת המשתנים "טריים". שכן יותר סביר למצוא שמשתנים שלא יהיו הרבה.
- Incremental GC - בצע קצת היישובים והזור לוזה אח"כ.
- Concurrently – Concurrent GC - רץ בו-זמנית הראשית.
- "Time constraints" – Realtime GC - נתון תחת

- Semantical garbage collection. משתנה שהותכו ניתן לא לשימוש בו יותר עד סוף ריצתה, אולם עדין היא שומרת רפנס אליו. רוב מנגנוני איסוף הזבל המתוחכמים ידעו לסכל זבל כזה.

- Escape Analysis – ניתוח שקובע אם משתנים אף פעם לא "בורחים" מהפונקציה וניתן לשחרר את הזיכרון שלהם בסיום הפונקציה.

## GC & the stack: escape analysis

- GC is always slower than stack-based memory management.
- In a pure GC, there are no automatic variables.
- In JAVA, local variables are:
  - Stack allocated builtin, atomic types: `int, double, boolean` etc. (JAVA forbids)
  - Stack allocated References to classes and arrays.
  - Heap allocated Classes and arrays (accessed only by references)

**Seemingly Innocent Program**

```
void foo() {
    int a[] = new int[1 << 20];
    List<Integer> b = new ArrayList<Integer>();
    ...
    // does a gets assigned to global variables?
}
for (int i = 0; i < 1<<20; i++)
    f(); // Lots of GC activity
```

With *escape analysis* a smart compiler can determine that variables `a` and `b` never "escape" function `foo()`, and then can be safely claimed when this function terminates.



### אלגוריתם ל-Deep Clone

- התחל מהערך הנוכחי
- עבור על כל רשות הערכים שניתן להגעה אליהם מערך זה:
  - לכל ערך שאנו מבקרים בו:
    - צין את הערך כ"ביקרנו".
    - המשך לכל הערכים שmorphים ממנו.
- הבעה? איך יודעים איזה ערכים מופנים ממנה? איך מפרשים את הביטים ככה שנדע? הרי בסך הכל אנחנו משתמשים על רצף של ביטים והערך שם יכול להיות חלק ממשו יותר גדול.
- שכפל את הרשת.

פתרון הבעיה ב- Deep Clone:  
כלומר איך אנחנו יודעים לעבור על כל הערכים או איך הערכים מחולקים?

הפתרון – Interpreting

– "הדרך" לפונק ערך כלשהו. למעשה זה ה"טיפוס" עצמו.

- בטיפוסיות סטטית – הקומפイルר יודע את ה- "Deciphering key" והוא בונה את הקוד על בסיס המידע הזה.

- בטיפוסיות דינמית – לכל ערך מצמידים את ה- "Deciphering key" וזה – "מערכת זמן ריצה" מפענחת את המפתח.

RTTI – Run Time Type Info – זה tag שמצוורף לכל ערך, שקובע את הטיפוס שלו.

- באופן יותר מדויק – זה Type Descriptor Reference ל- Type Descriptor שמשותף לכל הערכים של הטיפוס.

#### המیدע שטיפוס מספק:

- אורך ה"ערך".
- האם הוא מחולק לחלקים – ולאיזה חלקים.
- דרך מתאימה לפונCTION כל חלק.

:Statically Typed & Dynamically Typed  
 שימושים בשפות שהן Dynamically typed – RTTI והוא בדיקת שגיאות בזמן ריצה.

.Deep Cloning

.Garbage Collection

.Serialization

בשפתה שהן Dynamically typed יש שימוש נוסף ל- RTTI והוא בדיקת שגיאות בזמן ריצה.

## The challenge of deep clone

"Algorithm" for Deep Clone:

- Start from current value.
- Traverse the network of values accessible from it.
- Duplicate this network

How should we "traverse" the network?

Definition (Network Traversal: breadth- (or depth-) first search)

In Each Value we Visit:

- Mark the value as "visited"
- Proceed to all values it references

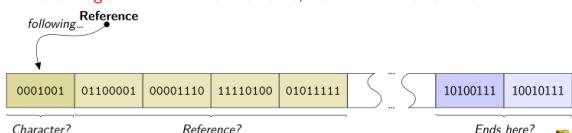
But, there is a catch...

Definition (Network Traversal: breadth- (or depth-) first search)

In Each Value we Visit:

- Mark the value as "visited"
- Proceed to all values it references

The challenge: When we reach a value, we do not what's in it!



## Summary: the "meaning" of bits and bytes

A value is represented in memory as a sequence of bits and bytes.

Components:

- Integers
- Floating point values
- Characters
- References
- Arrays
- Sets in bit mask representation.
- etc.

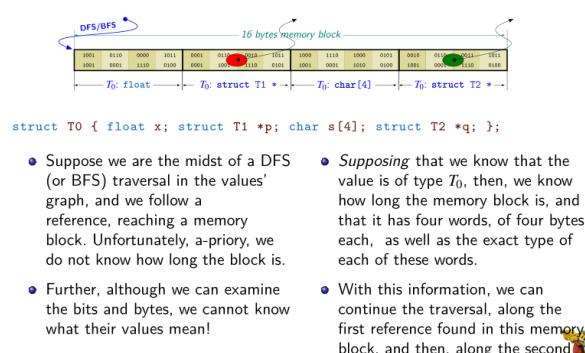
Deciphering a Value

- The values' type is the key
- It gives meaning to the bit representation.

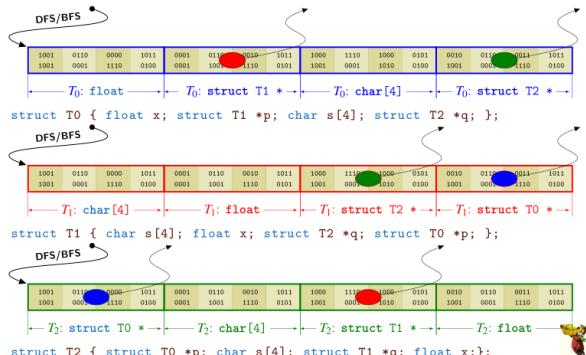
Information provided by type:

- Value's length
- Partitioning into sections
- Appropriate way of interpreting each section

## A step in a BFS/DFS tour



But, the visited block could be of any type!



## Interpreting bit and bytes as values of a type

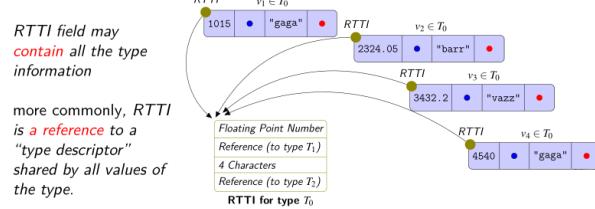
Definition (Static typing)

The compiler knows the "deciphering key", and it generates code based on this information.

Definition (Dynamic typing)

A "deciphering key" is attached to each value; the run-time system decodes the key.

The "deciphering key" is nothing but the type!



## Designing an algorithm for traversing values

Can we use static type information?

No!!!

- The network of objects typically contains values of very many distinct types
- The traversal algorithm should know
  - the type of each visited value,
  - the types of each of the values it references
- It is impractical to generate a different traversal algorithm for each input program as per the different that occur in it.

RTTI is the answer!

Definition (Run-time type information)

Run-time type information (or RTTI for short) is a tag attached to each value, which specifies its type.

Application of RTTI in different kinds of PLs:

Statically Typed

- Deep cloning,
- Garbage collection, and,
- Serialization.

Dynamically Typed

- Deep Cloning,
- Garbage Collection,
- Serialization, and,
- Run time type checks

## C, C++, & RTTI

- As a result of the "no hidden cost" language principle, C does not and cannot have RTTI.
- As a result, C cannot have general purpose GC, serialization, cloning or any deep operations.
- Due to the "C-compatibility at almost all costs" language principle, C++ does not and cannot have RTTI.
- As a result, C++ cannot have general purpose GC, serialization, cloning or any deep operations.
- C++ has a limited form of RTTI for the implementation of `virtual` functions.
- More on these mysterious "vptr" and "vtbl" in our OOP course.

## Use of RTTI in the implementation of different PLs

- Consider a variable `today` which references an object with (say) three fields: `year`, `month`, `year`
- How is `today.day=35` being implemented?

prog. lang.	C	JAVA	JAVASCRIPT
syntax	today->day=35	today.day=35	today.day=35
static typing	✓	✓	X
dynamic typing	X	✓	✓
RTTI	X	✓	✓
type punning	✓	X	X
implementation	1. dereference today 2. advance by off(day) 3. update field	1. dereference today 2. ignore RTTI 3. advance by off(day) 4. update field	1. dereference today 2. examine RTTI 3. determine off(day) 4. advance by off(day) 5. update field

## Comments on use of RTTI in PLs

- When and how is `off(day)`, the function determining the field offset, determined?
- In statically typed languages:
  - at compile time
  - from the static type of `today`.
- In dynamically typed languages:
  - at runtime
  - from the RTTI of `*today`
- In C, the `actual` type of `*today` could be `anything` (due to type punning).
- In JAVA, the `actual` type of the object that `today` refers to, can be any class that `extends` class `Date`.

שאלות:

<sup>61</sup>הסביר כיצד מתכוון שפת C++ איפשר פונקציות מקוונות על אף שאין בשפה איסוף אשפה

-

פתרונות: ב-C++ לא ניתן לפונקציה מקוונת להשתמש במסתנים שה- Storage class שלהם הוא `auto` ונמצאים בפונקציה המキיפה. ניסיון לעשות זאת יזרק את השגיאה הבאה:

error: use of local variable with automatic storage from containing function

לכן, למרות שיש פונקציות מקוונות - אין צורך במנגנון איסוף אשפה. חשוב להזכיר גם שפונקציה מקוונת הנוצרת בדרך זו, אינה יכולה גם לגשת למשתני מחלוקת שהם לא `static` של מחלוקת בהן היא מקוונת (חוון מאשר המחלוקת שבה הפונקציה מוגדרת.).

דוגמא כללית ל"קינון פונקציות ב-C++":<sup>62</sup>

```
int main() // it's int, dammit!
{
    struct X { // struct's as good as class
        static void a()
        {
        }
    };

    X::a();

    return 0;
}
```

<sup>63</sup>הסביר מדוע נדרש איסוף אשפה לשם מיימוש פונקציות מקוונות שהם ערכיים סוג א'.

(שאלה 4 חורף ב)

פתרונות: (אושר ע"י פרופ' גיל)

"כדי לקיים פונקציות מקוונות שהם ערכיים סוג א' – זה אומר שהפונקציות עדיין ייכילו את הסביבה שלהם גם כשההקשר שלהם נגמר (לדוגמא פונקציות מקוונות כאשר אנחנו רוצים להשתמש בהם מבלתי השתמש בהן בפונקציה שהיא מקוונת בתוכה) אז אנחנו צריכים לשמר את הסביבה שלהם ב- heap, בכך שלא ימתו בזמן שאנחנו עוד צריכים אותם. אבל אם אנחנו עושים את זה אז אנחנו עלולים להעמיס על heap מאוחר וקשה למתקנת לדעת בדיקת מתי אין יותר צורך בסביבה או איך ואיפה למחוק את הדברים, לכן איסוף האשפה הינו מהותי בכך לנקוט את הסביבות ששמרו על ה-.heap".

<sup>64</sup>מדוע ב- Lisp נדרש איסוף אשפה?

(מועד א' חורף 14)

התשובה שכותבה במתבון:

<sup>61</sup>נלקח מהאתר – [קישוב](#).

<sup>62</sup>נלקח מהאתר – [Stack Overflow](#) – [קישוב](#).

<sup>63</sup>נלקח מהאתר – [Safot.net](#) – [קישוב](#).

<sup>64</sup>נלקח מהאתר – [Safot.net](#) – [קישוב](#).

"משתי סיבות:

ראשית, בשפת LISP פונקציות הן עצמים סוג א', יש בשפה closure ופונקציות אוניברסליות (lambada). שנייה, על אף שלשפה יש, יש בה אופטימיזציה המבוססת על שיתוף.

הסביר נוסף (לא אוושר):

"גניחה שהגדרת פונקציה שהיא closure. הפונקציה זה תכיל מידע על כל משתני הסביבה שלה. אתה תוכל (בגלל שפונקציות הן עצמים מסוג א') להחזיר את ה-closure הזה כערך או להעביר אותו לפונקציות אחרות שיעשו בו כל מיני שימושים.

זה מחייב שכל המשתנים והערכים בסביבה של ה-closure יהיו קיימים כל עוד הפונקציה בשימוש, שכן הם יוקצו על ה-heap ולא על ה-stack. אי אפשר לקבוע בזמן קומפלילציה מהי יש לשחרר את משתני הסביבה הללו משום שלא ניתן לקבוע בזמן קומפלילציה מהיירה בזמן ריצה. שכן תהיה חיב GC שיקבע בזמן ריצה האם לשחרר את המשתנים האלה מה-heap."

<sup>65</sup> **הסביר מודיע איסוף אשפה דורש RTTI.**

פתרון (לא אוושר):

"RTTI הוא המידע שיש במשתנה המאפשר לנו לדעת איך לקרוא אותו ולמה לשיקח אותו. אחת התכונות של RTTI, כמו כן, הוא מהיכן הוא הגע, כלומר למי יש הקשר אליו.

בשים לב שאחד החלקים החשובים באיסוף אשפה היא לא רק למצוא את כל המשתנים המוקצים, אלא גם את כל מה שהמשתנים המוקצים מחשרים אליהם. וללא ה-RTTI שיגיד לנו מה נמצא בתאים (ובכך יאפשר לנו לדעת איך להגיע לתאים האחרים במקרה שיש) לא יוכל לדעת אילו תאים הם אינם אשפה ולא יוכל בעצם לאסוף אשפה."

## 5. תכנות אימפרטיבי

(פרק 6+-commands.pdf)

### ביטוי (Expression)

- מייצרת ערך.
- לא משנה את מצב התכנית.
- ביטוי אטומי (Atomic Expression) :

  - מה שמכיל כל דבר שלא מכיל בתוכו עוד expressions, לדוגמה ליטרלים.
  - דברים כמו ערכים, שיכולים להניב רק ערך אחד.
  - בשפות שונות הם מוגדרים קצר אחרת,
  - אבל מה שרשום למעלה זה בגודל.

### פקודה (Command) – חלק מתכנית מחשב אשר:

- לא מייצרת ערך.
- משנה את מצב התכנית.
- דוגמאות:

  - I/O – קריאה, כתיבה.
  - השמות&הצבות.
  - Return ○ לולאות.
  - התניות.
  - Skip – לא עושה דבר.

נזכיר ש אין \*פקודות\* בשפות פונקציונליות טהורות.

Command Constructors –

לא מזורה ערך – בנויגוד לפונקציה נניה, שיכולה להחזיר ערך.

אלו למעלה הן ההגדרות היבשות. בפועל, בשפות בהן יש האחזקה של פקודות וביטויים – פקודה יכולה להחזיר ערך.

If הוא לא – Statement – הוא מצפה לערך.

**בפועל ההבדל בין ביטוי ופקודה הוא לא כל כך ברור:**

### Expressions changing the program's state?

#### Nasty CS101 Exam Question

You are given a seemingly innocent PASCAL code, and asked...

```
Procedure Hamlet;
VAR
    happy: Boolean;

    Function toBe:Boolean;
    Begin
        happy := not happy;
        toBe := happy
    End;
Begin
    happy := false;
    If toBe and not toBe
        WriteLn("The Answer!");
End;
```

Could "The Answer" ever be written?

- Suppose that **toBe** is a function nested in procedure **Hamlet**,
- which may have access to a global variable,
- whose initial value is **false**,
- In fact, function **toBe** returns the value of this global variable,
- just after flipping it!
- So, the answer is,...

Yes!

לברר מה בדיק קורה בשקופית חזאת ואיך ה-"ביטוי" משנה את מצב התכנית? איזה ביטוי משנה את מצב התכנית?

**ביטוי פקודה** (Expression Command) – פקודות שבנוספ' לשינוי מצב התכנית, הן גם מזירות ערך. הדברים האמורים לא רק בפקודות אטומיות, אלא גם בפקודות תנאי, פקודות לולאה, קריאה לפרוצדורה, פקודה ריקה, וכו'.

זה רעיון אידיאליסטי שבן:

- כל ביטוי יכול להיות מוחלף בפקודה.
- כל פקודה היא ביטוי, כך שכל פקודה מזירה ערך.

הרחהה של פרופ' גיל<sup>66</sup>

"בתחילה היה ניסיון להפריד בין "ביטויים" ובין "פקודות". אנו רואים ניסיון כזה בפסקל, ובעוד שפות (אייפל למשל). התפיסה הייתה שפקודות נועדות לשנות את מצב הוכנית, מובילו ליציר ערך, וביטויים מחשבים ערך,ambil לשנות את מצב הוכנית. אלא שאם פקודות יכולות לקרוא לפונקציות, ובפונקציות אנחנו מרשימים פקודות (آن סיבה שלא להתר זאת) הרימושגים מתערבים. כך המצב ב-C, JAVA ובסופות רבות אחרות. קרייה לפונקציה יכולה להיות פקודה, ופונקציה יכולה להחזיר את הטיפוס UNIT שאומר בעצם שהיא לא מחזירה ערך."

בכל זאת, יותר הבדל אחד בין פקודות וביטויים. פקודות מאפשר להרכיב באמצעות שלושה בנאים ראשיים (בדוק WHETHER יודע מה המ), וגם אם פקודות אוטומיות יכולות להחזיר ערך, לא ברור מהו הערך שפקודות מורכבות מוחזירות.

המונה COMMAND-EXPRESSION יוחד לבניינים בשפות תכנות מסוימות שבהם כל פקודה מחזירה ערך. איחוד המושגים יוצר קשיים מסוימים, ואין הרבה דוגמאות מוצלחות לCOMMAND-EXPRESSION.

קבוצת הביטויים בשפה מוגדרים רקורסיבית:		קבוצת הפקודות בשפה גם כן מוגדרת רקורסיבית:
<b>:Atomic Expressions</b> <ul style="list-style-type: none"> <li>- הביטויים האוטומטיים –           <ul style="list-style-type: none"> <li>○ Literals</li> <li>○ Variable Inspection – בדיקת ערכו של משתנה.</li> </ul> </li> <li>- בניי הביטויים –           <ul style="list-style-type: none"> <li>○ אופרטורים כגון "+", "-", "..."</li> <li>○ קריאות לפונקציה:</li> </ul> </li> </ul>		<ul style="list-style-type: none"> <li>החולקה לפקודות אוטומטיות ובנאי פקודות היא שונה עברו כל שפת תכנות. המבנה הכללי זהה אבל יש מבחן עצום:</li> <li>- <b>פקודות אוטומטיות –</b> <ul style="list-style-type: none"> <li>○ הפקודה הריקה.</li> <li>○ השמה.</li> <li>○ רצפים "sequencers" – נדבר עליהם בשלב מאוחר יותר.</li> </ul> </li> </ul>
<b>Function call expression constructor</b> <div style="background-color: #ffffcc; padding: 10px;"> <p>Definition (Function call expression constructor [dynamic typing version])</p> <p>If <math>f</math> is a function taking <math>n \geq 0</math> arguments, and <math>E_1, \dots, E_n</math> are expressions, then</p> <math display="block">f(E_1, \dots, E_n)</math> <p>is an expression.</p> </div> <div style="background-color: #ffffcc; padding: 10px; margin-top: 10px;"> <p>Definition (Function call expression constructor [static typing version])</p> <p>Let <math>f</math> be a (typed) function of <math>n \geq 0</math> arguments,</p> <math display="block">f \in \tau_1 \times \dots \times \tau_n \rightarrow \tau.</math> <p>Let <math>E_1, \dots, E_n</math> be expressions of types <math>\tau_1, \dots, \tau_n</math>. Then,</p> <math display="block">f(E_1, \dots, E_n)</math> <p>is an expression of type <math>\tau</math>.</p> </div>	<b>:Command Constructor</b> <ul style="list-style-type: none"> <li>- בניי בлок הפקודות.</li> <li>○ בנאים של פקודות מותגנות.</li> <li>○ בניי של פקודות איטרטיביות.</li> <li>○ בניי הפקודות try... catch ... finally .with .</li> <li>○ בניי הפקודה .with .</li> </ul>	
<ul style="list-style-type: none"> <li>- קבוצת כל הביטויים האוטומטיים והבנאים שלהם תלויים בשפה, אך באופן כללי לא מגוונים יותר מדי.</li> </ul>		

בשפת C++ ישנו שני סוגי של פקודות אוטומטיות ( Atomic Command )		בפסקל נמצא 3 פקודות אוטומטיות:
(נתעלם מ- ) Sequencers	(נתעלם מ- ) Sequencers	(נתעלם מ- goto ה- sequencer היחיד בשפה)
<ul style="list-style-type: none"> <li>- <b>הפקודה הריקה.</b> השפה היא ספרטיסטית מוקלה – כך שאם בסוף מוטפים נקודה פסיק הוא לא חלק מהפקודה</li> <li>○ אין שינוי במצב הוכנית.</li> <li>○ אין חישובים.</li> <li>○ הייצוג הטקסטואלי הוא ";".</li> </ul>		

<sup>66</sup> נלקח מהאתר Safot.net – קישור.

<p><i>Empty command; no need for loop "body"; all work is carried out by the side-effects of the expression used in the loop condition</i></p> <pre>while (*s++ = *t++) ;</pre> <p><b>ביטוי המסומן כפקודה:</b></p> <p>פקודה אוטומטית היא גם "ביטוי שהוא נקודה פסיק":</p> <p><i>In C, assignment is an operator taking two arguments L (left) and R (right). The operator returns R, and as side-effect, assigns R into L.</i></p> <pre>0; i*1; i; (i=i)==i; i++ + ++i;</pre>	<p>– ומתחבאת שם פקודה ריקה.</p> <ul style="list-style-type: none"> <li>○ אין שניי ב"מצב"</li> <li>○ הוכנית.</li> <li>○ אין חישובים.</li> <li>○ הקיום תלוי בkontekst בלבד.</li> </ul> <p><b>השמה.</b></p> <p><b>קריאה לפונק'.</b></p>
--	--

## עברית שפה:C

### Command expressions in C

Definition (Command expressions in C)

If  $E$  is an expression, then  $E;$  is a command.

Table 6.2.1: C program elements

Element	Purpose	Example
<i>Expression</i>	evaluates to a value	$f() ? a + b : a - b$
<i>Command</i>	change program state (even vacuously)	$f() ? a + b : a - b;$ $i = 0;$
<i>Variable definition</i>	creates a variable and binds a name to it	$int i;$
<i>Variable declaration</i>	makes a binding; variable must be created elsewhere	$extern int i;$
<i>Definition &amp; initializer</i>	creates a variable, binds a name to it, and initializes it	$int i = 3;$
<i>Declaration &amp; initializer</i>	X	X

## More on atomic expressions in C

All C's atomic commands (including sequencers) are semicolon terminated

- Not every command includes a semicolon
- Not every semicolon is part of a command

Can you locate the atomic command(s) in this code?

```
struct Complex {  
    double x, y;  
};  
;  
main() {  
    int i, a[100];  
    for (i = 0; i < 100; i++)  
        a[i] = 100;  
    return 0;  
}
```

- במשך יש את "ניתוח הפקודות בשפת C" שנרשם לאחר ההסבר על Sequencers – יש שם הסבר חשוב על הנושא!

## Two kinds of atomic commands in JAVA<sup>5</sup>

Just as in C,

; the empty command is a lonely semicolon;

*Expression*: provided that the first step in the recursive decomposition of expression is "something" that has (might have) side-effects:

- Function call
- Operator with side effects:

Assignment e.g., =, +=, <<=, ...

Increment/decrement ++ and --; either prefix or postfix.

Object creation e.g., new Object()

- Nothing else!

---

<sup>5</sup>ignoring sequencers

פקודות אטומיות – פקודות הצבה השונות:

- הצבה רגילה ( Vanilla Assignment ) -

$v \leftarrow e$

e משוערך.

הערך המשוערך מוכנס למשתנה – v.

הצבה מרובה ( Multiple Assignment ) – צורה של השמה הרגילה -

$v_1, v_2, \dots, v_n \leftarrow e$

e משוערך.

הערך המשוערך מוכנס למשתנים:  $v_1, v_2, \dots, v_n$ .

עדכון ( Update ) – צורה של השמה הרגילה -

$v \leftarrow \varphi(\cdot, e_1, e_2, \dots, e_n)$

$v \leftarrow \varphi(v, e_1, e_2, \dots, e_n)$

לדוגמא בשפת C, הפקודות:

i++;

i\*=2;

הצבה מקבילה ( Collateral Assignment ) -

Syntactic sugar for:

$v_1, v_2 \leftarrow e_1, e_2$

- $e_1$  מהושב ומוכנס למשנה  $v_1$
- $e_2$  מהושב ומוכנס למשנה  $v_2$
- שתי הפעולות מתבצעות ב- *Collateral* – במקביל.
- לא ניתן להשתמש בסוג הצבה זה על מנת להחליף ערכים של משתנים.
- תיאורית אפשרי, אבל לא שימושי במיוחד.
- **הצבה סימולטנית ( Simultaneous Assignment )**

$(v_1, v_2) \leftarrow (e_1, e_2)$

- $e_1$  מהושב ומוכנס למשנה  $v_1$ , בדיק כמו *Collateral Assignment*
- $e_2$  מהושב ומוכנס למשנה  $v_2$ , בדיק כמו *Collateral Assignment*
- שתי הפעולות קוררות במקביל.
- ניתן להשתמש בסוג הצבה זה על מנת להחליף ערכים של משתנים (בניגוד ל- *Collateral Assignment*).
- ניתן להסתכל על הצבה זו כהצבה של tuples.

שאלה שקשורה לנושא:

- 67 מה ההבדל בין פקודה לפונקציה?

פתרונות: (אושר ע"י פרופ' גיל)

"פקודה משנה את מצב התוכנית ולא מזירה ערך".

פונקציה היא רצף של פקודות וביטויים העשויה לקבל פרמטרים. אפשר לבדוק בין פונקציה אונומית, ובין פונקציה שיש לה שם.

אליה הן ההבדלות היבשות.

בפועל – פקודה יכולה להחזיר ערך, בשפות שבן הייתה האחדה של ביטויים ופקודות. זה המצב למשל ב BCPL ו- ICON.

בשפות פונקציונליות טהורות, פונקציה אינה יכולה לשנות את מצב התוכנית. בשפות אלו למעשה אין מצב לתוכנית. בשפות אימפרטיביות, פונקציה עשויה לשנות את מצב התוכנית, וכיולה אף שלא להחזיר ערך (למען הדיווק – להחזיר טיפוס UNIT שיש לו ערך אחד בלבד).

#### מספר הגדרות:

:Skip Command

NOP, \relax, ";" ...

:Block Commands (מוסבר בהמשך)

.Sequential block constructor

.Collateral block constructor

.Concurrent block constructor

- 68 **Programmatically Identical** אומר שלא משנה באיזה סדר נעשה את רצף הפקודות (לדוגמא בחיבור מספרים – אם נחבר 7 קודם ואו 4 או 4 קודם ואו 7) – אנחנו מקבלים אותה התוצאה בסוף הרצף. لكن, קטעי קוד SMBצעים אותן פקודות בסדר אחר ומניבים אותה התוצאה הם זרים תכנותית. בגדול – לא משנה סדר הפעולות, מקבלים אותה התוצאה.

- 69 **Semantically equivalent**

אומרסדר אחר של פעולות עלול להוביל לתוצאה שונה. למשל, יכולות להיות שתי תכניות שמכילות את אותן הפקודות – אבל עברו שני סידורים אחרים של הפקודות האלה – מקבלים אותה שונה. אותו התוכן – יכול להוביל לתוצאה שונה.

<sup>67</sup> נלקח מהאתר Safot.net – קישור.

<sup>68</sup> פיסבוק – מאושר ע"י פרופ' גיל Ayelet Kravi

<sup>69</sup> פיסבוק – מאושר ע"י פרופ' גיל Ayelet Kravi

## Sequential block constructor

### Definition (Sequential block constructor)

If  $C_1, \dots, C_n$  are commands,  $n \geq 0$ , then

$$\{C_1; C_2; \dots; C_n\} \quad (4.1)$$

is a composite command, whose semantics is **sequential**:  $C_{i+1}$  is executed after  $C_i$  terminates.

- Most common constructor
- Makes it possible to group several commands, and use them as one, e.g., inside a conditional
- If your language has no skip command, you can use the empty sequence, {}.

**Separatist Approach:** semicolon separates commands; used in PASCAL; mathematically clean; error-prone.  
**Terminist Approach:** semicolon terminates commands (at least atomic commands);

## Collateral block constructor

### Definition (Collateral block constructor)

If  $C_1, \dots, C_n$  are commands,  $n \geq 0$ , then

$$\{C_1 \sim C_2 \sim \dots \sim C_n\} \quad (4.2)$$

is a composite command, whose semantics is that  $C_1, \dots, C_n$  are executed **collaterally**.

Very rare, yet (as we shall see) important

- Order of execution is **non-deterministic**
- An optimizing compiler (or even the runtime system) can choose "best" order
- Good use of this constructor, requires the programmer to design  $C_1, \dots, C_n$  such that, no matter what, the result is
  - programmatically identical, or
  - at least, semantically equivalent

## Programmatically identical vs. semantically equivalent

### Programmatically Identical

Now these are the generations of the sons of Noah, Shem, Ham, and Japheth: and unto them were sons born after the flood.

- ① The sons of Japheth; Gomer, and Magog, and Madai, and Javan, and Tubal, and Meshech, and Tiras...
- ② And the sons of Ham; Cush, and Mizraim, and Phut, and Canaan
- ③ The children of Shem; Elam, and Asshur, and Arphaxad, and Lud, and Aram

```
{
  grandsons += 7;
  ~
  grandsons += 4;
  ~
  grandsons += 5;
}
```

### Semantically Equivalent

```
{
  humanity.add("Adam");
  ~
  humanity.add("Eve");
}
```

At the end, both "Adam" and "Eve" will belong to **humanity**; but the internals of the **humanity** data structure might be different.

## Concurrent block constructor

### Definition (Concurrent block constructor)

If  $C_1, \dots, C_n$  are commands,  $n \geq 0$ , then

$$\{C_1 | C_2 | \dots | C_n\} \quad (4.3)$$

is a composite command, whose semantics is that  $C_1, \dots, C_n$  are executed **concurrently**.

Common in concurrent PLs, e.g., OCCAM

- Just like "collateral"...
- Commands can be executed in any order; order of execution is non-deterministic
- An optimizing compiler (or even the runtime system) can choose "best" order
- Good use of this constructor, requires the programmer to design  $C_1, \dots, C_n$ ; such that, no matter what, the result is, programmatically identical, or semantically equivalent



## Collateral vs. concurrent collateral

**Collateral** really means "not guaranteed to be sequential", or "undefined"; PL chooses the extent of defining this "undefined", e.g.,

"the order of evaluation of  $a$  and  $b$  in  $a+b$  is unspecified. Also, the runtime behavior is undefined in the case  $a$  and  $b$  access the same memory".

**Concurrent** may be executed in parallel, which is an extent of definition of a **collateral execution**.

"the evaluation of  $a+b$  by executing  $a$  and  $b$  concurrently; as usual, this concurrent execution is *fair* and *synchronous*, which means that...".

:Conditional Commands

## Conditional commands

Definition (Conditional command constructor)

If  $C_1, \dots, C_n$  are commands,  $n \geq 1$ , and  $E_1, \dots, E_n$  are boolean expressions, then

$\{E_1?C_1 : E_2?C_2 : \dots : E_n?C_n\}$  (5.1)  
is a conditional command.

## Semantics of conditional commands

$\{E_1?C_1 : E_2?C_2 : \dots : E_n?C_n\}$

Can be:

**Sequential:** Evaluate  $E_1$ , if true, then execute  $C_1$ , otherwise, recursively execute the rest, i.e.,  $\{E_2?C_2 : \dots : E_n?C_n\}$ .

**Collateral:** Evaluate  $E_1, E_2, \dots, E_n$  collaterally. If there exists  $i$  for which  $E_i$  evaluates to true, then execute  $C_i$ . If there exists more than one such  $i$ , arbitrarily choose one of them.

**Concurrent:** Same as collateral, except that if certain  $E_i$  are slow to execute, or blocked, the particular concurrency regime, prescribes running the others.

Example of a concurrency regime:

Strong fairness:

In any infinite run, there is no process which does not execute infinitely many times.

## :Semantics of conditional commands

- נבצע הורכה של הביטוי הראשון. אם הוא אמת, נבצע את ה- Command ההפוך לו. אחרת, נמשיך.
- דומה רק שהשערוך מתבצע ללא הגדרת סדר. אנחנו לא יודעים באיזה סדר הערכים משוערכים.
- בדוק כמו Collateral עם הבדל אחד – דברים עלולים להתבצע במקביל – מקרה מיוחד של Concurrent

שאלה שקשורה לנושא:

- <sup>70</sup> הסבר מה ההבדל בין לולאות Definite וlolalot Indefinite. הסבר גם מדוע סמנטיקה קולטරלית לא יכולה להיות רק בלולאות שהן definite.

תשובה:

"lolalot definite היא lolala שנייה לדעת טרם תחילת הביצוע של lolala כמה פעמים היא מתבצעת. יתכן שבבולות lolalot ייחשבו באופן דינמי ובזמן ריצה, אבל בכל זאת נדרש שלפני ביצוע lolala בהרואה, מספר האיטרציות ידוע. ישנו שפota שמאגדירות lolalot כאלו מחלק מהגדרת השפה. הדוגמא הבולת היא פסקל. ב- C אין הגדרה כזו כחלק מהשפה. שמו לב של lolalot שמתייחס בתוכן break לא יכולות להיות definite."

לעומת זאת, lolala indefinite היא lolala שבה לא ניתן לדעת מראש כמה פעמים היא תתבצע. אחרי ביצוע כל איטרציה, יש לבדוק את תנאי הסיום.

lolala קולטראלית היא כזו שאין הגדרה של סדר הביצוע של האיטרציות. lolala肯定性 definite יכולה להיות מוגדר, יוכל להיות לא מוגדר, שהוא קולטראלי. שמו לב של lolalot שמתייחס בתוכן continue יכולות עקרונות להיות קולטראלית.

[הסבר<sup>71</sup>: אם lolala מתיירה break או אי אפשר להזות כמה פעמים היא תתבצע ואז כמובן שהיא לא definite, או שתחת תנאים מסוימים תתבצע מספר שונה של פעמים. lolala שמתייחס בתוכן continue יכולות להיות מכיוון ש- continue לבדו אינו פגע במספר האיטרציות של lolala.]

לא יתכן lolala שהיא indefinite תהיה קולטראלית, שכן אם נבצע את האיטרציות בסדר כלשהו, הרי יתכן שנבוצע איטרציה מסוימת, ולאחר הביצוע שללה, ביצוע של איטרציה אחרת, נגלה שלא היה צריך לבצע את האיטרציה הרואה.

ההגדרה חשובה, משום שבזכותה, אפילו מהדר שאינו חכם, יכול לבצע אופטימיזציה lolala מסוימת שモפעה בתכנית. בכל זאת, מהדר חכם יכול להזות lolala מסוימת הינה definite ולבצע אופטימיזציה מתאימה. יתרה מכך, מהדר חכם יכול להזות lolala מסוימת בתכנית יכולה להתבצע באופן קולטראלי.

<sup>70</sup> נלקח מבוחן חורף 2012 מועד א'.

<sup>71</sup> נלקח מהאתר Safot.net – קישוט.

## The "else" variants

Definition (Conditional command constructor with else clause)

If  $C_1, \dots, C_n, C_{n+1}$  are commands,  $n \geq 1$ , and  $E_1, \dots, E_n$  are boolean expressions, then  
 $\{E_1?C_1 : E_2?C_2 : \dots : E_n?C_n : C_{n+1}\} \quad (5.2)$   
is a conditional command, whose semantics is the precisely the same as the familiar  
 $\{E_1?C_1 : E_2?C_2 : \dots : E_n?C_n\},$   
where we define  
 $E_n = \neg E_1 \wedge \neg E_2 \wedge \dots \wedge \neg E_{n-1} \quad (5.3)$

The "else" clause is sometimes denoted by:

- default
- otherwise

## Variant #2 + #3 / many: if-then-else & cases

- Special construct for the case  $n = 1$  in the form of  
`if Condition then Statement  
[ else Statement ]`  
*your syntax may vary*
- Special construct for the case that
  - each of  $E_i$  is in the form  $e = c_i$
  - $e$  is an expression (usually integral), common to all  $i = 1, 2, \dots$
  - $c_i$  is a distinct constant expression for all  $i = 1, 2, \dots$`case Expression of  
{ constantExpression Statement }  
[ otherwise Statement ]`  
*your syntax may vary*

:Conditional command constructor with else clause

## Variant #1 / many: the "else" clause

Almost all languages use "else"  
`If thouWiltTakeTheLeftHand  
then  
  iWillGoToTheRight  
else  
  iWillGoToTheLeft`  
PASCAL uses "Otherwise"  
`case expression of  
  Selector: Statement;  
  ...  
  Selector: Statement  
  otherwise  
    Statement;  
  Statement  
end`  
(the Gnu-PASCAL's EBNF)

C uses "default"

```
int isPrime(unsigned c) {  
    switch (c) {  
        case 0:  
        case 1: return 0;  
        case 2: return 1;  
        default:  
            return isPrime(c);  
    }  
}
```

## Why special switch/case statement?

Because the PL designer thought...

- it would be used often
- it has efficient implementation on "wide-spread" machines
  - Dedicated hardware instruction in some architecture
  - Jump-table implementation
  - Binary search implementation

The above two reasons, with different weights, explain many features of PL.

*these are precisely the reasons for the particular specification of conditional in the form of "if-then-else" for the cases  $n = 1$*

## Cases variants?

- Range of consecutive integer values (in PASCAL)
- Cases of string expression
  - No straightforward efficient implementation
  - Added in later versions of JAVA after overwhelming programmers' demand
- Regular expressions in selectors
  - Exists in BASH
  - Seems natural for the problem domain
- General patterns in selectors
  - Exists in ML and other functional PLs
  - In the spirit of the PL type system
- No cases statement
  - In EIFFEL a pure OO language
  - Language designer thought it encourages non OO mindset

## Vanilla multi-way conditional?

- Exists in many languages, in the form of a special keyword
- `elseif`, or `elsif` or `ELIF`,
- e.g., in PHP you can write

```
"elseif" in PHP
if ($a > $b) {
    echo "$a_is_bigger_than_b";
} elseif ($a == $b) {
    echo "$a_is_equal_to_b";
} else {
    echo "$a_is_smaller_than_b";
}
```

## `else if? elseif?` what's the big difference?

- There is no big difference!
- `else if` many levels of nesting  
`elseif` one nesting level
- this might have an effect on automatic indentation, but modern code formatters are typically smarter than that!
- another small difference occurs if the PL requires the `else` part to be wrapped within "{" and "}".

: "Iterative command constructor", "State generator"

## Iterative command constructor

A very general pattern of iterative command constructor

### Definition (Iterative command constructor)

If  $S$  is a "program state generator" and  $C$  is a command, then

$$\text{forall } S \text{ do } C$$

is an iterative composite command whose semantics is the (sequential / collateral / concurrent) execution of  $C$  in all program states that  $S$  generates.

Note that with "sequencers" such as `break` and `continue`, iterative commands can be even richer!

## State generator?

The state generator  $S$  may be...

Range of integer (ordinal) values, e.g.,

```
For i := gcd(a,b) to lcm(a,b) do
  If isPrime(i) then
    Writeln(i);
```

Any arithmetical progression, e.g., in FORTRAN

```
Comment WHAT IS BEING COMPUTED???
      INTEGER SQUARE11
      SQUARE11=0
      DO 1000 I = 1, 22, 2
      SQUARE11 = SQUARE11 + I
1000  CONTINUE
```

### Expression, typically boolean:

- expression is re-evaluated in face of the state changes made by the command  $C$ ;
- iteration continues until expression becomes true, or,
- until expression becomes false,

### Generator, e.g., in JAVA

```
List<Thing> things = new ArrayList<Thing>();
for (Thing t : things)
    System.out.println(t);
```

### Cells in an array, e.g., in JAVA

```
public static void main(String[] args) {
    int i = 0;
    for (String arg: args)
        System.out.println(
            "Argument " +
            ++i +
            ": " +
            arg
        );
}
```

## Subtleties of the iteration variable

### ① the value of the expression(s) defining the range/arithmetical progression change during iteration...

The iteration range, as well as the step value are computed *only* at the beginning of the loop. (Check the FORTRAN/PASCAL manual if you are not convinced)

### ② the loop's body tries to change this variable...

The loop body should not change the iteration variable; The PL could either issue a compile-time error message (PASCAL), runtime error message (JAVA), or just state that program behavior is undefined.

### ③ What's the value of the iteration variable after the loop?

The iteration variable may not even exist after the loop (JAVA); or, its value may be undefined (PASCAL).

- the PL designer thought that programmers should not use the iteration variable after the loop ends
- if the value is defined, then collateral implementation is more difficult
- many architectures provide a specialized CPU instructions for iterations;
- the final value of the iteration variable with these instructions is not always the same

הן פקודות אוטומטיות שהפעלה שלן משנה את רצף התוכנית.

## דוגמאות:

- goto – מנקודה אחת בתכנית לאחורה.
- return – לסוף הפונקציה בו נמצאתה.
- break – יציאה מהאיטרציה הנוכחית שהוא נמצא בה.
- continue – עברו לראש האיטרציה הנוכחית.
- throw – הריגה שמעבירה את השיליטה ל- handler של הפונקציה שקרה.

הוא ישות שמצוינת פקודה ריקה בתכנית, בד"כ יש פקודות לא ריקות לפני ואחרי הפקודה הריקה שה- label מציין.

### Structured programming

...is a programming paradigm, characterized by

- "Three Controls": precisely three ways for marshaling control:
  - ➊ Sequence, e.g., `begin C1;C2...;Cn end` for  $n \geq 0$
  - ➋ Selection, e.g., `if ... then ... elseif ... else ... endif`.
  - ➌ Iteration, e.g., `while ... do ... done`
- Structured Control:
  - all control commands are in fact, command constructors.
  - control is marshalled through the program structure.

Theorem (The structured programming theorem (Böhm-Jacopini, 1966))

*Every flowchart graph G, can be converted into an equivalent structured program, P(G).*

### What are sequencers?

#### Definition (Sequencers)

*Sequencers are atomic commands whose execution alters the "normal" (structural) flow of control.*

Examples:

- `goto` from any program point to another
- `return` to the end of an enclosing function
- `break` out of an enclosing iteration
- `continue` to the head of an enclosing iteration
- `throw` exception, that transfers control to a handler in an invoking function

2

### Labels

To denote where `goto` will go to, one needs a *label*

#### Definition (Label)

A *label* an entity which denotes an empty command in the program text; typically there are non-empty commands before and after the empty command that the label denotes

Labels are a *deliberately disprivileged* entities.

### Label "literal" & first class labels

#### Literal labels

The label itself "is" the empty command which it denotes:

- *Identifiers*, as in C and assembly PLs
- *Integers*, as in PASCAL, BASIC and FORTRAN  
In BASIC, all commands must be labeled in a strictly ascending order

#### First class labels

BASIC, PL/I, and some other obscure languages, treat labels as first class values, which can be
 

- stored in variables
- passed as arguments,
- returned by functions, etc.

```
11: label v = a > b?           11: 12;
:                                :
12: goto v;
```

### Declared vs. ad-hoc labels

Declared label as in PASCAL; labels must be declared before they are used

Ad-hoc labels as in C/C++

```
int main() {
    http://www.cs.technion.ac.il/yogi;
    printf("Page_loaded_successfully\n");
}
```

### Ad-hoc labels may generate subtle bugs

```
int isPerfect(unsigned n) {
    switch (n) {
        default:
            return 0;
        case 6:
        case 28:
        case 496:
        case 8128:
        case 0x1FFF000u:
    }
    return 1;
}
```

spelling error in "default"

3

5

4

6

:Escape, Goto, Continue

## The persecuted goto

Restrictions on `goto`...

- Only within a block structure. FORTRAN,
- Only `goto` within a function.
- C does not allow inter-functional `goto`, but `gotos` are allowed in and out of a block.
- No `goto` from a bracketed command into itself. PASCAL
- No `goto` into a loop or into a conditional. C
- No `goto` into a compound command. PASCAL
- No `goto` into a nested function. PASCAL and ALGOL
- No `goto` at all. JAVA!

## Goto to a nesting function

Labels obey scope rules:

- If a variable of a nesting function is recognized in a nested function, the nested function can also `goto` to a label defined the nesting function.
- In case of recursion, labels denote a program point in the current activation.

## Escapes

### Definition (Escape)

An escape is a special kind of `goto` which terminates the execution of a compound command in which it is nested

Makes single entry, multiple exit commands:

- `exit` in ADA
- `break` in C/C++/JAVA

Useful for simplifying nesting structure

## Varieties of escape

- Escape any enclosing Loop. `exit l` in ADA and `break l` in PERL/JAVA, where  $\ell$  is a label of an enclosing loop
- Escaping out of a Function. `return` in C and FORTRAN
- Terminal escape. terminate the execution of the whole program; `halt` in FORTRAN.
- Specialized escape. `break` out of a `switch` command in C

## Continue

### Definition (Continue)

A continue is a special kind of an escape which can only occur within a iteration command; it terminates the execution of the current iteration and if there is a next iteration, it proceeds to it.

Just like `break`, useful for simplifying nesting structure.

- Continue any Enclosing Loop. `continue l` in JAVA, where  $\ell$  is a label of an enclosing iteration command, proceeds to the next iteration of the iteration command marked by  $\ell$ .
- Cannot be emulated by PASCAL `goto`, due to restrictions that PASCAL places on `goto`.

שאלות הקשורות לנושא:

- 72 ניתוח מבנה הפקודות בשפת C.

1. רשום את כל סוגי הפקודות האוטומיות בשפת C, בלבד מ `sequencers`. אל תצרכ הסבירים.

2. רשום את כל בניין הפקודות שבספת C. אל תצרכ הסבירים.

3. האם בשפת C יש ?multiple assignment? נמק.

4. מהם ?sequencers?

5. רשום את כל ה `sequencers` של שפת C. אל תצרכ הסבירים.

מקורו: חורף תשע"ה, מועד ב'.

פתרונות של פרופ' גיל:

"1. בלבד מ `sequencers`, בשפת C יש שני סוגים של פקודות אוטומיות:

o הפקודה הריקה, המסמנת על ידי סימן הנקודה ופסיק, ":".

o והפקודה הנוצרת מביטוי שאחריו מופיע סימן הנקודה ופסיק, ":".

טענות בדבר קיום פקודת הצבה, השמה, קרייה לפונקציה בדרך כלל מעידות על חוסר הבנה.

2. ישנו שלושה סוגים לבניאים דרישים - בניין התנאי, ובנאי הלולאה.

א. בניין השרשור מופיע ב C כזוג של סוגרים מסוולים, {...}.

ב. יש שני בניין תנאי בשפת C: תנאי if ופיזול רב ראשי switch. ואירועים שונים של שני בניאים אלו מסומנים על ידי תוספת מילים שמורות: else ו break case.

ג. יש שלושה בניין לולאה בשפת C:

for

while

do ... while

3. הפקודה\_multiple assignment\_ היא הפקודה多重 assignment. היא מזינה לשלוחה את הערך החדש של המשתנה אליו הוצב הערך. ולכן לכארה אין \_multiple assignment\_, אבל יש אופרטור הצבה, שהאסמונטטיביות שלו היא מימין לשמאלי (הפוכה מזו של רוב האופרטורים), ואשר מזהיר את הערך החדש של המשתנה אליו הוצב הערך.

שימוש אידiomטי באופרטור זה מאפשר להציב את אותו ערך למספר משתנים. משמעות הביטוי:

$$a = b = c$$

היא:

- |   |  |
|---|--|
| 1 | שערך של הביטוי <b>c</b>  |
| 2 | הצבה של ערך הביטוי <b>זה</b> ל <b>LVALUE</b> משמאלו, כלומר ל <b>b</b>            |
| 3 | הישוב של הערך של <b>b</b> לאחר ההצבה (לכל צורך מעשי, זה בדיקת הערך של <b>c</b> ) |
| 4 | הצבת ערך <b>זה</b> ל <b>LVALUE</b> משמאלו, כלומר ל <b>a</b>                      |

ביטוי זה הוא על כן בפועל **multiple assignment**.

4. Sequencers הם פקודות אוטומיות, אשר ביצוען עומד בסתייה לבניות של התחנית, כלומר שבירה של ה塊. המבנית של ביצוע סדרתי, תנאי, ולולאה, המיצג בדיאגרמות נשיא שנידרמן, ולפיו לכל בлок יש נקודת כניסה אחת ונקודת יציאה אחת. יש **sequencers** שמייצרים נקודת יציאה לא מבנית לבlok, כמו למשל **break**, **goto** לעומת זאת, מאפשר גם יותר מכניתה אחת לבlok.

הטענה "משנים את הביצוע הסדרתי", אינה מבהירה בצורה מלאה את התמונה המלאה.

ה- 5. Sequencers בשפת C הם:

```
goto  
break  
continue  
return
```

הערות :

- שימוש ב **break** בתוך פקודה **switch**, או **sequencer**, אינו **sequencer**. למעשה אי השימוש בו הוא חיוני,
- לא כל שימוש ב **return** הוא **sequencer**; השימוש ב **return** בסוף פונקציה, כדי להחזיר את הערך, הוא חיוני, ואיןו שובר את המבניות.



### What do we mean by "evaluation strategies"?

Issues of evaluation, e.g.,

- Order of operations
- Argument passing
- Caching

Two approaches:

- $\lambda$ -calculus
  - formal and precise
  - but,

- not so intuitive
- heavy on notation

some other time...

Algorithmic

- intuitive
- minimal notation
- but, not so formal and precise

our approach!

### "Undefined behavior" in PLs

PL designers do not specify everything

- If certain patterns are "bad" programming practice...
- If many "legitimate" implementation make sense...
- If different compilers / different architectures may take a performance toll from over-specification...

Then, the PL designer will tend to consider specifying "undefined behavior".

```
messy() {
    int i, n, a[100];
    for (i = 0; i < n; i++)
        printf("a[%d]=%d\n", i, a[i]);
}
```

C Specifying that `auto` variables are zero-initialized may cause an unnecessary performance overhead.

JAVA Advances in compiler technology make it possible for the compiler to produce an error message if an uninitialized variable is used.

### Collateral evaluation

Expressions + side-effects = evaluation order dilemma

In evaluating, e.g.,  $X + Y$ , the PL can decide which of  $X$  and  $Y$  is evaluated first, but, often PLs prefer to **refrain** from making a decision.

#### Definition (Collateral evaluation)

Let  $X$  and  $Y$  be two code fragments (expressions or commands). Then, all of the following are correct implementations of **collateral execution of  $X$  and  $Y$**

- $X$  is executed before  $Y$       • "interleaved" execution
- $X$  is executed after  $Y$       • simultaneous execution

So, in compiling and executing

```
#include <stdio.h>
int main() {
    return printf("Hello,\n") - printf("World!\n");
}
```

there is no telling what will be printed!

### Side-effects $\rightleftharpoons$ evaluation order question

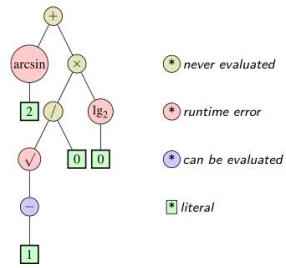
- If expressions have side-effects then there is clearly an evaluation order question

```
printf("Hello,\n") -
printf("World!\n");
```

- Question may pop up even if there are no side effects, e.g., the evaluation of the following pure-mathematical expression over  $\mathbb{R}$

$$\arcsin 2 + \frac{\sqrt{-1}}{0} \times \lg_2 0$$

whose evaluation tree is



Depending on the evaluation order, each of the red nodes may trigger a runtime error first

## Here is the bug!

In PASCAL, operator `and` evaluates **all** of its arguments (even if it becomes apparent that these are immaterial for the results)

```
Array bounds violation!
Procedure Convert(Var c: Char); Begin
  If c >= 'A'
    and
    c <= 'Z'
    and
    rot13[c] <> chr(0) (*X*)
  then
    c := rot13[c]
end;
```

Another annoying (and typical) case...

```
If p <> null and p^.next <> null then ...
```

## Eager/strict/applicative evaluation

### Definition (Eager (strict) evaluation order)

An **eager evaluation order** also called **strict evaluation order** specifies that all arguments to functions are evaluated before the procedure is applied.

- Also called **applicative order**
- Eager order does not specify in which order the arguments are computed; it can be
  - unspecified (collateral)
  - left to right
  - right to left
  - ...
- Most PLs use eager evaluation for
  - all functions,
  - the majority of operators

## Eager vs. short-circuit logical operators

### Definition (Short-circuit evaluation)

Short-circuit evaluation  $\wedge$  and  $\vee$  prescribes that the 2<sup>nd</sup> argument is evaluated only if the 1<sup>st</sup> argument does not determine the result

- Logical "and"** Evaluate the second argument only if the first argument is true
- Logical "or"** Evaluate the second argument only if the first argument is false

PL	Eager version	Short-circuit version
PASCAL	<code>and, or</code>	
C		<code>&amp;&amp;,   </code>
ML		<code>andalso, orelse</code>
EIFFEL, ADA	<code>and, or</code>	<code>and then, or else</code>

## Programming idioms with short-circuit semantics

Beautiful programming idiom (originated by PERL, but applicable e.g., in C)

```
f = fopen(fileName, "r");
|| die("Cannot open file %s\n", fileName);
```

And another one:

```
f == (FILE *)0 || ((void)fclose(f), f = (FILE *)0);
```

More tools of the trade

JAVA/C/C++ Operator "?": (just like if<sup>1</sup>)

C/C++ Operator ,

Similar to PROLOG, except that unlike PROLOG, each expression here yields only one result.

### Clever short circuit evaluation in BASH

A BASH program to remove contents of current directory.

```
for f in *; do
    echo -n "$f: "
    [-e $f ] || echo "already removed"
    [-d $f ] &&
        echo -n "removing directory" &&
            rmdir $f &&
                [-e $f ] && echo "... failed"
                || echo ""
            )
    [-e $f ] &&
        echo -n "moving to /tmp" &&
            mv -f $f /tmp &&
                [-e $f ] && echo "... failed"
                || echo ""
            )
done
```

- BASH commands may succeed or fail:
  - Success returns true (integer 0)
  - Failure returns false (error code ≠ 0)
- "[" is a command; it takes arguments; it may succeed or fail:
  - [-e f] 3 arguments to command "["; succeeds if file f exists
  - [-d f] 3 arguments to command "["; succeeds if directory f exists

### Emulating short-circuit operators with conditional commands

*poor substitute, operators occur in expressions, and expressions are not commands*

Logical "And"

```
If p^ <> null then
  If p^,next <> null
    (* some command *)
```

Logical "Or"

```
If p^ = null then
  (* command, possibly SKIP *)
else if p^,next <> null then
  (* some command *)
```

## Normal evaluation order

### Definition (Normal evaluation order)

In normal evaluation order arguments to functions are evaluated whenever they are used by the function.

- Is a generalization of "short circuit evaluation"
- The terms
  - "normal evaluation order" and
  - "normal order evaluation"are synonymous.
- Can be used to encapsulate any of the following C operators in functions:
  - &&
  - ||
  - ,
  - ?: :

### Normal vs. applicative order:

Normal order a "symbolic expression"<sup>2</sup> is passed; this expression is evaluated by the callee when it needs it.

Applicative order a "symbolic expression" is evaluated by the caller, the result is passed to the callee.

<sup>2</sup>"symbolic expression" is an imprecise name for  $\lambda$ -expression.

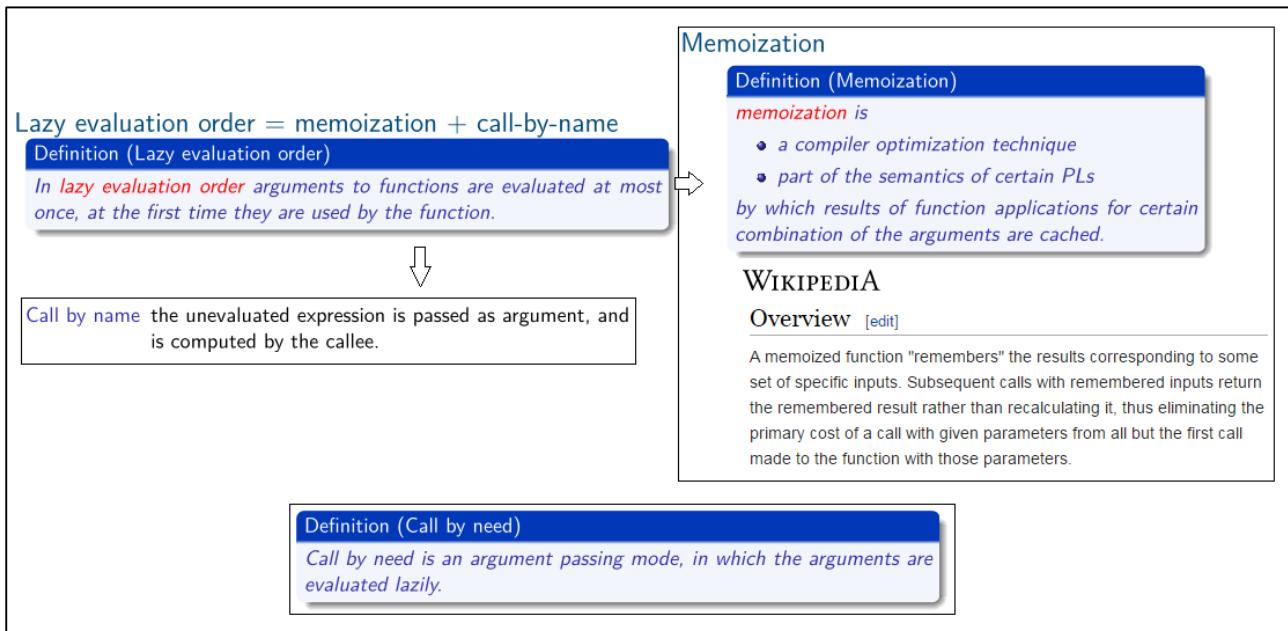
## Capitalizing on normal order evaluation

A variant of the "die" programming idiom:

### Clever "unless" function

```
static <T> // exists for each type T
T unless(
    boolean b, // !(precondition)
    normal T t, // value
    normal Exception e // in case
    precondition fails
) {
    if (b)
        throw e; // Do not evaluate t
    return t; // Do not evaluate e
}
```

- Parameters marked **normal** are evaluated (only) when used.
- If precondition *fails*, then exception *e* is evaluated and then thrown.
- If precondition *holds*, then argument *t* is evaluated and then returned.



### סיכום:

כאשר ישם שני ביטויים או יותר שיש להם Side Effects, נוצרת הדילמה של מה יהיה סדר השערוך שלהם. רוב שפות התכנות בוחנות שלא להתחייב מה יהיה סדר השערוך במאשנקרה Collateral Evaluation.

לעתים כדי לדעת כיצד תכנית תנаг במצב מסוים (או אפילו אם היא תקרוס) علينا לדעת את סדר השערוך של הביטויים.

### סדרי שערוך שונים:

- **Eager\Strict\Applicative Evaluation** - כאשר כל הפרמטרים לפונק' משוערכים טרם תחילת ריצתה (לפני שהפונקציה מופעלת – ע"י ה- Caller) – לא ידוע באיזה סדר, תליי שפה.
- **Short Circuit Evaluation** - בשערוך של ביטויים לוגיים הכלולים or או and ובهم הארגומנט השני ייחסב רק אם הראשון לא מספיק לקבע ערך הביטוי הכלול.
- **Normal Evaluation Order** - הארגומנטים לפונק' ישוערכו בכל פעם שיישנה בהם שימוש.
  - o אנו מעבירים "Symbolic Expression" וה- callee משוערך ע"י ה- caller הוא צריך אותו.
  - o ברגע ל- Eager\Strict\Applicative Order שבו ה- Symbolic Expression משוערך ע"י ה- caller (כלומר לפני הכניסה לפונקציה) והתוצאה מועברת ל- Callee.
- **Lazy Evaluation Order\Call by need** - ארגומנט לפונק' ישוערך בפעם הראשונה שבה יעשה בו שימוש בפונקציה ורק בפעם הראשונה.
  - o דומה מאוד ל- Normal Evaluation Order אבל בניגוד אליו – ב- Lazy כל ביטוי ישוערך רק פעם אחת. השערוך יבוצע בתחום ה- Callee וכך נדרש ציריך את הערך.

### מידע נוספת:

**Memoization** = caching function result -

### :Abstraction

הפשטה (בלועזית: אבסטרקציה) היא פעולה מחשבתייה הכוללת הגדרה ושימוש במושג חדש המቤטא מכנה משותף של מושגים קיימים. ההפשטה מאפשרת להתייחס לעניין מסוים בהקשר רצוי תוך התעלמות מהפרטים שאינם רלוונטיים להקשר. לדוגמה המושג "אדם" הוא הפשטה של הרבה סוגי של בני אדם (נשים, גברים, נזירים, מוסלמים וכו').

- **Function Abstraction** – מכיל ביטוי שצריך לחשב אותו.
  - **Procedure Abstraction** – מכיל פקודה שצריך לחשב אותה.
- היחסובים מתבצעים כאשר האבסטרקציה נקראת.

### – הכללה של אבסטרקציה.

**Formal Parameters** (פרמטרים פורמליים) – לדוגמה z בהגדרת הפונקציה הבאה:  
**fun circum ( r:real ) = 2 \* r;**

(פרמטרים אקטואליים) – לדוגמה:

circum(1.0)  
circum(a+b)

**Argument** – הוא ערך שניתן להיות מועבר לאסטרקציה.

- לדוגמה בפס卡尔 כל הערבים חוץ מקבצים יכולים להיות מועברים לאסטרקציה.
- לדוגמה ב- ML כל הערבים יכולים להיות מועברים לאסטרקציה.

שאלה:

<sup>73</sup> מה הבדל בין פרמטר לערך?

- 

תשובה:

"הבדל המהותי בין פרמטר לערך הוא סמנטי בעיקרו ונועד להבדיל בין הגדרה של פונקציה לבין קריאה וזימון שלה.  
בעיקרון פרמטרים הם שמקבלים הפונקציה בשורת ההגדרה שלה ואילו ארגומנטים הם אלו שמעברים בעת קריאה  
וזימון של הפונקציה דה פקטו.

כך למשל ניתן לראות בדוגמה שלහן:

```
int f(struct S *x);  
return f(&y);
```

כאשר x הוא פרמטר ו- y הוא הארגומנט של f.

.Actual Parameters Formal Parameter ובין Parameter Passing

קיימים מגוון של שיטות – Argument passing modes

: ניתן להלך אותם ל- 2 קונספטים עיקריים:

.by value, לדוגמה Copy -

.by-reference, לדוגמה Definitional -

(Thunks בהמשך מוסבר גם על

<sup>73</sup> נלקח מהאתר Safot.net – קישור.

### 7.1.5. Argument passing modes

Modes of applicative order:

<b>Call by value</b>	a copy of the evaluated argument is passed.
<b>Call by reference</b>	a reference to the evaluated argument is passed.
<b>Call by result</b>	no value is passed to callee, on return, value is copied back.
<b>Call by value-result</b>	call by value plus call by result. We also talk about "call by name", which is a nothing but "normal order"
<b>Call by name</b>	the unevaluated expression is passed as argument, and is computed by the callee.

### Implementation of argument passing modes

Implementation of call by ...

<b>value</b>	before call, push value onto stack
<b>reference</b>	before call, push address onto stack
<b>result</b>	after call, pop value from stack
<b>value-result</b>	<ul style="list-style-type: none"> <li>before call, push value onto stack</li> <li>after call, pop value from stack</li> </ul>
<b>name</b>	with "thunks" ...



Normal order arguments can be replaced by nested functions

#### Definition (Thunk)

A **thunk** is a compiler generated nested function that implements a "normal" order argument

- Thunks are only passed as arguments.
- Thunks are never returned.
- Therefore, inner functions as in PASCAL suffice.
- Below we shall see how nested functions are implemented.
- To implement "call by reference by name", the thunk simulating normal order arguments, must be able to return references to variables.



### Implementation of "call by name"

Think of automatic conversion of

```
Integer f(Integer a, b) {
    return unless( // Recall that unless is normal-order in its 2nd and 3rd arguments.
        b == 0,
        // Convert into a thunk:
        a / b,
        // Convert into a thunk:
        new ArithmeticException("Dividing_value_" + a + " / by_0 !");
    );
}
```

into

```
Integer f(Integer a, b) {
    Integer _1() { return a / b; } // 1st thunk implementing 1st argument
    Exception _2() { // 2nd thunk: implementing 2nd argument
        return new ArithmeticException(
            "Dividing_value_" + a + " / by_0 !");
    }
    return unless(b == 0, _1, _2);
}
```

לסיום:

- מוחושב לפני הקריאה, וMOVPERFOR - reference –callee, reference – ה- L-Value. לאחר מכן ניתן לשנות ולהקצותות לתוכה הערך.

פרמטרים אקטואליים: (Actual Parameters) - MOVPERFOR ה- R-value – Call by value – Callee ה- R-value – Call by value – השפעה או להקצות ערכיהם חדשים לתוכה מה שקיבלו. הערכים שפועלים על מה שקיבלו יושבים על ה-stack.

- Legality** : Any first class value.
- Entry effect**: Evaluated, value assigned to formal parameter.
- Exit effect** : none.

- שום ערך לא MOVPERFOR לפונקציה (Callee) אבל היא מוחזירה את הערך כעטקה (Result) לאחר מכן השועבר לה.

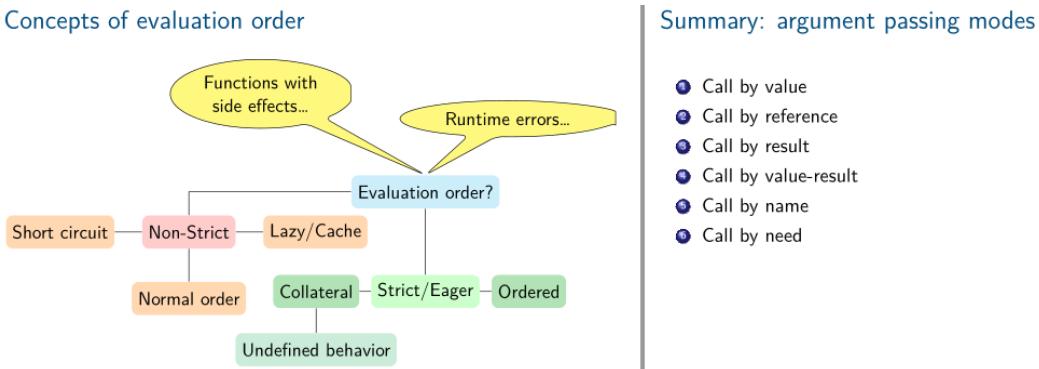
- Legality** : a variable.
- Entry effect**: none (initial value is undefined)
- Exit effect** : final value of the local variable assigned to argument variable (actual parameter)

.Call by Value + Call by Result – זהו – Call by value – result -

- Legality** : a variable.
- Entry effect**: Value assigned to formal parameter.

- **Exit effect** : Return value assigned to actual parameter.

- **Call by name** – הביטוי הלא משוערך מועבר כמו שהוא לפונקציה ומהו שפונקציית Callee. מtabצע בעזרה.
- **Call by need** – כאשר הארגומנטים משוערכים בצורה lazily – ככלומר ארגומנטים לפונקציה משוערכים לכל היותר פעם אחת – ורק כשהפונקציה משתמשת בהם.



### **Strict vs. Non-Strict Functions :**

- **Strict function in the  $n^{th}$  argument:** a function f is said to be strict in its  $n^{th}$  argument if it cannot be evaluated unless this argument is evaluated.
  - **Non-Strict in the  $n^{th}$  argument:** a function f is said to be non-strict in its  $n^{th}$  argument if there are cases in which this argument cannot be evaluated but f can.

לדוגמא עברו:

	<u>n</u>	<u>t</u>	<u>eager</u>	<u>normal-order</u>
♦	2	0.8	false	false
♦	0	0.8	<b>FAIL!</b>	false

- **Strict** ברגומנט הראשון שלה.
- **Non-Strict** ברגומנט השני שלה.

## Discrimination against function values

In PASCAL

- Can define variables of almost any type, but not of type function (**Discrimination**).
  - Functions can take function values as parameters, but functions cannot return function values (**Discrimination**).

## Function values in C

In C, function values are realized as “function pointers”

- Can be stored in variables
  - Can be stored in arrays
  - Can be fields of structures
  - Can be passed as parameters
  - Can be returned from functions

But, C

- does not provide means for creating function values at runtime (**Discrimination!**)
  - does not allow nested functions (**Discrimination!**)

שאלה שקשורה לנושא:  
74 - שאלה 10

א. כתוב תכנית אשר הפלט שלה יהיה המחרוזת "VR" אם בשפת התכנות העברת פרמטרים היא BY VALUE והפלט של אותה תכנית עצמה בשפה שבה העברת הפרמטרים היא BY REFERENCE. מהו המחרוזת "R".

ב. הסבר מהו שערוך מסוג "LAZY NORMAL ORDER EVALUATION" וציין מה הדומה והשונה ביןיהם.  
מקור: מועד א', אביב תשע"ה  
פתרונות סעיף א':

```
#include <iostream>
#include <string>

using namespace std;

void doSomething(string str) {
    //Erases the first character without reallocating the string.
    str.erase(0,1);
}

int main() {
    string s= "Something";
    string *t = &s;
    doSomething(s);

    if (&s == t) {
        cout<< "R";
    } else {
        cout<< "VR";
    }
    return 0;
}
```

פתרונות סעיף ב':

Normal Order Evaluation – הפונקציה משערכת את הארגומנטים הנכנסים בכל שימוש בארגומנט בפונקציה.  
Lazy Evaluation – הפונקציה משערכת את הארגומנטים רק בפעם הראשונה.

דוגמא: שניים משערכים את הארגומנטים בפעם הראשונה.  
שונה – Normal Order Evaluation משערכת את הארגומנטים בכל פעם שהפונקציה צריכה להשתמש בבייטוי.  
שוני נוסף הוא שב- Lazy Evaluation משתמשים יותר זיכרון מ- Normal Order Evaluation כדי לשמור את הערכים המשוערכים.



## :Environment

- זהה למשה set of bindings. מהם bindings? כל היישות (לדוגמה משתנים, פונקציות קבועים וכו') הנמצאים ב- Environment. Bindings Scoping. dynamic scope. lexical scope. static scope.
- "אוסף קישוריהם/מוזהיהם המוכרים בנזודה מסוימת בתכנית".<sup>75</sup>

סוגי ה- Scoping כדי לקבוע באיזו פונקציה נחשף את ה- Bindings:

### Bindings שלא התבצעו בתוך הפונקציה נקבעים ע"י:

#### Dynamic Scoping

פירוש שם (של משתנה לדוגמא) תלוי במצב התכנית כאשר מנסים לבדוק מה הוא אומר. ככלומר תלו依 בזמן ריצעה של הפונקציה – מתי שהפונקציה אקטיבית. Binding – תהייה לנו בזמן ריצעה רשימה של מיליון ונותהמש בה בצורה דומה ל- .Back Pointer

#### Static (lexical) Scoping

פירוש שם (של משתנה לדוגמא) תלוי באיפת הפונקציה הגדולה בקוד – והוא תלוי בהקשר של הפונקציה מבינה סינטקטית. - נעשה בעזרת Back Pointer (BP) שהוא מצביע על רשומת המחסנית של הגרסה האחרונה ביותר של הפונקציה המכילה את ההפונקציה הנקרהת.

:<sup>76</sup> מוקפדייה

### Lexical scope vs. dynamic scope [edit]

A fundamental distinction in scoping is what "part of a program" means. In languages with **lexical scope** (also called **static scope**), name resolution depends on the location in the source code and the **lexical context**, which is defined by where the named variable or function is defined. In contrast, in languages with **dynamic scope** the name resolution depends upon the **program state** when the name is encountered which is determined by the **execution context** or **calling context**. In practice, with lexical scope a variable's definition is resolved by searching its containing block or function, then if that fails searching the outer containing block, and so on, whereas with dynamic scope the calling function is searched, then the function which called that calling function, and so on.<sup>[4]</sup>

Most modern languages use lexical scoping for variables and functions, though dynamic scoping is used in some languages, notably some dialects of Lisp, some "scripting" languages like Perl, and some template languages.<sup>[c]</sup> Even in lexically scoped languages, scope for closures can be confusing to the uninitiated, as these depend on the lexical context where the closure is defined, not where it is called.

Lexical resolution can be determined at **compile time**, and is also known as **early binding**, while dynamic resolution can in general only be determined at **run time**, and thus is known as **late binding**.

### :Static Binding

הסבירה נקבעת בצורה סטטית לפי מקום ההגדלה ההפונקציה.  
במובן זה "ירוש" רק מי שבתוכו הוא מוגדר, ככלומר נקבע ע"י ה- scope.  
לא משנה איפה פונקציה קוראת לו – זה נקבע ע"פ המקום בו הוא נמצא.  
<sup>77</sup> "כאשר הקישור בין ישות למזהה שלה נעשית בזמן קומpileציה בהקשר ההגדלה של הישות".

### :Dynamic Binding

הסבירה נקבעת בצורה דינמית לפי מי שקורא לפונקציה.  
<sup>78</sup> "כאשר הקישור בין ישות למזהה שלה נעשית בזמן ריצעה בהקשר הסביבה הנוכחי של הישות".  
Bash, Lisp  
דוגמאות:

מ- 56 שאלות ותשובות של אריה,<sup>75</sup>

<sup>76</sup> [https://en.wikipedia.org/wiki/Scope\\_\(computer\\_science\)#Lexical\\_scope\\_vs.\\_dynamic\\_scope](https://en.wikipedia.org/wiki/Scope_(computer_science)#Lexical_scope_vs._dynamic_scope)

מ- 56 שאלות ותשובות של אריה.

מ- 56 שאלות ותשובות של אריה.<sup>77</sup>

מ- 56 שאלות ותשובות של אריה.<sup>78</sup>

```

//Dynamic Binding vs Static Binding:
const s=2; //Binding!

function foo(i:integer):integer //Binding i with a parameter.
begin
    foo:=i*s;
end;

Procedure P;
    const s=3; //Binding once again of a value and a name.
begin
    ...
    //what will the value "foo(1)" be, if used here?
    ...
end;

//In dynamic Binding, "foo(1)" will get the value 3.
//In Static Binding, "foo(1)" will get the value 2.

```

שאלות בהקשר זהה:

**:static scoping vs dynamic scoping<sup>79</sup>** -  
נתון הקוד הבא בשפה דמוית פסקל:

```

program karnash;
const m = 1;
function f(n : Integer) : Integer
    f:=m+n;
function g(n : Integer) : Integer
    g:=f(n*m);
function h(n : Integer) : Integer
    const m = f(n+m);
    h:=g(n);
begin
    const m = 2;
    writeln(g(10));
    writeln(h(5));
    writeln(m);
end.

```

1. מה יהיה הפלט של התוכנית, אם היא משתמשת בשיטה של static scoping. אין לזרף הסברים.
2. מה יהיה הפלט של התוכנית, אם היא משתמשת בשיטה של dynamic scoping. אין לזרף הסברים.

פתרונות (אושר ע"י פרופ' גיל):

1. הפלט הוא:

11  
6  
2

2. הפלט הוא:

22  
54  
2

<sup>80</sup> האם נכון לומר שאין ייצוג ל-environment במחסנית לשפת C מכיוון שהשפה לא מאפשרת קינון פונקציות? -  
האם ניתן להגדד מעט את הקשר בין השניים?

"במילה אחת: נכון."

בקצת יותר פירוט. באופן כללי ביחס הסביבה של פונקציה, היא אוסף ה BINDINGS הנגיש לפונקציה - קלומר, אוסף השמות הנגינשים, והישויות אותן מציינים השמות הללו.  
בשפת C הסביבה מורכבת מ-

<sup>79</sup> נלקח מהאתר Safot.net – [קישורים](#).

<sup>80</sup> נלקח מהאתר Safot.net – [קישורים](#).

- א. פרמטרים.
  - ב. הגדרות מקומיות .
  - ג. הגדרות גלובליות .
- אבל, הגדרות הגלובליות בשפת C הן קבועות, ואין תלויות בקריאה."

## Environment vs. scope

In simple words...

- Q: What's the environment?
- A: All variables which are in "scope"
- Q: What's scope?
- A: The extent of locations in the program in which a binding is recognized.

Two scoping (binding) rules are found in PLs

**Lexical scoping** Bindings not made in function, are determined by where the function is **defined**.

**Dynamic scoping** Bindings not made in function, are determined by when the function is **active**.

Environment could be

- Defined statically (static scoping)
- Defined dynamically (dynamic scoping)

## Dynamic scoping

### Definition (Dynamic scoping)

*The environment is determined by calls; if  $f_1$  calls  $f_2$ , then  $f_2$  "inherits" bindings made by  $f_1$ .*

Dynamic scoping cares not whether there is any relationship between  $f_1$  and  $f_2$ .

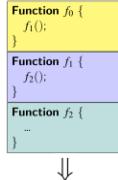
- Found in LISP (but not in SCHEME)
- Found in TeX(text processing PL used to produce this material)
- Found in PERL!
- Found in BASH!
- Found in HTML/CSS!

The de-facto semantics of the C preprocessor.

When does  $f_2$  "inherit" bindings made in  $f_1$ ?

**Answer I/II:** (dynamic scoping) If  $f_2$  is called by  $f_1$ .  
(sometimes called dynamic binding)

Machine stack



- Inheritance is inductive:  $f_2$  inherits bindings of  $f_0$  if  $f_1$  is called by  $f_0$ .
- Environment defined by program dynamics
- Conversely, the scope of bindings made in  $f_1$  is:
  - all functions called, directly, or indirectly by  $f_1$
  - determined dynamically
  - hence the term *dynamic scoping*

## Semantics of dynamic scoping

```

void (*exchange)(int, int);
void sort(int a[], int n) {
    void swap(int i, int j) {
        const int t = a[i];
        a[i] = a[j];
        a[j] = t;
    }
    ...
    exchange = swap;
}
    
```

```

#define SIZEOF(z) (sizeof(z) / sizeof(z[0]))
int main(int argc, char **argv) {
    int a[500];
    int b[500];
    ...
    sort(b, SIZEOF(b)); <-->
    (*exchange)(10,12);
}
    
```

What would work?

- sort calling qsort? Yes.
- qsort calling qsort? Yes.
- qsort calling pivot? Yes.
- main calling swap? Yes.

## The runtime bindings dictionary

The "binding dictionary":

- Local state is represented by a dictionary
- Dictionary supports mapping between names and entities
- Typical entities are variables, constants, and functions.
- Typical implementations of dictionary are linked list and hash table.

The environment:

- The environment is represented as a "back pointer" (BP) to the most recent stack frame.
- Thus, we have a "stack of dictionaries".
- Search for name is carried out by searching in the stack of dictionaries.
- There are means to make this search quite efficient.

## Intricacies of static scoping

### Definition (Static scoping)

*The environment is determined by scope; if  $f_1$  is defined in  $f_2$ , then  $f_2$  "inherits" bindings made by  $f_1$ .*

- There might be more than one activation record of  $f_1$  on the stack
- The caller of  $f_2$  might not be  $f_1$
- Static scoping means the **most recent** version of  $f_1$ .

When does  $f_2$  "inherit" bindings made in  $f_1$ ?

**Answer II/II:** (static scoping) If  $f_2$  is defined **inside**  $f_1$ .  
(sometimes called static binding, or lexical scoping)

Program code

```

Function f0 {
    Function f1 {
        Function f2 {
            ...
        }
    }
}
    
```

- By induction,  $f_2$  inherits bindings of  $f_0$  if  $f_1$  is defined inside  $f_0$ .
- Environment defined by static code structure.

## Convoluted calls

Who can call function  $f_2()$ ?

```

Function f0() {
    Function f1() {
        Function f2() {
            f4() {}
        }
    }
    Function g1() {
        Function g2() {
            g3() {}
        }
    }
}
    
```

The caller of  $f_2$  is not necessarily  $f_1$ ; it could be

- function  $f_2$  itself
- function  $f_3$  defined in  $f_2$
- function  $f_4$  defined in  $f_3$
- ...
- another function  $g_1$  defined in  $f_1$
- function  $g_2$  defined in  $g_1$
- function  $g_3$  defined in  $g_2$
- ...

But not function  $f_0$  within which  $f_1$  is defined!

**רשומת הפעלה** היא מייצגת את המשק בין הפונק' הקוראת לפונק' הנקראת. היא שומרת מידע שיאפשר לשזר את מצב הפונק' הקוראת בתום ריצת הפונק' הנקראת. כמו כן שומרת את המצב של הפונק' הנקראת ואת הארגומנטים שהועברו לה. דבר אחרון שנשמר הוא הסביבה (Environment), שמאפשרת את ה-"ירושה" של משתנים הגדרות וכו' מהקורסא לקרים.

## Function variables & dangling references

Similarly, suppose that PASCAL had function variables...

```
VAR
  fv: Integer -> Char;
Procedure P;
  VAR
    v: _ ;
  Function f(n: Integer): Char;
  begin
    _v_ := n;
    begin
      fv := f;
    end
  begin
    P;
    fv(0) := v;
  end;
```

- The environment "dies" as a stack frame is popped.
- C forbids nested functions for this reason.
- PASCAL forbids function variable for this reason
- To make first class "function values", the activation record cannot be allocated on the hardware stack.

## Nested functions escaping the nest (Gnu-C)

```
#include <stdio.h>

typedef int (*F)(int);

// The nest:
F makeAdder(int a) {
    // The nested:
    int add(int x) {
        // The use of environment (variable a)
        const int $ = a + x;
        printf("%f. add(): return %d\n", $);
        return $;
    }
    // The escape:
    return add;
}

// Refuge for escaped values:
F add5, add7;
```

- F is an alias for the type of a function that takes an integer and returns an integer.
- Function add is nested in function makeAdder, the nest.
- The nested uses the environment (e.g., variables) of the nest.
- The nested escapes the nest.
- The escaped value will be stored in global variables add5 and add7, the refuge.

## Lost environment

```
// Provide refuge to the escaping:
void initialize() {
    add5 = makeAdder(5);
    add7 = makeAdder(7);
}

int dozen() {
    // Use the nested that escaped:
    return [add7(add5(0))]; // X
}

// Force program crash:
int main() {
    initialize();
    printf("dozen=%d\n", dozen());
    return 0;
}

f. add(): return 7
Segmentation fault (core dumped)
```

## Forcing access to lost environment

- Store in the refuge two versions of function add, the nested that escaped.
- Both use the environment of the nest, function makeAdder.
- But function makeAdder is not active anymore.
- So, we are doomed for trouble...

## What are closures?

- Closures can be thought of as
  - First class nested functions.
  - Functions that close over their environment.
  - Functions whose activation record is heap managed (rather than stack managed).
- Many modern PLs support closures, where a function can be created in a way that it “keeps with it” the variables that exist in its scope.
- Often used in functions-that-return-functions
- You can't do that in PASCAL, C, C++
- Very common in functional languages, ML, PYTHON, JAVASCRIPT and JAVA (since version 8.0; uses a hack).

## Closures in JAVASCRIPT

**A closures factory**

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

**Usage**

```
var add3 = makeAdder(3);
document.write(add3(7));
document.write(add3(12));
var add8 = makeAdder(8)
document.write(add8(12));
```

### Output:

```
10
15
20
```

## Closures and lifetime

- **Q:** what's the lifetime of a variable enclosed in a closure?
- **A:** the lifetime of the enclosing value.
  - In this example, the lifetime of “`delta`” for each function returned by `makeAdder` is the lifetime of the return value of `makeAdder`.
  - The same as lifetime of fields in a record allocated on the heap: live as long as the record is still allocated.

In general, you cannot have closures without using GC, rather than the stack for implementation of activation records. With closures, activation records are allocated from the heap.

## Closures in ML

Just as in JAVASCRIPT, in ML, all functions are closures.

- Standard programming idiom of the language
- Also supports anonymous functions
- Function values are first class values (including the environment)

## Emulation closures with C++ function objects

Class of “adding something” function objects:

```
class FunctionObjectForAdding {
public:
  FunctionObjectForAdding(int _b): b(_b) {}
  int operator() (int a) { return a + b; }
private:
  int b; // Saved environment
};
```

- A function object stores a copy of relevant pieces of the environment as data members of the class.
- Environment copy is managed by the function object, which, just like any other C++ variable, can be
  - present on the stack
  - allocated from the heap
  - global
- Memory management is the programmer's responsibility.

## Closures in JAVA?

- In JAVA the only first class values are objects.
- Hence, in JAVA functions are second class citizens.
- JAVA does not have real closures.
- Still, you can imitate closures with objects.

Imitation is just like C++'s function objects

## Function objects with JAVA

Class of function object

```
class Log {
  final double logBase; // Captured environment
  Log(final double base) {
    logBase = Math.log(base);
  }
  public double apply(double v) {
    return Math.log(v) / logBase;
  }
}
```

Using the function object:

```
public class L {
  public static void main(String[] args) {
    final Log log2 = new Log(2);
    System.out.println("Log base 2 of 1024 is " + log2.apply(1024));
  }
}
Log base 2 of 1024 is 10.0
```

תמונה של השפה המאפשרת לגשת לפונקציה והמשתנים שלה גם לאחר שההקשר של הפונקציה פסק מלהתקיים.

אפשר לפונקציה לרשום עמה את כל המשתנים שמוגדרים ב-**Scopes** שלה.

עם **Closures** רשותה הפעלה (ובפרט הסביבה) מוגצת ו נשמרות על ה-**Heap**.

.First Class Nested Functions ניתן לחושב עליו-כ-

:ML דוגמא ב- -

```
//In ML:
fun f x = fn y => x+y;
f 3 // The 3 will be in the Closure
```

ב-**Closure** לא מספיק לנו המצב ( execution ) ( הנקחי, אנחנו צריכים גם את הסביבה. <sup>82 81</sup> התנאים של השפה הנדרשים -**Closure** -

.First Class Nested Functions .פונקציות הן מטיפוס ראשון -

.Nested functions .השפה צריכה לאפשר פונקציות מקוונות -

.Garbage Collection .ניהול זיכרון אוטומטי –

<sup>81</sup> מה-56 שאלות של אריה.

<sup>82</sup> נלקח גם מהאתר Safot.net – קישור.

<sup>83</sup> פונקציות מקווננות ופונקציות מטיפוס ראשון נדרשים בשליל Closure מכיוון שהוא הקשר שבו אנחנו רוצים את זה – Closure מכלתיתילה. הניהול זיכרנו האוטומטי נדרש משום שהוא חייבים מגנון שינה את הסביבה של ה- Closure נשמר על ה- Heap.

בשפה שבה פונקציות הן לא first class, אין closure. First Class Function הוא first class function ולכך בלאו אין closures – פונקציות הן לא final בלאו אומר שהערך לא ישנה במהלך הריצה של התכנית.

שאלה שקשורה לנושא ה- Closures:

2 דוגמאות לשפה/דיאלקט שבו יש פונקציות מקוונות אך אין בהן closure?

בשאלה זו, אתם מתבקשים לחתם 2 דוגמאות. אחת של שפה, והאחרת של דיאלקט של שפה. ב-2 הדוגמאות קיימות פונקציות מקוונות, ולא קיים closure.

א. מהו הקושי המתעורר?

ב. במקרה הראשון, הקושי נפתר על ידי חסימות (אלו בדיק?) של השפה, המחייב הוא פגיעה בעקרון השיוויון (איך בדיק?). באיזו שפה/עגה מדובר? השב, נמק, והדגם.

ג. במקרה השני, הקושי נפתר באמצעות עיקרונו יסוד זה, וכייזד הוא מתמודד עם הקושי?

פתרון: (אושר ע"י פרופ' גיל)

"שפה: פסקל."

דיאלקט של שפה: Gnu-C.

א. הקושי המתעורר הוא במקרה שרוצים להשתמש בפונקציה פנימית שמשתמשה במשתנה שהוגדר בפונקציה החיצונית, אך הפונקציה החיצונית סיממה את ריצתה. כדי שמקורה זה ייתמך בשפה, השפה צריכה לתמוך ב- Closures, כך שהפונקציה הפנימית תמיד תשמור את הסביבה שלה, כולל המשתנה החיצוני לה, ולכן תמיד תעבור.

ב. במקרה של שפת פסקל, הקושי נפתר על ידי אפליה נגד פונקציות, כך שלא ניתן להחזיר פונקציה מקוונת מתוך הפונקציה החיצונית (אי אפשר גם לשמר פונקציות במשתנים), וכך בכל מקום בתוכנית שבו אפשר להריץ את הפונקציה הפנימית יותר, הפונקציה החיצונית עדין רצה. המחר ששולם הוא כפי שהזכרתי אפליה נגד פונקציות, וזהו אחת הסיבות לכך שערכי פונקציות הם לא ערכים סוג א' (First Class Values).

ג. במקרה של הדיאלקט Gnu-C, הקושי נפתר על ידי עקרון היסוד ב- C – הטוען שהמתכונת צריך להיות אחראי לכך שהתכוונית במצב תקין בכל רגע נתון. עבר פונקציות מקוונות הדבר בא לידי ביטוי בכך שאמנם התנהגות התוכונית לא מוגדרת במקרה שהזוכרתי [כאשר פונקציה פנימית משתמשת במסתנים חיצוניים לה מושצת לאחר סיום ריצת הפונקציה החיצונית], אבל מצופה מהמתכונת להימנע מהטיסוטואציה הזאת עצמה".

### Generator

- הוא פונקציה שמייצרת רצף (יכול להיות בגודל 0 וגם אינסוף) של תוצאות במקומות תוצאה אחת.
- סוג פונקציות כזו יכול לשמש אותנו לשילטה על התוכונית כמו פקודות תנאי ולולאות.
- שומר על ה- Activation Record בדומה ל- Closure.
- שומר על ה- Context of the callee – ובכך מאפשר לנו לחזור למצב בו השארנו אותו לאחר שייצאנו.
- מוגדר כמו פונקציה, כמעט מילאית (Yield).
- מייצר-Sequenc של תוצאות, במקומות להחזיר תוצאה יחידה.
- ב- ICON כל Generator expression הוא DOMIMים לפירדיקטים ב- Prolog.

<sup>83</sup> נלקח גם מהאתר Safot.net – קישור.

## Generators

Generator is a function...

Iterator is an object...

### Definition (Generator)

A **generator** is a function that produces a sequence of results, instead of returning a single result.

### Definition (Iterator)

An **iterator** is an object which can be used to traverse a datastructure such as a list.

The sequence may be:

- empty
- infinite
- (but not both)

## Generators vs. functions

	Function	Generator
# results:	1	0, 1, 2, ..., ∞
Terminology:	<ul style="list-style-type: none"> <li>• return</li> <li>• call / invoke</li> </ul>	<ul style="list-style-type: none"> <li>• yield / produce</li> <li>• initiate</li> <li>• retrieve</li> </ul>

- A function must return one value, even if it has nothing to offer.
- Generators have two kinds of "invocations":
  - `initiate` start the production process
  - `retrieve` obtain the next element from the sequence

## Functions vs. generators in PYTHON

A generator looks like a function, but behaves like an iterator:

Function <code>fu</code>	Generator <code>gen</code>
Definition:	<code>def fu(x):     return x</code>
Invoking as function:	<code>def gen(x):     yield x</code>
Invoking as generator:	<pre>gen(3) &lt;generator object gen at 0x7f65af3eae60&gt;</pre> <pre>for i in gen(3):     print i 3</pre>

בדומה ל-**Generators**, **Closures**, **המישר את פעולה הפונק'** וכן מה ערכה. **:Generator** דוגמא לולאה הבנויה מ-

## The generating context

```
foreach (int p in Powers(2, 30)) {
    Console.Write(p);
    Console.Write(" ");
}
```

is implemented like so:

```
// This variable is inaccessible to programmer
GeneratingRecord _;

for (_ = Powers(2, 30);
     _.resumable();
     int p = _.retrieve())
{
    Console.Write(p);
    Console.Write(" ");
}
```

.ב-**Java** נוכל למשתמש במערכות שתி מחלקות כאשר אחת מהן היא האיטרטור. כמו כן ישנו פיצרים נוספים של איטרטורים ב-**Java** שמאפשרים פעולות דומות לאלו שנთן לעשות עם **Generators**.

היא פונקציה שיכולה להשוו את הביצוע שלה ע"י פקודת `yield` בכל זמן בביצוע שלה ולהמשיך את ביצוע התוכנית מהפוקודה הבאה אחרי ה-`yield`.

היא אבסטרקציה הולכת בתוכה את מצב הביצוע של פונקציה מסוימת, ומאפשרת לשזרו. ה- Continuation – **Continuation Closure** – לפיקר כללי יותר מ- Closure במשמעותה המקורייה, אלא גם במקרה שלשיי במהלך ביצועו.

- תיאור מופשט של מצב תוכנית מחשב – הפיכת ה- Activation Record לאורה סוג א'.

## coroutines

### Definition (Coroutine)

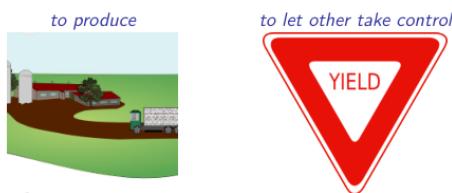
- A **Coroutine** is a function, which can
- **suspend** anywhere in its execution by executing a **yield** command.
  - **resume** a suspended execution, proceeding to the command that follows the suspending **yield**

Coroutines vs. generators:

- Generators are also called semicoroutines
- Generator can only **yield** to its caller
- Coroutine typically **yield** to another coroutine
- have (partial) implementation in mainstream PLs such as C, C++, JAVA, PYTHON, PERL, Object-PASCAL, SMALLTALK and more
- implementations did not really catch

### Overloading of "yield": two unrelated meanings

Two overloaded meanings of the word "yield" in English:



In theory of PL:

Generators to produce

Coroutines to let other take control

### Coroutines vs. threads

Both offer multitasking:

Threads preemptive; execution can be interrupted at any point

Coroutine cooperative; control passed when the

Zillions of weird scheduling schemes?

Threads Yes!

Coroutine only with careless programming

Race conditions?

Threads Yes!

Coroutine No!

Generally speaking, coroutines grant the programmer fine control of scheduling, but it may take skill to effectively use this control.

### Unconvincing example of coroutines

A sow and its two little piglets may all die in the pigsty

```
void sow() { for (;;) {
    defecate();
    if (hungry). feed();
    if (thirsty). drink();
    yield piglet1;
}}
```

```
void piglet1() { for (;;) {
    defecate();
    if (hungry || thirsty). suck();
    yield piglet2;
}}
```

```
void piglet2() { for (;;) {
    defecate();
    if (hungry || thirsty) {
        suck();
        do something
    }
    yield sow;
}}
```

If **piglet2** is not hungry neither thirsty.

- sow may die
- piglet1 will then die
- piglet2 himself will eventually die

### Issues with the pigsty

It is not clear at all

- how the coroutines are initiated?
- how each coroutine gains access to the other?
- how can one create multiple instances of the same coroutine?  
(create several active invocations of (say) **piglet2()**)

## Continuations

Quite an obscure and abstract definition:

### Definition (Continuation)

A **continuation** is an abstract representation of the control state of a computer program

More plainly, a continuation is making the activation record (more generally, the generating record) into a first class citizen.

- Invented many years ago
- Has full implementation in many dead and obscure PLs
- Has partial/non-official implementation in many mainstream PLs
- Did not catch (yet!)

### Data stored in a continuation

**Code** Where the function is stored in memory, its name, and other meta information, e.g., for debugging.

**Environment** Activation record, including back-pointers as necessary.

**CPU state** The saved registers, most importantly, the **PC** register

**Result** Last result **returned/yielded** by the function

**Termination status** can the continuation be continued

### Operations on continuations

Operation	Semantics
<code>c ← initiate(f...)</code>	Create a new continuation for the call <code>f(...)</code>
<code>c' ← clone(c)</code>	Save execution state for later
<code>resumable(c)</code>	Determine whether resumption is possible
<code>c ← resume(c)</code>	Resume execution of continuation <code>c</code>
<code>retrieveable(c)</code>	Determine whether computation produced a result
<code>x ← retrieve(c)</code>	Obtain the produced result of <code>c</code> <code>generator</code> a generated value <code>continuation</code> a continuation record

Continuations are slightly more general than "generation records" in that they may **yield** continuations, rather than just plain values.

### What can you do with continuations?

- Good old function calls
- Closures
- Generators
- Exceptions
- Coroutines

**REFERENCE** שומר **CONTINUATION**<sup>84</sup> לכל משתני הסביבה. עקרונית, זה כולל גם את המשתנים הגלובליים, אבל אין צורך לשמר אותם בנפרד, כיון שהם חיים. ה **CONTINUATION** אינו משמר את ערכם של המשתנים, לא אלו של הפונקציות בהן הוא מקנן, לא שלו. מכך וחומר, שלא את ערכם של המשתנים הגלובליים.

Generators נקראים סמי-קו-רוטיניות שכן הם יכולים לעשות `yield` רק לפונק' שקרה להם. קו-רוטינה בד"כ תעשה `yield` לכו-רוטינה אחרת.

שאלות בהקשר הזה:

#### ?**garbage collection** מהיבב כמעט?

תשובה (אורשה ע"י פרופ' גיל):  
"אני חושב שהסיבה דומה לשיבת שיש צורך ב GC עם ערך Closure. אנו יושב **Continuation** אגחנו שומרים רפרנסים לסייעיה של הפונקציה, וכך נראה מנגנון סביר לניהול הזיכרון, שכן לא ברור באיזה שלב בritch התוכנית לא יהיה צורך יותר בסביבה הזאת, ובמשתנים שלא שמוקצים על העירמה. בפרט עם **Continuation** ניתן למשם Closure וכאן אותן סיבות בדיקת יהוו רלוונטיותפה (השאלה לגבי Closure מופיע בהרבה מקומות באתר עם תשיבות טובות)".

#### .**CONTINUATION** – תן הסבר ודוגמה ל-

תשובה (לא אושרה ע"י פרופ' גיל):  
Continuation היא אבסטרקציה של closure. בשונה מ closure שמאפשר הפעלת פונקציה גם לאחר שהחיבור אליה נגמר, לדוגמה קריאה לפונקציה שנמצאת בתחום פונקציה אחרת מחוץ לפונקציה החיצונית. Continuation מאפשר לעצור את ריצת הפונקציה ברגע מסוים ולהמשיכו אחר כך. דוגמה לכך זה מנגנון ה generator ב C#. באמצעות `yield` אנו יכולים ליצור את ריצת התוכנית ולהחזיר Continuation לשוכל לשומר אותו ולהמשיך את ריצת התוכנית אח"כ.

#### ?**Closure** – מהו נבדל מ-

Continuation היא אבסטרקציה הלוכדת בתוכה את מצב הביצוע של פונקציה מסוימת, ומאפשרת לשזרו. ה Continuation לפיק כללי יותר מ- Closure במבנה זה, שהוא לא לוcid רק את הקשר בזמן הפעלת הפונקציה, אלא גם בנקודת כלשהו במהלך ביצועו.

#### 88 **Acknowledgment** – **Continuation** מתקשר למثال לימוש ב "Yield Generators קשור ל-

:Generators (כמו גם Coroutines) מאפשר למשם Continuation (C# Yield Break ו- "Return"). יש שני סוגים קריאה ל- Continuation הראשוון, שניין לקרוא לו start – מייצר Continuation חדש. השני, שניין לקרוא לו Resume, לזקח Continuation שנוצר, ממשיך את הביצוע ממנו, עד לפקודה הבאה, המזירה ערך של ה Generator והן ערך של ה Continuation שמננו אפשר להמשיך את הביצוע. הפקודה .Continuation אינה מזירה Yield Break

<sup>84</sup> נלקח מהאתר Safot.net – קישור.

<sup>85</sup> נלקח מהאתר Safot.net – קישור.

<sup>86</sup> נלקח מהאתר Safot.net – קישור.

<sup>87</sup> נלקח מבוחן מועד ב' אביב/מועד א' קיץ 2013.

<sup>88</sup> נלקח מבוחן מועד ב' אביב/מועד א' קיץ 2013.