# שפת Racket

## הקדמה

Racket היא שפת תכנות, שמבוססת על ניב Scheme של Lisp, ונוצרה בתור פלטפורמה לתכנון ומימוש שפות תכנות. זוהי שפה פונקציונאלית. סביבת העבודה המתאימה לשפה היא DrRacket, אנו נעבוד עם גרסה 7.7. מדריך שימושי של כל היכולות של Racket ניתן למצוא [כאן](https://docs.racket-lang.org/guide/).

### תכנות פונקציונלי

זוהי פרדיגמת תכנות השמה דגש על חישוב ביטוי תוך שימוש בפונקציות ככלי ההפשטה העיקריים. זאת בניגוד לפקודות (Statements) שהן הכלי העיקרי של שפות התכנות מהפרדיגמה הנפוצה יותר, הפרדיגמה האימפרטיבית. הפרדיגמה הפונקציונלית רואה את הפעולה של תוכנית כפונקציה מקלט אל פלט - המחשב מקבל קלט מסוים, מבצע עליו חישוב ומוציא פלט. תוכנית פונקציונלית בנויה בהתאם, כרצף של הצהרות על פונקציות ומשתנים, שלאחריהם מתבצעת קריאה לפונקציה, הנעזרת בפונקציות אחרות, וכן הלאה, עד שבסופו של דבר מוחזר ערך כלשהו, והוא ערך הפלט - תוצאת החישוב.

Racket היא בנוסף שפת פונקציונאלית "טהורה" שבה אין בו שינוי מצב, כלומר אין משתנים או אובייקטים שמשנים את מצבם הפנימי אלא כל עיבוד מידע נעשה על ידי יצירת ערכים חדשים כנדרש. שינוי מצב של המערכת נקרא "תופעות לוואי" (side effect). מכיוון שכל תקשורת של תוכנית עם העולם החיצון (קלט/פלט) משנה מצב של אובייקט ומסיבות נוספות, של נוחות או יעילות, גם שפות פונקציונליות "טהורות" כוללות תופעות לוואי, אך חלקן מרכזות את המבנים בעלי "תופעות הלוואי" במקום מוגדר, לדוגמה, במבנה שנקרא מונדה.

## סביבת עבודה

ב-DrRacket סביבת העבודה מחולקת ל-2. החלק העליון הוא אזור ההגדרות ומכיל הגדרות, פונקציות וכו'. ניתן לשמור חלק זה בקובץ plt ולהשתמש בו שוב ושוב. בתחילת הקובץ צריך לציין באיזה שפה עובדים. אנו בקורס זה נעבוד עם השפה pl ולכן נרשום בתחילת הקובץ:

#lang pl

החלק השני הוא אזור האינטראקציה בו ניתן להכניס קלט ולקבל פלט.

## הערות

כתיבת הערה בשורה בודדת תעשה בעזרת ;; לדוגמא:

;; comment

כתיבת הערה בהמשך לשורת קוד תעשה בעזרת ; לדוגמא:

(+ 5 6) ; 5 + 6 = 11

כתיבת בלוק הערות יעשה בעזרת |# ו- #| לדוגמא:

#|

This program does…

|#

## אובייקטים וטיפוסים

אובייקטים הבסיסיים בשפה הם:

* משתנים בוליאניים: true, false. אפשר גם #t, #f.
* מספרים מכל הסוגים: 1, 0.5, ½, 1+2i.
* מחרוזות: מוקפות בגרשיים, “apple”.
* סמלים: גרש בודד בהתחלה, ‘apple.
* תווים: לפני כל תו יש לרשום #\. לדוגמה, #\a.

לכל אובייקט ב-Racket יש טיפוס נתונים בפני עצמו, כלומר כל אובייקט הוא טיפוס נתונים. איחוד מספר טיפוסים יכול להוות טיפוס כללי יותר. לדוגמה, איחוד של true ו-false זה טיפוס מסוג Boolean. ניתן להגדיר טיפוס חדש כך:

(define-type g ())

ניתן גם לשאול האם אובייקט הוא מטיפוס כלשהו וכן להשוות בין אובייקטים. התשובה תהיה true או false.

(integer? 5.0) ; false

(number? 12) ; true

(eq? 'apple 'apple ) ; true

## ביטויים ואופרטורים

באופן כללי ביטויים ב-Racket מתחלקים לשלושה סוגים:

1. אובייקטים: נכתבים ללא סוגריים (כגון מספרים מחרוזות וכו').
2. ביטויים הכוללים אופרטורים. רושמים קודם את האופרטור ולאחר מכן את האופרנדים. האופרנדים יכולים להיות מוגדרים ברקורסיה. כל ביטוי כזה צריך להיות עטוף בסוגריים. לדוגמה:

(+ 1 2) ; 3

(- (\* 2 3) 1) ; 5

(= 1 1) ; true

(string-append “a” “b”) ; ”ab”

1. ביטויים מיוחדים המשתמשים במילים שמורות, כמו התניות, השמה והכרזה עליהם נדבר בהמשך.

## התניות

### משפט if

כל משפט if מוקף בסוגריים ומקבל שלושה פרמטרים:

1. ביטוי המחזיר ערך בוליאני - true או false.
2. מה להחזיר אם הפרמטר הראשון מחזיר true.
3. מה להחזיר אם הפרמטר הראשון מחזיר false.

(if <condition-expr>

<positive-expr>  
 <negative-expr>)

(if (< 2 3) 10 20) ; 10

### משפט cond

מאוד מזכיר משפט switch ב-Java כאשר יש המון תנאים שרק אחד צריך לפעול אם מתקיים. משפט cond מוקף בסוגריים ומכיל רצף לא מוגבל של סוגריים מרובעים שבכל אחד מהם יש ביטוי המחזיר ערך בוליאני ולאחר מכן מה לבצע אם הביטוי הבוליאני מחזיר true. הבדיקה תתבצע לפי סדר התנאים עד שימצא תנאי המחזיר ערך true כך שתתבצע הפעולה שאחריו ונצא מהמשפט cond. מקובל להוסיף בסוף כל התנאים גם else למקרה שאף אחד לא מחזיר true.

( cond

[< condition > < to do expression >]

[< condition > < to do expression >]\*

[else < else expression >])

( cond

[( eq? 'a 'b) 0]

[( eq? 'a 'c) 1]

[ else 2])

## קבועים

ניתן להגדיר אובייקט קבוע באמצעות אופרטור define, המקשר בין שם לערך.

(define <name> <expression>)

(define PI 3.14)

## פונקציות

בדומה להגדרת קבועים ניתן להגדיר פונקציות באמצעות אופרטור define. רושמים define, לאחר מכן בסוגריים את שם הפונקציה ואת שמות הפרמטרים שהיא מקבלת (אם יש), ולבסוף ביטויים שיתבצעו בכל קריאה לפונקציה.

(define (<function name> <arg1>…)

<expression>)

ב-Racket הפרמטרים לפונקציה הם dynamic typing, כלומר לא צריך לציין איזה טיפוס כל פרמטר צריך להיות אלא הפונקציה תעבוד על כל טיפוס. אולם אם בכל זאת רוצים שהפונקציה תעבוד רק על פרמטרים מטיפוס מסוים, או תחזיר רק ערך מטיפוס מסוים, יש להכריז על הפונקציה לפני המימוש שלה. ההכרזה כוללת את שם הפונקציה (מוקף בנקודותיים), טיפוס כל פרמטר, חץ וטיפוס ערך החזרה.

(:<function name> :<arg1type>… -> return type)

(: pythgoras : Natural Natural -> Number)

(define (pythgoras x y)

(sqrt (+ (\* x x) (\* y y))))

## Local Binding

מאפשר ליצור סביבה מקומית המורכבת ממעטפת וגוף. בשלב המעטפת מגדירים משתנים ובשלב הגוף מבצעים עליהם פעולות.

(let ([<id > <expr >]

[<id > <expr >]\*)

<expr >)

(let ([x 10]

[y 11])

(+ x y))

בשלב המעטפת לא ניתן לבצע השמה עבור משתנים שרק הוגדרו שכן כולם מוגדרים ביחד.

(let ([x 10]

[y (+ x 1)]) ; not valid

(+ x y))

## מחרוזות

אופרטורים לטיפול במחרוזות:

* string-append – מחבר מחרוזות.
* string-ref – מקבל מחרוזות ואינדקס ומחזיר את האיבר במחרוזת באינדקס.
* string=? – מקבל שתי מחרוזות ובודק אם הם שוות.

## רשימות

רשימה ב-Racket מוגדרת באופן רקורסיבי.

* בסיס - רשימה ריקה, מסומנת ב-null או ‘().
* מעבר - כל רשימה היא זוג איברים שבה האיבר השני הוא גם כן רשימה. כלומר כל רשימה מורכבת מזוגות של איברים כך שבכל זוג האיבר הראשון הוא איבר ברשימה והאיבר השני הוא המשך הרשימה. האיבר האחרון ברשימה הוא null.

### יצירת רשימה

ישנן 3 דרכים ליצור רשימה:

1. באמצעות אופרטור cons המקבל איבר ורשימה ומחבר אותם לרשימה אחת כאשר האיבר שקיבל בראש הרשימה.

(cons 1 null)

(cons 1 (cons 2 ‘()))

(cons 1 2) ; not valid in PL

1. באמצעות אופרטור list המקבל רצף של איברים ובונה מהם רשימה רקורסיבית. אין צורך לציין null בסוף.

(list 1 2 3 4)

(cons 1 (list 2 3 4))

1. באמצעות סוגריים שלפניהם מגיע גרש בודד ובתוכם האיברים ברשימה. אין צורך לציין null בסוף.

‘(1 2 3 4)

ניתן גם להגדיר רשימות בתוך רשימות. ברשימה בתוך רשימה אין צורך שיהיה גרש מחוץ לרשימה הפנימית, שכן רשימות מוגדרות ברקורסיה ולכן זה מתווסף אוטומטית.

(cons (list 1 2 3) null) ; equivalent to ‘( (1 2 3) )

### פונקציות על רשימות

* append – מקבל שתי רשימות ומשרשר אותן אחת אחרי השנייה.

(append (list 1 2) (list 3 4)) ; ‘(1 2 3 4)

* first – מקבל רשימה ומחזיר את האיבר הראשון ברשימה. ישנן פונקציות זהות גם ל-second, third, fourth.

(first (list 1 2 3)) ; 1

* rest – מקבל רשימה ומחזיר את אותה הרשימה ללא האיבר הראשון. פקודה שקולה cdr.

(rest (list 1 2 3)) ; '(2 3)

* list-ref – מקבל רשימה ואינדקס i ומחזיר את האיבר ה-i ברשימה. האינדקסים ברשימה מתחיל מ-0.

(list-ref '(1 2 3) 2) ; 3

* list? – מקבל אובייקט ומחזיר true אם ורק אם האובייקט הוא רשימה, אחרת מחזיר false.

(list? (cons 1 (cons 2 '()))) ; #t

* pair? – מקבל אובייקט ומחזיר true אם ורק אם האובייקט הוא צמד.

## הגדרת טיפוסים

ניתן להגדיר טיפוס חדש שלו יש תתי טיפוסים, הנקראים "וריאנטים", ואף להגדיר איזה סוג ערכים כל אחד צריך להכיל. הגדרה זו מאוד דומה לבניית מחלקות בשפה עילית, כאשר הוריאנטים הם כמו בנאים. התחביר לבניית טיפוס כזה הוא:

(define-type <id >

[<id > <type >\*]

[<id > <type >\*] \*)

(define-type Animal

[Snake Symbol Number Symbol]

[Tiger Symbol Number])

(Snake 'Slimey 10 'rats)

(Tiger 'Tony 12)

### בדיקת שייכות

ניתן גם לשאול על אובייקט האם הוא שייך לטיפוס מסוים:

(Animal? (Snake 'Slimey 10 'rats )) ; #t

כאשר רוצים בפרט לבדוק לאיזה וריאנט שייך אובייקט מסוים נשתמש במשפט cases. במשפט מסוג זה יש להכניס אובייקט ולאחר מכן בסוגריים מרובעים לציין את כל הוריאנטים שאנו רוצים לבדוק שייכות אליהם. צריך לקשר את הפרמטרים של האובייקט לשמות כלשהם שניתן להם. לבסוף יש לרשום איזה פעולות לבצע עבור כל אפשרות. ניתן להוסיף בסוף else למקרה שלא שייך לאף אפשרות. התחביר למשפט cases הוא.

(cases <expr >

[<id > (<param >\*) <expr >]

[<id > (<param >\*) <expr >]\*

[else <expr >])

(: animal-name : Animal -> Symbol )

(define (animal-name a)

(cases a

[( Snake n w f) n]

[( Tiger n sc) n]))

(animal-name (Snake 'Slimey 10 'rats )) ; 'Slimey

## רקורסיה

## רקורסית זנב

כאשר הפקודה האחרונה של פונקציה היא קריאה רקורסיבית לעצמה זו רקורסית זנב. היתרון ברקורסיה זו שהיא מונעת את השארת כל הקריאות במחסנית ובמקום זאת בזיכרון המחסנית תהיה רק הקריאה הבאה שצריכה להתבצע.

רקורסית זנב ללא פרמטרים אפשר להחליף באיטרציה (תכנות מתקדם, מצגת 6, שקופית 27).

## match

זוהי פונקציה שמקבלת ערך ומחפשת תבנית שמתאימה לערך זה. עבור כל תבנית יש להגדיר איזה ערך להחזיר אם אכן הערך מתאים לתבנית זו. התחביר של פונקציה זו הוא:

(match value

[pattern1 result1]

[pattern2 result2]

...)