חלק יבש

1. הסבירו במילותיכן את המונחים סביבה (environment), כריכה (binding), ותחימה (scoping), ואת הקשר ביניהם.
   1. Environment – סביבת העבודה של שגרה, מספקת לשגרה פונקציות, קבועים ומשתנים הזמינים לשגרה ולא הוגדרו על ידה.
   2. Binding – כריכה, מספקת את הקשר בין שם לערך\תא בזכרון. כאשר סביבה מספקת פונקציות, קבועים ומשתנים, היא בעצם מספקת סטים של כריכות.
   3. Scoping – תחימה, כשמה מגדירה את תחום החיים של משתנה, קבוע או פונקציה. לחילופין ניתן לומר כי כאשר מגדירים תחום חיים לנ"ל אזי יש כריכה בין השם המזהה לבין ערכו.
2. הסבירות במילותיכן את המונחים העמסה (overloading) והסתרה (hiding), ואת ההבדלים ביניהם.
   1. Overloading – העמסה, מתאר מצב בו מספר ישויות משתמשות באותו השם. לדוגמה "+" בשפת C אשר יכול לחבר 2 משתנים מסוג int או לחילופין מסוג float. השם אותו השם אך הכריכה שונה בשני המקרים.
   2. Hiding – הסתרה, מתאר את המצב הבא:
      1. ישנה ישות א' בשם X.
      2. בתוך תחום אחר אשר מכיר בישות א' יש ישות ב' גם כן בשם X.

בתחום בו ישות א' חי, שימוש בשם X יוביל לישות ב' – ישות ב' מסתירה את ישות א'.

ההבדל החשוב בין העמסה והסתרה הוא בכך שהעמסה אינה מסתירה ישויות אלא מעניקה

דרך לגשת לישויות שונות שנכרכו לאותו השם.

1. הקוד בהמשך הוא הגדרה חוקית של פונקציה בשפת ML האהובה. הסבירו אילו סביבות (enviroments) ואילו כריכות (bindings) נוצרות בזמן הגדרה (definitions) של הפונקציה boo. תארו או ציירו את ההיררכיה של הסביבות (lookup order).

fun boo boo =

   let  type boo = int

       val boo : boo = boo

       val boo : { boo : boo } = { boo = boo }

   in

        # boo boo

   end

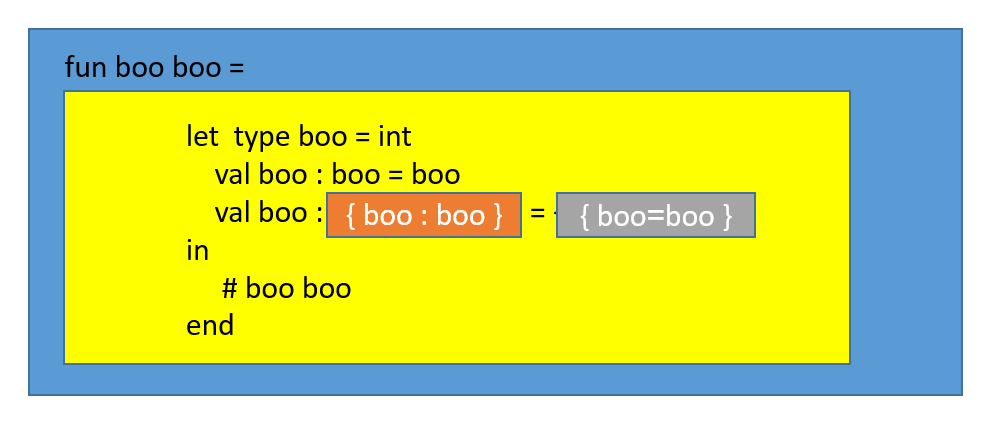
בזמן הגדרת השגרה הנ"ל נוצרות:

* סביבה המתחילה בתחילת הגדרת הפונקציה (fun) ונגמרת בסופה (end).
* כריכה בין השם boo לפונקציה הממומשת בקטע, הכריכה מוגדרת על ידי סימן השוויון בשורה הראשונה של הקוד.

שאר הסביבות הכריכות וההגדרות אינן מורצות בזמן יצירת\קימפול הקוד.

1. הסבירו אילו סביבות (environments) ואילו כריכות (bindings) נוצרות בזמן הפעלה (application) של הפונקציה boo. תארו או ציירו את ההיררכיה של הסביבות (lookup order).

נתאר את הסביבות באמצעות איור:

באיור הנ"ל נבחין ב4 סביבות:

* + 1. הראשית בכחול.
    2. תת סביבה ראשונה בצהוב, מגדירה משתנים וטיפוסים מקומיים. כאן נוצרות הכריכות הבאות:
       1. כירכת טיפוס בשם boo לישות int.
       2. כירכת קבוע בשם boo מטיפוס boo, (1) לישות שנתקבלה על ידי הסביבה הראשית.
       3. כריכת קבוע בשם boo מסוג של טיפוס המוגדר בסביבה חדשה, ולכן נתאר אותה בהמשך.
    3. תת סביבה שניה בכתום, מגדירה טיפוס חדש בעל שדה בשם boo מטיפוס boo
    4. תת סביבה שלישית באפור, אשר יוצרת את הטיפוס האנונימי שהוזכר לעיל.

1. נמספר את המופעים (13 סה"כ) של המילה boo לפי סדר הופעתן בקוד עבור על מופע רשמו מה מציין השם boo. ציינו אם מדובר בהכרזה על שם (declaration) או בהפניה עליו (reference), וכן אם מדובר בהעמסה (overloading) או בהסתרה (hiding). הסבירו במידת הצורך מהי הסביבה (environment) ומהי הכריכה (binding) הרלוונטית.

fun (1) boo (13) boo =

   let  type (2) boo = int

       val (3) boo : (6) boo = (9) boo

       val (4) boo : { (7) boo : (10) boo } = { (11) boo = (12) boo }

   in

        # (5) boo (8) boo

   end;

* 1. Declaration – declaring the binding between the name boo and the functions routine.
  2. Declaration, Overloading – overloads the name boo with declaration 1.
  3. Declaration, Hiding – declares a local entity that hides the name of 13.
  4. Declaration, Hiding – declares a local entity that hides the name of 3.
  5. Reference – the # sign, means that we will access the boo field of the entity that is binded to 8.
  6. Reference – to the type declared by 2.
  7. Declaration, Hiding – declaring a name of a field in a new type. This hides the entity boo from 3.
  8. Reference – access to the boo field of the entity declared by 4.
  9. Reference – access to the boo entity binded to 13.
  10. Reference – to the type declared by 2.
  11. Reference – to the field declared by 7,10.
  12. Reference – to the entity declared by 3.
  13. Declaration, Overloading – declaring a function argument with the same name as the function.

1. החליפו את המופעים (13 סה"כ) של boo בשמות חדשים כלשהם באופן עקבי כך שלא תהיה העמסה או הסתרה של שמות אך תוך שימור משמעות הקוד. יש לרשום את השמות במסגרות הממוספרות.

fun (1) foo (13)  boo =

   let  type (2)  jew = int

       val (3)  man : (6)  jew = (9)  boo

       val (4)  mama : { (7)  small : (10)  jew } = { (11) small = (12)  man }

   in

        # (5)  small (8)  mama

   end;

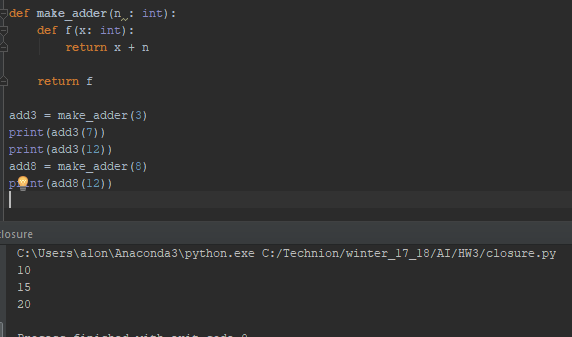
1. מה עושה boo?

בקצרה מחזירה את הערך שהתקבל.

באריכות:

* + 1. מגדירה טיפוס בשם X מסוג INT.
    2. מגדירה ישות בשם Y מטיפוס X עם הערך שנתקבל כארגומנט לפונקציה.
    3. מגדירה ישות Z מטיפוס בעל שדה בשם R מטיפוס X ומישמת אליו את Y.
    4. לבסוף מחזירים את ערך השדה R של של טיפוס Z.

1. מה זה closure, ואיך המושג קשור לפונקציות מקוננות? בפרט, האם יכולה להיות שפה עם פונקציות מקוננות ובלי closure, או בלי פונקציות מקוננות ועם closure? נמקו והביאו דוגמה מבוארת לשימוש ב-closure בשפת Python.

* Closure מאפשר לשגרות לגשת למשתנים מסויימים גם מחוץ לטווח ההכרה שלהם.
* כאשר שגרה א' מקוננת בתוך שגרה ב' לשגרה א' יש גישה למשתנים של שגרה ב'. משתנים אלו כוללים בתוכם גם משתנים מקומיים אשר משוחררים כאשר שגרה ב' מסתיימת. במידה ושגרה א' ממשיכה להתקים (מוחזרת או מועברת לשגרה אחרת), בשגרה מקוננת רגילה תתקבל שגיאה על גישה לערך שאינו חוקי. מנגד, Closure מגדירה כי משתנה מקומי בשימוש של שגרה מקוננת א' ממשיך להתקים כל עוד שגרה א' קיימת.
* יכולה להיות שפה ללא סגירות (Closure) אך עם פונקציות מקוננות, לדגומה – Gnu-C (הרצאה 7 שקף 94).
* לא ניתן להגדיר שפה עם סגירות אך ללא פונקציות מקוננות. ללא פונקציות מקוננות טווח החיים של משתנים מקומיים מאוד ברור, בתוך תחום השגרה, לכן ללא שגרות מקוננות אין צורך בסגירות.
* נשתמש בדוגמה מההרצאה:

השגרה make\_adder יוצרת פונקציה, אך לאחר שmake\_adder חוזרת (ולכן גם המחסנית שלה מתקפלת) נצפה כי גישה למשתנה המקומי n תגרור שגיאה. אך מאחר ופייתון תומכת בסגירות, משתנה זה ממשיך לחיות למרות שהוא מחוץ לטווח ההגדרה שלו.

1. אילו הפשטות מתקדמות (כפי שנלמדו בפרקים 7.2 ו-7.4 בחוברת ההרצאות) קיימות בשפת Kotlin?