# שפות תכנות

## דקדוק חסר הקשר

כדי להגדיר שפת תכנות צריך קודם להגדיר איזה ביטויים שייכים לשפה. נעשה זאת באמצעות דקדוק חסר הקשר מצורת BNF (Backus-Naur Form). דוגמה לדקדוק מצורה זו הוא:

<AE> ::= <num>

| <AE> + <AE>

| <AE> - <AE>

בדוגמה לעיל <AE> היא מילה לא טרמינלית, כלומר ניתן לגזור ממנה אחד מבין שלושה ביטויים. <num> הוא מספר כלשהו, נתייחס אליו בתור אוסף של אינסוף תווים. מילה שייכת לשפה של דקדוק אם יש מספר סופי של צעדים גזירה שבסופם מתקבלת המילה והיא מורכבת רק מתווים סופיים (לא טרמינליים). שפת הדקדוק בדוגמה היא כל ביטוי אריתמטי עם האופרטורים חיבור וחיסור בלבד. נראה כיצד ניתן לקבל את הביטוי 1-2+3:

<AE> ; ==>

<AE> + <AE> ; (2) ==>

<AE> + <num> ; (1) ==>

<AE> - <AE> + <num> ; (3) ==>

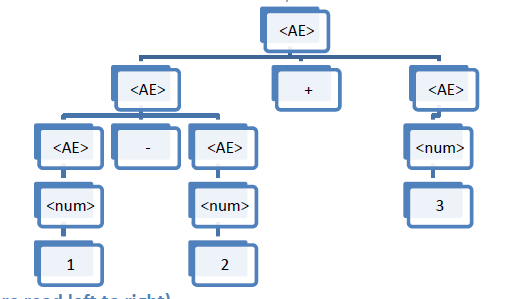
<AE> - <AE> + 3 ; (num) ==>

<num> - <AE> + 3 ; (1) ==>

<num> - <num> + 3 ; (1) ==>

1 - <num> + 3 ; (num) ==>

1 - 2 + 3 ; (num)

לכל מילה בשפה ניתן להראות את אופן בגזירה באמצעות עץ גזירה. המילה שמתקבלת מעץ כזה היא שרשור העלים משמאל לימין. לדוגמה עץ הגזירה המתאים עבור הגזירה של 1-2+3 לעיל יראה כך:

## רב משמעות - Ambiguity

כאשר יש מספר דרכים בדקדוק לגזור את אותה המילה נאמר שבדקדוק קיימת תופעה של רב-משמעות. תכונה זו בעייתית מאוד ולא רצויה בדקדוק, שכן רצף סדר הגזירות מראה לנו את סדר הפעולות על המילה שעבור אותה מילה יכול להיות בעל משמעות שונה. ישנם מספר שיטות למנוע בעיית רב משמעות:

1. הוספה של מילים לא טרמינליות לדקדוק כך שבכל כלל גזירה יש מילה לא טרמינלית אחת בלבד.

<AE> ::= <AE>

| <num> + <AE>

| <num> - <AE>

1. הוספת סוגריים לכלל הגזירה.

<AE> ::= <num>

| { <AE> + <AE> }

| { <AE> - <AE> }

1. בכל כלל גזירה האופרטורים יהיו בהתחלה ולא באמצע.

<AE> ::= <AE>

| + <AE> <AE>

| - <AE> <AE>

## Parser

נרצה לכתוב parser שמקבל כקלט מחרוזת ובודק אם היא מילה בשפה, ואם כן מחזיר עץ תחביר אבסטרקטי עבור המילה שקיבל. לדוגמה עבור הביטוי 3+4 נרצה שיחזיר עץ מהצורה:

(Add (Num 3) (Num 4) )

לשם כך ה-parser צריך לבצע שלושה פעולות:

1. קריאת המחרוזת לתוך מבנה נתונים בסיסי המייצג את הדקדוק BNF של השפה.
2. ניתוח תחבירי של המחרוזת: האם נגזר מהדקדוק של השפה ולפי אילו רצף של כללי גזירה.
3. אם שייך לשפה יש להמיר את רצף כללי הגזירה לעץ תחביר אבסטרקטי.

כדי לעשות זאת ב-Racket נגדיר טיפוס חדש באמצעות define-type המייצג את הדקדוק BNF. כל כלל גזירה הוא וריאנט בטיפוס החדש. לאחר מכן נגדיר פונקציה שמקבלת מחרוזת ומחזירה את הטיפוס החדש שהגדרנו.

### דוגמה

דוגמה ל-parser עבור שפה של ביטויים אריתמטיים עם האופרטורים חיבור וחיסור בלבד:

(define-type AE

[Num Number]

[Add AE AE]

[Sub AE AE])

(: parse : String -> AE)

(define (parse code)

(parse-sexpr (string->sexpr code)))

(: parse-sexpr : Sexpr – >AE)

(define (parse-sexpr sxp)

(match sxp

[(number : n) (Num n)]

[(list ‘+ l r) (Add (parse-sexpr l) (parse-sexpr l))]

[(list ‘- l r) (sub (parse-sexpr l) (parse-sexpr l))]

[else (error ‘parse-sexpr “bad syntax in ~s” sxp]))

(test (parse “{ + 3 4}”) => (Add (Num 3) (Num 4))(

## Evaluation

לאחר שבנינו parser צריך לבנות evaluator. ה-evaluator הוא פונקציה שמקבלת עץ תחביר אבסטרקטי של ביטוי בשפה שפלט ה-parser, ומחזיר את ההערכה של השפה על אותו ביטוי. לדוגמה, עבור שפת כל הביטויים האריתמטיים עם האופרטורים חיבור וחיסור, הפעלת פונקציית ההערכה על עץ התחביר האבסטרקטי הבא צריכה להחזיר 7.

(Add (Num 3) (Num 4))

### Compositionality

תכונה חשובה של דקדוק היא Compositionality, שאומרת שהמשמעות של כל ביטוי מורכב E תלויה במבנה התחבירי של E ובמשמעויות החלקים הפשוטים של E. במילים אחרות, כדי להעריך כל ביטוי מורכב E צריך רק לדעת להעריך ביטויים פשוטים יותר שמהם מורכב E שנמצאים ברמת הגזירה האחרונה של E, וביחס בין כל ביטויים פשוטים אלו. לא צריך לדעת גזירות ברמות עמוקות יותר של הביטוי.

ב-Racket תכונה זו מאפשרת לנו ליצור evaluator המעריך ביטוי מורכב בקלות על ידי הפעלה רקורסיבית של ה-evaluator על הבנים שלו ועם הוספת חישובים קלים בהתאם ליחס שבין הבנים.

### דוגמה

דוגמה ל-evaluator עבור שפה של ביטויים אריתמטיים עם האופרטורים חיבור וחיסור בלבד (המשך דוגמה מסעיף קודם):

(: eval : AE -> Number)

(define (eval expr)

(cases expr

[(Num n) n]

[(Add l r) (+ (eval l) (eval r))]

[(Sub l r) (- (eval l) (eval r))]))

כעת ניתן לחבר את ה-parser מסעיף קודם עם ה-evaluator לפונקציה אחת run שמקבלת מחרוזת המייצגת קוד ומחזירה את ההערכה הסופית של הקוד. פונקציה זו היא הממשק היחיד של המשתמש אל השפה שיצרנו.

(: run : String -> Number)

(define (run str)

(eval (parse str)))

(test (run "3") => 3)

(test (run "{+ 3 4}") => 7)

(test (run "{+ {- 3 4} 7}") => 6)

### פיצול ה-parser וה-evaluator

עקרון בסיסי בשפות תכנות הוא פיצול הקוד לשני שכבות, אחד מנתח את הקוד ומפרמל אותו במידה והוא אכן תקין (סינטקס), והשני מעריך את הקוד (סמנטיקה). לשכבה הראשונה אחראי ה-parser ולשכבה השנייה ה-evaluator. הפיצול בין שני תפקידים אלו הוא מאוד חשוב מפני שמאפשר לנו לבצע שינויים בשפה בקלות. שינוי של חלק אחד לא אמור לפגוע בחלק השני.

## החלפה (יצירת משתנים)

פעמים רבות כשאנו מתכנתים אנו נתקלים בצורך להשתמש בביטויים שחוזרים על עצמם. במקרים כאלו נרצה לתת לביטוי החוזר על עצמו כינוי קצר ומהותי, כך שלהבא בכל מקום בו צריך להופיע הביטוי נרשום את הכינוי שלו בלבד. לשיטה זו יש מספר יתרונות:

* יעילות - לא צריך לחשב את הביטוי מספר פעמים אלא מספיק פעם אחת בלבד. בכל מקום בו מופיע הכינוי נחליף אותו בתוצאת החישוב.
* מונע שכפול של קוד. לשכפול קוד יש חסרונות רבים, העיקרי והנפוץ שבהם הוא שמשנים קוד במקום אחד ושוכחים לשנות במקום אחר.
* מאפשר למתכנת לבטא את עצמו בצורה טובה יותר ולהעביר מידע על ידי נתינת שמות איכותיים לביטויים.
* קוד פשוט וקריא יותר.

נרצה ששפת התכנות תאפשר למתכנת להגדיר כינויים לביטויים. כינויים אלו נקראים "מזהים" (Identifiers) או "משתנים" (Variables). בשפה שנבנה בקורס זה נגדיר את האפשרות לתת שמות לביטויים על ידי אופרטור with המקבל:

1. רשימה של מזהה (identifier) וביטוי בשפה שיכנס לתוך המזהה.
2. ביטוי נוסף בשפה שמשתמש במזהה.

לדוגמה, בקוד הבא למזהה x נכנס הביטוי 4+2 ולאחר מכן מחשבים .

(with (x (+ 4 2))

(\* x x))

כדי לעשות זאת ב-Racket נצטרך לשנות את הגדרת טיפוס הנתונים, ה-parser וה-evaluator. נעבור על כל חלק ונתאר איזה שינויים יש לבצע. הדוגמה שנשתמש בה בכל החלקים תהיה שפה של ביטויים אריתמטיים עם האופרטורים חיבור, חיסור, כפל וחילוק, המאפשרת לתת מזהה לביטויים.

### טיפוס הנתונים

נצטרך להוסיף שני וריאנטים לטיפוס נתונים המייצג ביטוי בשפה. וריאנט אחד כדי לבנות מזהה המקבל symbol, ווריאנט שני המקבל symbol ושני ביטויים, הראשון נכנס למזהה והשני משתמש במזהה.

(define-type WAE

[Num Number]

[Add WAE WAE]

[Sub WAE WAE]

[Mul WAE WAE]

[Div WAE WAE]

[Id Symbol]

[With Symbol WAE WAE])

### Parser

נוסיף בתוך הפונקציה match תבנית המתאימה למחרוזת המייצגת ביטוי המתחיל ב-with. במידה והמחרוזת מייצגת ביטוי חוקי נשתמש בווריאנט שהגדרנו לעיל כדי להחזיר את הביטוי המתאים. אולם כיוון שזה ביטוי יחסית מורכב לביטויים אחרים בשפה ומכיל הרבה סוגריים, נרצה להחזיר שגיאה מתאימה במידה וביטוי שמתחיל ב-with אינה מייצגת ביטוי תקין.

(: parse-sexpr : Sexpr -> WAE)

(define (parse-sexpr sexpr)

(match sexpr

[(number: n) (Num n)]

[(cons 'with more)

(match sexpr

[(list 'with (list (symbol: name) named\_expr) body)

(With name (parse-sexpr named\_expr) (parse-sexpr body))]

[else (error 'parse-sexpr "bad ‘with' syntax in ~s" sexpr)])]

[(list '+ l r) (Add (parse-sexpr l) (parse-sexpr r))]

[(list '- l r) (Sub (parse-sexpr l) (parse-sexpr r))]

[(list '\* l r) (Mul (parse-sexpr l) (parse-sexpr r))]

[(list '/ l r) (Div (parse-sexpr l) (parse-sexpr r))]

[else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))

### החלפה - Substitutions

כדי ליצור evaluator עבור ביטוי מהצורה:

{With id WAE1 WAE2}

צריך קודם ליצור פונקציה שתחליף את כל המופעים של id ב-WAE2 בהערכה של WAE2, ולאחר מכן להעריך כרגיל את הביטוי שמתקבל. כלומר כל ההערכה לביטוי עם with צריך להיות מהצורה הבאה:

eval ( (with (id WAE1) WAE2) ) = eval (subst (WAE2, id, eval (WAE1) ) )

אמנם לא ניתן לבצע את ההחלפה בצורה פשוטה, שבה כל הופעה של id מוחלפת ב-WAE1, שכן יכולים להיות מקרים מורכבים יותר שבהם נקבל שגיאה. לדוגמה, בתוך WAE2 יש ביטוי with מקונן עם id. כדי לעשות החלפה בצורה הנכונה עלינו להגדיר ראשית מספר הגדרות.

* Binding instance - מופע של מזהה כאשר מצהירים עליו לראשונה.

{with {x 5}

{+ x {with {x 3}

x}}}

* Scope - זהו כל הביטוי WAE2 שבו משתמשים במזהה.

{with {x 5}

{+ x {with {x 3}

x}}}

* Bound instance - מופע של מזהה נקרא bound אם הוא מופיע בתוך scope של מזהה ואינו binding instance של מזהה פנימי עם אותו שם.

{with {x 5}

{+ x {with {x 3}

x}}}

* Free instance - מופע של מזהה נקרא free אם הוא לא bound instance ולא binding instance.

פונקציית ה-subst תקבל ביטוי WAE2 שבו יש להחליף את המזהים, מזהה id וביטוי WAE1 שאותו יש להחליף במזהה. פונקציית ה-subst תחליף את כל המופעים של המזהה בתוך ה-scope שאינם בתוך scope של מזהה פנימי עם אותו שם. בניסוח אחר, תחליף את כל המופעים החופשיים בתוך WAE2:

(: subst : WAE Symbol WAE -> WAE)

(define (subst expr from to) ; from is the id, and to is the expression to switch with id

(cases expr

[(Num n) expr]

[(Add l r) (Add (subst l from to) (subst r from to))]

[(Sub l r) (Sub (subst l from to) (subst r from to))]

[(Mul l r) (Mul (subst l from to) (subst r from to))]

[(Div l r) (Div (subst l from to) (subst r from to))]

[(Id name) (if (eq? name from) to expr)]

[(With bound-id named-expr bound-body)

(With bound-id

(subst named-expr from to)

(if (eq? bound-id from)

bound-body ; don't do anything.

(subst bound-body from to)))]))

### Evaluation

כמו שציינו לעיל, פונקציית ה-eval על ביטוי מהצורה (With id WAE1 WAE2) צריכה להפעיל את פונקציית subst על WAE2 שבה נחליף כל מופע של id ב-eval של WAE2. הסיבה שלא מחליפים בין id ל-WAE2 היא כדי שלא נצטרך לחשב את WAE2 מספר פעמים במקרה ש-id מופיע כמה פעמים ב-WAE1. אולם כיוון שמ-eval חוזר מספר שהוא לא ביטוי WAE, נפעיל על הערך שחזר את הבנאי של Num. על הביטוי שחזר מ-subst נפעיל רקורסיבית את eval. אם ה-eval מקבל symbol נחזיר שגיאה, שכן כנראה יש בעיה בהגדרת המשתנה של המשתנה, שאילו היה מוגדר טוב היינו עושים לו subst לפני ה-eval.

(: eval : WAE -> Number)

(define (eval expr)

(cases expr

[(Num n) n]

[(Add l r) (+ (eval l) (eval r))]

[(Sub l r) (- (eval l) (eval r))]

[(Mul l r) (\* (eval l) (eval r))]

[(Div l r) (/ (eval l) (eval r))]

[(With bound-id named-expr bound-body)

(eval (subst bound-body

bound-id

(Num (eval named-expr))))]

[(Id name) (error 'eval "free identifier: ~s" name)]))

(: run : String -> Number)

(define (run str)

(eval (parse str)))

(test (run "5") => 5)

(test (run "{with {x {+ 5 5}} {with {y {- x 3}} {+ y y}}}") => 14)

(test (run "{with {x 5} {with {x x} x}}") => 5)

(test (run "{with {x 1} y}") =error> "free identifier")

## הגדרת פונקציות

נרצה ששפת התכנות תאפשר למשתמש להגדיר פונקציות שבהם הוא בוחר מהם הפרמטרים שהפונקציה מקבלת ומכניס ביטוי בשפה שמשתמש בפרמטרים אלו. לאחר הגדרת הפונקציה צריך לאפשר למשתמש לקרוא לפונקציה שהגדיר עם ערכים תקינים שיבחר.

* בשפה שנבנה בקורס זה כדי להגדיר פונקציה יש להשתמש באופרטור fun המקבל id, שהוא פרמטר לפונקציה, וביטוי בשפה המשתמש ב-id.

(fun (x) (\* x x))

* כדי לקרוא לפונקציה יש להשתמש באופרטור call המקבל ביטוי בשפה המייצג פונקציה וביטוי נוסף שיכנס לתוך id בפונקציה.

(call (fun (x) (\* x x))

5)

כדי שיהיה אפשר לקרוא לפונקציה מספר פעמים בלי שנצטרך לרשום את הפונקציה כל פעם מחדש, יש להכניס את הפונקציה לתוך מזהה ולתת לו שם באמצעות with.

(with (sqr (fun (x) (\* x x)))

(+ (call sqr 5)

(call sqr 6))) ; return 61

כדי לעשות זאת ב-Racket נצטרך לשנות את הגדרת טיפוס הנתונים, ה-parser, פונקציית ה-subst וה-evaluator. נעבור על כל חלק ונתאר איזה שינויים יש לבצע. הדוגמה שנשתמש בה בכל החלקים תהיה שפה של ביטויים אריתמטיים עם האופרטורים חיבור, חיסור, כפל וחילוק, המאפשרת לתת מזהה לביטויים ולהגדיר פונקציות. נקרא לשפה זו FLANG.

### טיפוס הנתונים

נצטרך להוסיף שני וריאנטים לטיפוס נתונים המייצג ביטוי בשפה. וריאנט אחד בשביל fun והשני בשביל call

(define-type FLANG

[Num Number]

[Add FLANG FLANG]

[Sub FLANG FLANG]

[Mul FLANG FLANG]

[Div FLANG FLANG]

[Id Symbol]

[With Symbol FLANG FLANG])

[Fun Symbol FLANG]

[Call FLANG FLANG])

### Parser

נוסיף בתוך הפונקציה match תבנית המתאימה למחרוזת המייצגת ביטויים המתחיל ב-fun ו-call. במידה והמחרוזת מייצגת ביטוי חוקי נשתמש בווריאנט שהגדרנו לעיל כדי להחזיר את הביטוי המתאים. אולם כיוון שאלו ביטויים יחסית מורכבים לביטויים אחרים בשפה ומכילים הרבה סוגריים, נרצה להחזיר שגיאה מתאימה במידה וביטוי אינו תקין.

(: parse-sexpr : Sexpr -> FLANG)

(define (parse-sexpr sexpr)

(match sexpr

[(number: n) (Num n)]

[(cons 'with more)

(match sexpr

[(list 'with (list (symbol: name) named\_expr) body)

(With name (parse-sexpr named\_expr) (parse-sexpr body))]

[else (error 'parse-sexpr "bad ‘with' syntax in ~s" sexpr)])]

[(cons 'fun more)

(match sexpr

[(list 'fun (list (symbol: name)) body)

(Fun name (parse-sexpr body))]

[else (error 'parse-sexpr "bad `fun' syntax in ~s" sexpr)])]

[(list 'call fun arg) (Call (parse-sexpr fun) (parse-sexpr arg))]

[else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))

[(list '+ l r) (Add (parse-sexpr l) (parse-sexpr r))]

[(list '- l r) (Sub (parse-sexpr l) (parse-sexpr r))]

[(list '\* l r) (Mul (parse-sexpr l) (parse-sexpr r))]

[(list '/ l r) (Div (parse-sexpr l) (parse-sexpr r))]

[else (error 'parse-sexpr "bad syntax in ~s" sexpr)]))

(test (parse “(with (sqr (fun (x) (\* x x)))

(+ (call sqr 5)

(call sqr 6)))”)

* (With ‘sqr

(Fun ‘x (Mul) (Id ‘x) (Id ‘x)))

(Add (Call (Id ‘sqr) (Num 5))

(Call (Id ‘sqr) (Num 6)))))

### החלפה - Substitutions

פונקציית ה-subst תקבל ביטוי expr שבו יש להחליף את המזהים, סימבול (from) המייצג פרמטר וביטוי נוסף בשפה (to) שאותו יש להחליף במזהה. עבור ביטויים מסוג Call פונקציית ה-subst פשוט תחליף את כל המופעים של הפרמטר from ב-to, כמו שהגדרנו עבור האופרטורים האריתמטיים הרגילים. עבור ביטויים מסוג Fun, יש לבדוק האם המופע של הפרמטר הוא bound ורק אם לא להחליף, באופן דומה למה שלמדנו ב-With.

(: subst : FLANG Symbol FLANG -> FLANG)

(define (subst expr from to)

(cases expr

[(Num n) expr]

[(Add l r) (Add (subst l from to) (subst r from to))]

[(Sub l r) (Sub (subst l from to) (subst r from to))]

[(Mul l r) (Mul (subst l from to) (subst r from to))]

[(Div l r) (Div (subst l from to) (subst r from to))]

[(Id name) (if (eq? name from) to expr)]

[(With bound-id named-expr bound-body)

(With bound-id

(subst named-expr from to)

(if (eq? bound-id from)

bound-body

(subst bound-body from to)))]

[(Call l r) (Call (subst l from to) (subst r from to))]

[(Fun bound-id bound-body)

(if (eq? bound-id from)

expr ; don't do anything.

(Fun bound-id (subst bound-body from to)))]))))

### Evaluation

נשים לב שבשונה מהשפה WAE שבה eval תמיד החזיר מספר, כאן הפעלה של eval על ביטוי מסוג Fun, המייצג פונקציה, אינה מחזירה מספר אלא את הפונקציה עצמה. לכן בשפה FLANG פונקציית ה-eval צריכה להחזיר ביטוי שמייצג או מספר (Num Number) או פונקציה (Fun Symbol FLANG). כיוון ש-FLANG עצמו הוא ביטוי שעוטף גם מספר וגם פונקציה, נגדיר ש-eval תחזיר ביטוי מסוג FLANG. בפונקציה run נמיר ביטוי זה אל מספר.

נשים לב שכעת eval על ביטוי אריתמטי לא יוכל לפעול כך: (+ (eval l) (eval r)), שכן eval מחזיר FLANG ולא מספר, ולא ניתן להפעיל פעולה אריתמטית על שני ביטויי FLANG. כדי לפתור בעיה זו, נוסיף פונקציית עזר, arith-op, שתקבל פעולה אריתמטית על מספרים ושני ביטויי FLANG, ותחזיר ביטוי FLANG מהצורה Num n, כאשר n הוא מספר כלשהו. לשם כך היא תיעזר בפונקציה פנימית שמקבלת ביטוי Num ומחזירה את המספר שהוא מייצג. נשים לב ששני ביטויי ה-FLANG ש-arith-op מקבלת צריכים להיות מסוג Num ולא Fun, כדי שנוכל לבצע ביניהם פעולה אריתמטית. אם זה לא המצב נחזיר שגיאה.

(: arith-op : (Number Number -> Number) FLANG FLANG -> FLANG)

(define (arith-op op expr1 expr2)

(: Num->number : FLANG -> Number)

(define (Num->number e)

(cases e

[(Num n) n]

[else (error 'arith-op "expects a number, got: ~s" e)]))

(Num (op (Num->number expr1) (Num->number expr2))))

נוסיף לפונקציית ה-eval את שני הוריאנטים החדשים של Fun ו-Call. עבור ביטוי מסוג Fun נחזיר את הביטוי כולו. עבור ביטוי מסוג “Call FLANG FLANG”, אם הביטוי FLANG הראשון הוא פונקציה, נחליף בפונקציה את הפרמטר בתוצאה של evalעל הביטוי FLANG השני כדי לחסוך בחישובים. אך אם הביטוי FLANG הראשון הוא לא פונקציה, נחזיר פשוט שגיאה.

(: eval : FLANG -> FLANG) ;

(define (eval expr)

(cases expr

[(Num n) expr]

[(Add l r) (arith-op + (eval l) (eval r))]

[(Sub l r) (arith-op - (eval l) (eval r))]

[(Mul l r) (arith-op \* (eval l) (eval r))]

[(Div l r) (arith-op / (eval l) (eval r))]

[(With bound-id named-expr bound-body)

(eval (subst bound-body

bound-id

(eval named-expr)))] ; no (Num ...)

[(Id name) (error 'eval "free identifier: ~s" name)]

[(Fun bound-id bound-body) expr]

[(Call (Fun bound-id bound-body) arg-expr) ; nested pattern

(eval (subst bound-body

bound-id

(eval arg-expr)))]

[(Call something arg-expr)

(error 'eval "`call' expects a function, got: ~s" something)]))

### run

הפונקציה run שומרת את התוצאה שחוזרת מ-eval. אם הערך שחזר הוא ווריאנט מסוג Num, המייצג את הערך הסופי, מחזיר את המספר. אך אם הווריאנט הוא ביטוי מסוג Fun המייצג פונקציה, מחזירה שגיאה.

(: run : String -> Number)

(define (run str)

(let ([result (eval (parse str))])

(cases result

[(Num n) n]

[else (error 'run "evaluation returned a non-number: ~s" result)])))

## Substitution Cache

בשפה שבנינו עד כה, יצירת משתנים וקריאה לפונקציה מתבצעים בצורה מאוד לא יעילה. שכן עבור כל יצירת משתנה או קריאה לפונקציה, משתמשים בפונקציה subst שעוברת על כל המשך הקוד ומחליפה ערך נתון בכל המופעים החופשיים של מזהה (משתנה או הפרמטר). דרך יותר יעילה הייתה להעביר רשימה, המשמשת כ-cache, של כל המזהים שנתקלנו בהם עד כה ואת הערכים המתאימים להם, כך שבכל פעם שניתקל במזהה נחליף אותו בערך המתאים לו מתוך ה-cache. השימוש ב-cache צריך להיות כך שאם נגדיר מזהה x ובתוך ה-scope שלו נגדיר שוב מזהה בשם x, אזי רק בתוך ה-scope הפנימי תתבצע דריסה של הערך של x, אך מחוץ ל-scope הפנימי הערך של x יחזור להיות לפי ההגדרה הראשונה.

כדי להוסיף מימוש יעיל זה לשפה שלנו, צריך קודם להגדיר טיפוס חדש המשמש מבנה נתונים ל-cache.

### טיפוס הנתונים ל-cache

טיפוס הנתונים צריך להיות רשימה של זוגות, כך שכל זוג הוא מזהה וערך. נייצג מזהה באמצעות symbol וערך באמצעות ביטוי FLANG שניתן להעריך אותו באמצעות eval. אנו צריכים שלושה פונקציות שימושיות המקבלות את טיפוס הנתונים של ה-cache כפרמטר:

1. empty - מקבל SubstCache ומחזיר cache ריק.
2. extend - מקבל SubstCache וזוג של מזהה וערך, ומשרשר זוג זה ל-SubstCache.
3. lookup - מקבל מזהה ו- SubstCacheובודק אם המזהה נמצא ברשימה. במידה וכן, מחזיר את הערך של המזהה, אחרת מחזיר הודעת שגיאה. נשתמש בפונקציה מובנית ב-Racket הנקראת “assq”, המקבלת ערך ורשימה של זוגות, ומחזירה את הזוג הראשון שבו נמצא הערך. אם לא נמצא הערך באף זוג ברשימה מחזירה #f.

(define-type SubstCache = (Listof (List Symbol FLANG)))

(: empty-subst : SubstCache)

(define empty-subst null)

(: extend : Symbol FLANG SubstCache -> SubstCache)

(define (extend id expr sc)

(cons (list id expr) sc))

(: lookup : Symbol SubstCache -> FLANG)

(define (lookup name sc)

(let ([cell (assq name sc)])

(if cell

(second cell)

(error 'lookup "no binding for ~s" name))))

### Evaluation

לפי המימוש שתיארנו אין צורך להשתמש יותר בפונקציה subst וניתן למחוק אותה. השינוי היחידי הנוסף, מלבד הגדרת טיפוס נתונים, הוא לשנות את פונקציית ה-eval. פונקציה זו עתה מקבלת ביטוי FLANG ו-SubstCache שבה שמורים כל המזהים והערכים שלהם.

השינויים המרכזיים בתוך הפונקציה eval הם:

* בכל קריאה רקורסיבית ל-eval יש להוסיף את ה-SubstCache שהפונקציה קיבלה.
* עבור הווריאנט With, מוסיפים ל-cache זוג נוסף של המזהה שהוגדר ב-with וההערכה שלו.
* עבור הווריאנט Id, נחפש את המזהה ב-cache באמצעות lookup. אם המזהה נמצא ב-cache הפונקציה תחזיר את הערך שלו, אך אם לא נמצא תחזיר הודעת שגיאה.
* עבור קריאה לפונקציה Call, קודם נעריך את הביטוי המייצג את הפונקציה כדי לבדוק שהוא תקין, כלומר הוא מסוג Fun. במידה והפונקציה תקינה, נוסיף ל-cache זוג נוסף של המזהה שהוגדר ב-Fun ואת הערך שנשלח כפרמטר ב-Call.

(: eval : FLANG SubstCache -> FLANG)

(define (eval expr sc)

(cases expr

[(Num n) expr]

[(Add l r) (arith-op + (eval l sc) (eval r sc))]

[(Sub l r) (arith-op - (eval l sc) (eval r sc))]

[(Mul l r) (arith-op \* (eval l sc) (eval r sc))]

[(Div l r) (arith-op / (eval l sc) (eval r sc))]

[(With bound-id named-expr bound-body)

(eval bound-body

(extend bound-id (eval named-expr sc) sc))]

[(Id name) (lookup name sc)]

[(Fun bound-id bound-body) expr]

[(Call fun-expr arg-expr)

(let ([fval (eval fun-expr sc)])

(cases fval

[(Fun bound-id bound-body)

(eval bound-body

(extend bound-id (eval arg-expr sc) sc))]

[else (error 'eval "`call' expects a function, got: ~s" fval)]))]))

### run

כמו שהגדרנו run בסעיף הקודם, אלא של-eval צריך להוסיף cache ריק באמצעות הפונקציה empty-subst.

(: run : String -> Number)

(define (run str)

(let ([result (eval (parse str) empty-subst)])

(cases result

[(Num n) n]

[else (error 'run "evaluation returned a non-number: ~s" result)])))

## Dynamic and Static Scopes

בקוד יש הרבה מזהים (משתנים ופרמטרים), ויכול להתבצע דריסה של מזהים בתוך קוד פנימי יותר. כאשר אנו קוראים לפונקציה שמשתמשת במזהים שהוגדרו לפני הגדרת הפונקציה באמצעות with, כמו בקוד למטה, יש שתי שיטות כיצד להתייחס למזהים אלו:

Static scope - מגדירים את ערכי המזהים החופשיים בזמן הגדרת הפונקציה.

Dynamic scope - מגדירים את ערכי המזהים החופשיים בזמן הקריאה לפונקציה.

(run "{with {x 3}

{with {f {fun {y} {+ x y}}}

{with {x 5}

{call f 4}}}}")

לדוגמה, בקוד לעיל לפי static scope צריך להחזיר 7, כיוון שבזמן הגדרת f אנו בעצם מגדירים אותו להיות (+ 3 y), כיוון ש-x בעת ההגדרה שווה ל-3. אולם לפי dynamic scope הקוד יחזיר 9, כיוון שהפונקציה f מוגדרת להיות (+ x y), כאשר ערכו של x נקבע בזמן הקריאה הפונקציה, שהוא 5.

בשפה שאנו מגדירים תמיד נרצה להשתמש בשיטת Static scope כיוון ש-Dynamic scope יכולה להביא לטעויות אליהן לא התכוון מתכנת השפה. Racket מוגדרת לפי Static scope אולם ניתן להחליף ל-Dynamic scope אם משתמשים בשפה #lang pl dynamic. נדגים מספר בעיות בשיטה זו:

#lang pl dynamic

(define x 123)

(define (getx) x)

(define (f1 x) (getx))

(define (f2 y) (getx))

(test (getx) => 123)

(test (let ([x 456]) (getx)) => 456)

(test (f1 999) => 999)

(test (f2 999) => 123) # problem: two equal function with different output to same input

(define (foo x)

(define (helper)

(+ x 1))

helper) # foo returns a function that calculates (+ x 1)

(test ((foo 0)) => 124) # problem: intended to return 1 but returned 124

(define (add x y)

(+ x y))

(test (let ([+ \*]) (add 6 7)) => 42) # problem: can change function outside its definition

הדוגמה האחרונה היא הכי בעייתית אולם יש בה יתרון המאפשר למתכנת להתאים אליו אישית כל פונקציה, אפילו כאלו שאינו יכול לגשת למימוש שלהם.

השפה FLANG שהגדרנו בסעיף ו' עם פונקציית subst היא לפי Static scope, אולם השפה היותר יעילה שהגדרנו בסעיף ז' עם cache היא לפי Dynamic scope. נרצה לשנות את השפה בסעיף ז' כך שתוגדר לפי שיטת Static scope. הדרך לעשות זאת היא באמצעות הגדרה "סביבה" לפונקציה בעת הגדרתה.

## סביבה לפונקציה

אנו רוצים שהשפה שהגדרנו בסעיף ז', עם cache, תוגדר לפי שיטת Static scope. לשם כך פונקציית eval על ביטוי מסוג Fun, המייצג פונקציה, צריכה להחזיר את הפונקציה אך בנוסף גם סביבה שהיא ה-cache במצבו הנוכחי של הגדרת אותה פונקציה. נקרא לשילוב של פונקציה עם סביבה Closure. עבור ביטוי מסוג Call, eval תוסיף את הזוג של הפרמטר והערך שלו **לסביבה של הפונקציה בעת הגדרתה ולא לסביבה במצב נוכחי.** כך ההערכה של הפונקציה תיעשה לפי הסביבה בזמן ההגדרה של הפונקציה ולא לפי מצב ה-cache בזמן הקריאה לפונקציה.

### הגדרת טיפוסים

נשים לב שעבור ביטוי Fun, eval צריכה להחזיר Closure שאינו ביטוי בשפה FLANG. לכן נגדיר טיפוס חדש בשם VAL המייצג את כל הערכים ש-eval יכולה להחזיר, שהם מספר ו-Closure.

בנוסף נגדיר טיפוס ENV המייצג סביבה. בדומה ל-cache מסעיף קודם, אנו מעוניינים ליצור סביבה ריקה, להוסיף זוג של מזהה וערך, ולחפש ערך של מזהה בסביבה. כאן נעשה זאת באופן קצת שונה, את שני הפעולות הראשונות נגדיר כווריאנטים של הטיפוס ואת הפעולה השלישית נגדיר כפונקציה.

(define-type VAL

[NumV Number]

[FunV Symbol FLANG ENV])

(define-type ENV

[EmptyEnv]

[Extend Symbol VAL ENV])

(: lookup : Symbol ENV -> VAL)

(define (lookup name env)

(cases env

[(EmptyEnv) (error 'lookup "free identifier ~s" name)]

[(Extend id val rest-env)

(if (eq? name id)

val

(lookup name rest-env))]))

### Evaluation

ראשית נצטרך לשנות את הפונקציה arith-op כך שתקבל ותחזיר VAL במקום FLANG. וכן תעבוד עם NumV במקום Num.

(: arith-op : (Number Number -> Number) VAL VAL -> VAL)

(define (arith-op op expr1 expr2)

(: NumV->number : VAL-> Number)

(define (NumV->number e)

(cases e

[(NumV n) n]

[else (error 'arith-op "expects a number, got: ~s" e)]))

(NumV (op (NumV->number expr1) (NumV->number expr2))))

השינויים המרכזיים בתוך הפונקציה eval הם:

* עבור ביטוי Num נחזיר ביטוי מסוג NumV המתאים לטיפוס VAL.
* עבור ביטוי מסוג Fun נחזיר Closure, כלומר ווריאנט של VAL המורכב מפונקציה וסביבה.
* עבור ביטוי מסוג Call קודם נוודא שאכן הביטוי המייצג את הפונקציה תקין, כלומר הוא מסוג FunV. במידה והפונקציה תקינה, **נוסיף לסביבה של הפונקציה** זוג נוסף של המזהה שהוגדר ב-FunV ואת הערך שנשלח כפרמטר ב-Call. שינוי קטן זה של הוספת המזהה לסביבה בעת הגדרת הפונקציה ולא לסביבה במצב נוכחי עושה את כל ההבדל בין מימוש לפי Dynamic scope ל-Static scope.

(: eval : FLANG ENV -> VAL)

(define (eval expr env)

(cases expr

[(Num n) (NumV n)]

[(Add l r) (arith-op + (eval l env) (eval r env))]

[(Sub l r) (arith-op - (eval l env) (eval r env))]

[(Mul l r) (arith-op \* (eval l env) (eval r env))]

[(Div l r) (arith-op / (eval l env) (eval r env))]

[(With name named-expr body)

(eval body (Extend name

(eval named-expr env)

env))]

[(Id name) (lookup name env)]

[(Fun name body) (FunV name body env)]

[(Call fun-exp arg-exp)

(let ([fval (eval fun-exp env)])

(cases fval

[(FunV name body f-env)

(eval body (Extend name

(eval arg-exp env)

f-env))]

[else (error 'eval "expects a function, got: ~s" fval)]))]))

### run

נשאר אותו הדבר עם ההתאמות לטיפוסים החדשים.

(: run : String -> Number)

(define (run code)

(let ([res (eval (parse code) (EmptyEnv))])

(cases res

[(NumV n) n]

[else (error 'run "evaluation returned a non-number: ~s" res)])))

## טיפוסים בשפה

הטיפוסים של שפה הם כל סוגי הערכים אותם היא יכולה להחזיר. ניתן לזהות טיפוסים אלו באמצעות כל הבנאים של הטיפוס נתונים ש-eval. מחזיר.

### Dynamic and Static Typing

אלו הן שיטות להגדרת מזהים בשפת תכנות. לפי Static typing לפני כל הגדרת מזהה, (משתנה, פרמטר או ערך חזרה מפונקציה), צריך להגדיר מהו טיפוס המשתנה. לעומת זאת ב-Dynamic typing אין צורך להגדיר מהו טיפוס המזהה אלא ניתן לתת לו ערך מיד עם הגדרתו. השיטות שנוקטות השפות שראינו בקורס זה הן:

* Racket - Dynamic typing.
* PL - Static typing.
* FLANG - Dynamic typing.

## סביבה גלובלית

בשפה הנוכחית שכתבנו, הפעולות המתמטיות (+, -, \*, /) אינן מוגדרות פונקציות אלו מילים שמורות. אם ננסה לקרוא לאופרטור כמו פונקציה, בדרך הבאה:

(run “(Call + 1 2)”)

נקבל שגיאת parser ש-Call צריך לקבל שני פרמטרים ולא 3. גם אם ננסה להריץ:

(run “(Call + 1)”)

נקבל שגיאה מ-lookup ש-"+" הוא free identifier, כלומר אין לו ערך שמור לו. נרצה להרחיב את השפה כך שאופרטורים ייחשבו כפונקציות. לשם כך נגדיר סביבה גלובלית שבה שמורים כל האופרטורים שלנו ואת הפונקציות אותן הן מייצגות. הסביבה ההתחלתית של התוכנית בפונקציה run לא תהיה סביבה ריקה emptyEnv אלא הסביבה הגלובלית.

כיוון שכל אופרטור מקבל שני פרמטרים, כדי לבצע עליהם פעולה חשבונית, ולא אחד, נצטרך לשנות את פונקציות ה-parse ו-eval כך שיאפשרו להגדיר ולקרוא פונקציות עם שני פרמטרים. נגדיר פונקציה שיוצרת סביבה גלובלית בדרך הבאה:

(: createGE : -> ENV)

(define (createGE)

(Extend '/

(FunV2 'x 'y (Div (Id 'x) (Id 'y)) (EmptyEnv))

(Extend '\*

(FunV2 'x 'y (Mul (Id 'x) (Id 'y)) (EmptyEnv))

(Extend '-

(FunV2 'x 'y (Sub (Id 'x) (Id 'y)) (EmptyEnv))

(Extend '+

(FunV2 'x 'y (Add (Id 'x) (Id 'y)) (EmptyEnv))

(EmptyEnv))))))

### parsing

נוסיף עוד שני בנאים לשפה, כך שיהיה אפשר להגדיר פונקציה Fun2 עם שני פרמטרים, וכן לקרוא לפונקציה כזו באמצעות Call2, המקבלת פונקציה ושני ערכים לפרמטרים. נוסיף שני בנאים אלו לפונקציה parse-sexpr. מכיוון שכל האופרטורים מטופלים בתור פונקציות אין צורך לעשות להם parsing באופן ייחודי, ולכן נמחק קטע קוד זה.

(define-type FLANG

[Num Number]

[Add FLANG FLANG]

[Sub FLANG FLANG]

[Mul FLANG FLANG]

[Div FLANG FLANG]

[With Symbol FLANG FLANG]

[Id Symbol]

[Fun Symbol FLANG]

[Fun2 Symbol Symbol FLANG]

[Call FLANG FLANG]

[Call2 FLANG FLANG FLANG])

(: parse-sexpr : Sexpr -> FLANG)

(define (parse-sexpr sxp)

(match sxp

[(number: n) (Num n)]

[(symbol: name) (Id name)]

[(cons 'with more)

(match sxp

[(list 'with (list (symbol: name) named-expr) body)

(With name (parse-sexpr named-expr) (parse-sexpr body))]

[else (error 'parse-sexpr "bad with syntax in ~s" sxp)])]

[(cons 'fun more)

(match sxp

[(list 'fun (list (symbol: name)) body)

(Fun name (parse-sexpr body))]

[(list 'fun (list (symbol: name1) (symbol: name2)) body)

(Fun2 name1 name2 (parse-sexpr body))]

[else (error 'parse-sexpr "bad fun syntax in ~s" sxp)])]

[(list 'call fun-exp arg-exp) (Call (parse-sexpr fun-exp) (parse-sexpr arg-exp))]

[(list 'call fun-exp arg1-exp arg2-exp)

(Call2 (parse-sexpr fun-exp)

(parse-sexpr arg1-exp)

(parse-sexpr arg2-exp))]

;; [(list '+ l r) (Add (parse-sexpr l) (parse-sexpr r))]

;; [(list '- l r) (Sub (parse-sexpr l) (parse-sexpr r))]

;; [(list '\* l r) (Mul (parse-sexpr l) (parse-sexpr r))]

;; [(list '/ l r) (Div (parse-sexpr l) (parse-sexpr r))]

[else (error 'parse-sexpr "bad syntax in ~s" sxp)]))

(: parse : String -> FLANG)

(define (parse code)

(parse-sexpr (string->sexpr code)))

### Evaluation

נוסיף ל-Val בנאי נוסף המייצג פונקציה עם שני פרמטרים. ב-eval נוסיף הערכה של הגדרה וקריאה של פונקציה עם שני פרמטרים. לסיום, ב-run נאתחל עם הסביבה הגלובלית ולא סביבה ריקה.

(define-type VAL

[NumV Number]

[FunV Symbol FLANG ENV]

[FunV2 Symbol Symbol FLANG ENV])

(: eval : FLANG ENV -> VAL)

(define (eval expr env)

(cases expr

[(Num n) (NumV n)]

[(Add l r) (arith-op + (eval l env) (eval r env))]

[(Sub l r) (arith-op - (eval l env) (eval r env))]

[(Mul l r) (arith-op \* (eval l env) (eval r env))]

[(Div l r) (arith-op / (eval l env) (eval r env))]

[(With name named-expr body)

(eval body (Extend name

(eval named-expr env)

env))]

[(Id name) (lookup name env)]

[(Fun name body) (FunV name body env)] ;- closure - a function is now a closed object

[(Fun2 name1 name2 body) (FunV2 name1 name2 body env)]

[(Call fun-exp arg-exp)

(let ([fval (eval fun-exp env)])

(cases fval

[(FunV name body f-env) (eval body (Extend name

(eval arg-exp env)

f-env))]

[else (error 'eval "expects a single parameter function, got: ~s" fval)]))]

[(Call2 fun-exp arg1-exp arg2-exp)

(let ([fval (eval fun-exp env)])

(cases fval

[(FunV2 name1 name2 body f-env)

(eval body (Extend name2

(eval arg2-exp env)

(Extend name1

(eval arg1-exp env)

f-env)))]

[else (error 'eval "expects a two-parameter function, got: ~s" fval)]))]))

(: run : String -> Number)

(define (run code)

(let ([res (eval (parse code) (createGE))])

(cases res

[(NumV n) n]

[else (error 'run "evaluation returned a non-number: ~s" res)])))

## אובייקט Pair

נרצה להגדיר אובייקט המייצג זוג של ערכים ב-Racket. הפונקציות השימושיות לאובייקט זה הן:

* mycons - יוצרת זוג.
* myfirst - מקבלת זוג ומחזירה את האיבר הראשון.
* mysecond - מקבלת זוג ומחזירה את האיבר השני.

לשם כך נגדיר פונקציה mycons שמקבלת שני פרמטרים ומחזירה פונקציה pair שמפעילה פונקציה selector על שני איברים אלו. הפונקציה myfirst מקבלת את הפונקציה pair ומכניסה לה בתור פרמטר פונקציה שמחזירה את האיבר הראשון. mysecond עושה אותו הדבר רק עם פונקציה שמחזירה את האיבר השני.

(define (mycons x y)

(define (pair selector)

(selector x y))

pair)

(define (myfirst p)

(p (lambda (a b) a)))

(define (mysecond p)

(p (lambda (a b) b)))

; example

(define p1 (mycons 4 5))

(myfirst p1) ; => 4

(mysecond p1) ; => 5