פקולטה: מדעי הטבע

מחלקה: מדעי המחשב ומתמטיקה

שם הקורס: שפות תכנות

קוד הקורס: 2-7036010

תאריך בחינה: \_24/12/2015 סמ' \_ב'\_\_\_מועד \_\_מיוחד 2\_\_

משך הבחינה: 3 שעות

שם המרצה: ערן עמרי

חומר עזר: אסור

שימוש במחשבון: לא

הוראות כלליות:

* כתבו את תשובותיכם בכתב קריא ומרווח (**במחברת התשובות בלבד** ולא על גבי טופס המבחן עצמו.
* בכל שאלה או סעיף **(שבהם נדרשת כתיבה, מעבר לסימון תשובות נכונות)**, ניתן לכתוב – **לא יודע/ת** (מבלי להוסיף דבר מעבר לכך) – ולקבל 20% מהניקוד על השאלה או הסעיף (לא חל על חלקי סעיפים).
* אפשר להסתמך על סעיפים קודמים גם אם לא עניתם עליהם.
* יש לענות על כל השאלות.
* ניתן להשיג עד 109 נקודות במבחן.
* לנוחיותכם מצורפים שלושה קטעי קוד עבור ה- interpreter של FLANG בסוף טופס המבחן. הראשון – במודל ה-substitution, השני במודל הסביבות והשלישי במודל ה- Substitution-cache.

שאלה 1 — BNF — (20 נקודות):

נתון הדקדוק (BNF) הבא:

**<TREE> ::= <ATOM>**

**| [ <TREE> ] <OP> { <TREE> }**

**<ATOM> ::= <num>**

**<OP> ::= + | - | / | \***

כאשר **<num>** מתאר ערך מספרי כלשהו על-פי הגדרת RACKET.

סעיף א' (5 נקודות):

מהי השפה שמגדיר הדקדוק? כתבו תיאור קצר (עד ארבע שורות), אך ברור (הקפידו להשתמש במונחים נכונים). ציינו האם ביטויים בשפה הם prefix, infix, או postfix.

סעיף ב' (5 נקודות):

הראו גזירה עבור מילה תוך שימוש בלפחות 7 כללי גזירה (ציינו בצורה מפורשת מהי המילה אותה גזרתם). ציינו את הכלל שבו השתמשתם בכל מעבר.

סעיף ג' (5 נקודות):

האם הדקדוק הנתון הינו רב-משמעי (כלומר, האם הוא סובל מ-ambiguity)? הסבירו את תשובתכם (בפרט, אם כן – הסבירו כיצד ניתן לתקן זאת מבלי לשנות את השפה).

סעיף ד' (5 נקודות):

השתמשו בדקדוק הנ"ל ובסמנטיקה של שפת PL בכדי להסביר את מושג ה-compositionality. בפרט, ציינו האם תכונה זו מתקיימת. אם לדעתכם היא אינה מתקיימת הסבירו כיצד ניתן לתקן זאת. אם לדעתכם היא מתקיימת תנו דוגמה לגבי המילה שבחרתם בסעיף ב'.

שאלה 2 — שאלות כלליות — (14 נקודות):

סעיף א' (5 נקודות): (סמנו את כל התשובות הנכונות)

אילו מהמשפטים הבאים נכונים?

1. חיפוש שמות מזהים חופשיים דורש הערכת הקוד (evaluation) ולכן, רק בשפות פונקציונאליות מותר לבצעו תוך כדי parse.
2. הפקודה x=x+1; בשפת C ובשפת Java היא side-effect שכן היא משנה משהו בזיכרון מבלי שניתן לראות את השינוי בקוד עצמו.
3. בתכנית הכתובה בשפה פונקציונאלית צפוי שנראה פחות side-effects מאשר בשפה בתכנית אימפרטיבית.
4. השפה FLANG שכתבנו בקורס היא שפה פונקציונאלית המתייחסת לפונקציות כ-first class.

סעיף ב' (4 נקודות): (סמנו את כל התשובות הנכונות)

נתון הקוד הבא ב-Racket?

**(: foo : Natural (Number -> Number) (Listof Number)**

**(Listof Number) -> (Listof Any))**

**(define (foo n f l1 l2)**

**(if (null? l1)**

**(cons (list n) l2)**

**(let ([q (f (first l1))])**

**(if (> n 0)**

**(foo (- n 1) f (rest l1) (cons (first l1) l2))**

**(foo n f '() l2)))))**

**(: Ide : Number -> Number)**

**(define (Ide c) c)**

1. הקריאות הרקורסיביות ל-foo אינן ברקורסיית זנב. בפרט, בחישוב יש פתיחת סביבה לוקאלית של let השקולה להפעלה מקומית של פונקציה אנונימית.
2. ניתן להכניס את הפונקציה foo ללולאה אינסופית (בעזרת בחירת פרמטרים נכונים – אך לא כאלה שבעצמם נכנסים ללולאה אינסופית).
3. ההתניה על **(> n 0)**היא הכרחית כדי שה – type-checker של Racket ידע שהשורה הבאה אחריה הינה חוקית.
4. הערך המוחזר של הפונקציה אינו יכול להיות (Listof Number).

סעיף ג' (5 נקודות):

הריצו את שורת הקוד הבאה, על פי הקוד מסעיף ג'.

**(foo 10 Ide '(1 2 3 4) '())**

שאלה 3 — (34 נקודות):

נתון הקוד הבא:

**(run "{with {w {fun {y} {call q y}}}**

**{with {q {fun {d} {+ d d}}}**

**{call w 2}}}")**

סעיף א' (12 נקודות):

תארו את הפעלות הפונקציה eval בתהליך ההערכה (יש 13 הפעלות) של הקוד מעלה במודל ההחלפות (על-פי ה-interpreter העליון מבין השלושה המצורפים מטה) - באופן הבא – לכל הפעלה מספר *i* תארו את הפרמטר האקטואלי ה - *i* () וכן את הערך המוחזר מהפעלה זו ().

**הסבירו בקצרה כל מעבר**. ציינו מהי התוצאה הסופית.

**דוגמת הרצה:** עבור הקוד

**(run "{with {x 1} {+ x 2}}")**

היה עליכם לענות (בתוספת הסברים)

**= (With x (Num 1) (Add (Id x) (Num 2)))**

**= (Num 3)**

**= (Num 1)**

**= (Num 1)**

**= (Add (Num 1) (Num 2))**

**= (Num 3)**

**= (Num 1)**

**= (Num 1)**

**= (Num 2)**

**= (Num 2)**

**Final result: 3**

סעיף ב' (14 נקודות):

תארו את הפעלות הפונקציה eval בתהליך ההערכה של הקוד מעלה במודל ה-substitution-cache (על-פי ה-interpreter התחתון מבין השלושה המצורפים מטה) - באופן הבא – לכל הפעלה מספר *i* תארו את – הפרמטר האקטואלי הראשון בהפעלה מספר *i (עץ התחביר האבסטרקטי), את*  – הפרמטר האקטואלי השני בהפעלה מספר *i (רשימת ההחלפות) ואת*  – הערך המוחזר מהפעלה מספר *i.*

**הסבירו בקצרה כל מעבר**. ציינו מהי התוצאה הסופית.

**דוגמת הרצה:** עבור הקוד

**(run "{with {x 1} {+ x 2}}")**

היה עליכם לענות (בתוספת הסברים)

**= (With x (Num 1) (Add (Id x) (Num 2)))**

**= '()**

**= (Num 3)**

**= (Num 1)**

**= '()**

**= (Num 1)**

**= (Add (Id x) (Num 2))**

**= '((x (Num 1))**

**= (Num 3)**

**= (Id x)**

**= '(x (NumV 1))**

**= (Num 1)**

**= (Num 2)**

**= '((x (Num 1))**

**= (Num 2)**

**Final result: 3**

סעיף ג' (8 נקודות):

מה היה קורה לו היינו מבצעים את ההערכה במודל הסביבות? מהי התשובה הרצויה? מדוע? (אין צורך לבצע הערכה)

הסבירו בקצרה מדוע נתקבלו התוצאות כפי שנתקבלו בסעיפים א' ו-ב'.

תשובה מלאה לסעיף זה לא תהיה ארוכה מחמש שורות.

שאלה 4 — הרחבת השפה **FLANG** במודל הסביבות — (41 נקודות):

נרצה להרחיב את השפה **FLANG** ולאפשר מספר משתנה של ארגומנטים לאופרטורים האריתמטיים ולפונקציות המוגדרות בקוד. להלן דוגמאות לטסטים שאמורים לעבוד:

**(test (run "{with {x 5} {\* x x x}}") => 125)**

**(test (run "{+ {- 10 2 3 4} {/ 3 3 1} {\*}}") => 3)**

**(test (run "{+ 1 2 3 4}") => 10)**

**(test (run "{+}") => 0)**

**(test (run "{\*}") => 1)**

**(test (run "{/ 4}") => 4)**

**(test (run "{- 4}") => 4)**

**(test (run "{/ 16 2 4}") => 2)**

**(test (run "{/}") =error> "bad syntax")**

**(test (run "{call {fun {x} {+ x x x 1}} 4}")**

**=> 13)**

**(test (run "{with {identity {fun {x} x}}**

**{with {foo {fun {} {- 2 3 3}}}**

**{call {call identity foo}}}}")**

**=> -4)**

**(test (run "{with {f {fun {y z x} {+ z y}}}**

**{call f 4 1 3}}")**

**=> 5)**

**(test (run "{with {f {fun {y z y} {+ z y}}}**

**{call f 4 1 3}}")**

**=error> "parse-sexpr: parameters should not repeat in `fun' syntax")**

**(test (run "{with {f {fun {y z w} {+ x y}}}**

**{with {x 5}**

**{call f 4 1}}}")**

**=error> "eval: wrong number of arguments to function")**

**(test (run "{call {with {x 3}**

**{fun {x y z} {- x y z}}}**

**4 2 3}")**

**=> -1)**

הערה: השתמשו בדוגמאות אלו בכדי להבין את דרישות התחביר, את אופן הערכת הקוד וכן את הודעות השגיאה שיש להדפיס במקרים המתאימים.

סעיף א' (הרחבת הדקדוק) (5 נקודות):

הוסיפו את הקוד הנדרש (בתוך הסוגריים המרובעים – 4 השלמות סה"כ לסעיף זה) ל –

**#|**

**The grammar:**

**<FLANG> ::= <num>**

**| { + <FLANG> ... }**

**| { - <FLANG> <FLANG> ... }**

**| { —«fill-in 01»—}**

**| { —«fill-in 02»— }**

**| { with { <id> <FLANG> } <FLANG> }**

**| <id>**

**| { fun { —«fill-in 03»— } <FLANG> }**

**| { call —«fill-in 04»— }**

**|#**

סעיף ב' (הרחבת הטיפוס FLANG) (6 נקודות):

הוסיפו את הקוד הנדרש (בתוך הסוגריים המרובעים – 6 השלמות סה"כ לסעיף זה) ל –

**(define-type FLANG**

**[Num Number]**

**[Add —«fill-in 05»— ]**

**[Sub —«fill-in 06»— ]**

**[Mul —«fill-in 07»—]**

**[Div —«fill-in 08»—]**

**[Id Symbol]**

**[With Symbol FLANG FLANG]**

**[Fun —«fill-in 09»—]**

**[Call —«fill-in 10»— (Listof FLANG)])**

סעיף ג' (פונקציות עזר לניתוח תחבירי) (8 נקודות):

לצורך תהליך הניתוח התחבירי (parsing), נשתמש בפונקציות העזר שתבנו כעת.

ראשית, כתבו פונקציה:

**(: not-member : Symbol (Listof Symbol) -> Boolean)**

המקבלת סימבול ורשימה של סימבולים ומחזירה true אם הסימבול אינו מופיע ברשימה ו-false אחרת.

שנית, כתבו פונקציה:

**(: all-unique : (Listof Symbol) -> Boolean)**

המקבלת רשימה של סימבולים ומחזירה true אם כל הסימבולים ברשימה שונים זה מזה ו-false אחרת.

סעיף ד' (parsing) (8 נקודות):

השתמשו בקוד הבא (ובפונקציות העזר שכתבתם) –

**(: parse-sexpr\* : (Listof Sexpr) -> (Listof FLANG))**

**;; to convert a list of s-expressions into a list of FLANGSs**

**(define (parse-sexpr\* sexprs)**

**(map parse-sexpr sexprs))**

והוסיפו את הקוד הנדרש (בתוך הסוגריים המרובעים – 7 השלמות סה"כ לסעיף זה) ל –

**(: parse-sexpr : Sexpr -> FLANG)**

**;; to convert s-expressions into FLANGs**

**(define (parse-sexpr sexpr)**

**(match sexpr**

**[(number: n) (Num n)]**

**[(symbol: name) (Id name)]**

**[(cons 'with more)**

**(match sexpr**

**[(list 'with (list (symbol: name) named) body)**

**(With name (parse-sexpr named)**

**(parse-sexpr body))]**

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

**[(cons 'fun more)**

**(match sexpr**

**[(list 'fun (list (symbol: names) ... ) body)**

**(if —«fill-in 11»— ;; verify all are different**

**(Fun —«fill-in 12»— )**

**(—«fill-in 13»—)];; If found repetition**

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

**[(list '+ args ...) (Add (parse-sexpr\* args))]**

**[(list '- fst args ...) —«fill-in 14»—]**

**[—«fill-in 15»—]**

**[—«fill-in 16»—]**

**[(list 'call fun args ...) —«fill-in 17»—]**

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

סעיף ה' (14 נקודות):

בסעיף זה נכתוב את חלק הקוד המבצע הערכה (evaluation).

שימו לב – ההגדרות הבאות נשארות בדיוק כמו בקוד האינטרפרטר המצורף מטה.

**;; Types for environments, values, and a lookup function**

**(define-type ENV**

**[EmptyEnv]**

**[Extend Symbol VAL ENV])**

**(define-type VAL**

**[NumV Number]**

**[FunV (Listof Symbol) FLANG ENV])**

**(: lookup : Symbol ENV -> VAL)**

**(define (lookup name env)**

**(cases env**

**[(EmptyEnv) (error 'lookup "no binding for ~s" name)]**

**[(Extend id val rest-env)**

**(if (eq? id name) val (lookup name rest-env))]))**

השתמשו בהגדרות הפורמליות הבאות –

**#| Formal specs for `eval':**

**eval(N,env) = N**

**eval({+ E ...}, env) = eval(E,env) + ...**

**eval({- E1 E ...}, env) = eval(E1,env) - (eval(E,env) + ...)**

**eval({\* E ...}, env) = eval(E, env) \* ...**

**eval({/ E1 E ...}, env)=eval(E1, env) / (eval(E, env) \* ...)**

**eval(x,env) = lookup(x,env)**

**eval({with {x E1} E2},env) =**

**eval(E2,extend(x,eval(E1,env),env))**

**eval({fun {x ... } E},env) = <{fun {x ... } E}, env>**

**eval({call E1 E ... },env1)**

**= eval(Ef,extend(x,eval(E2,env1),env2))**

**if eval(E1,env1) = <{fun {x} Ef}, env2>**

**and number of parameters is as implied by f**

**= error! otherwise |#**

והוסיפו את הקוד הנדרש (בתוך הסוגריים המרובעים – 10 השלמות סה"כ לסעיף זה) לפונקציות הבאות:

פונקציית עזר – ראשית נכתוב פונקציית עזר אשר תדע להוסיף רשימת זוגות לסביבה קיימת על-ידי הפעלה רקורסיבית של הבנאי המתאים.

**(: extend\* : (Listof Symbol) (Listof VAL) ENV -> ENV)**

**;; extends a given environment according to the**

**;; lists of names and values, respectively.**

**;; Assumes that the two lists are of the same length.**

**(define (extend\* names vals env)**

**(if (and (null? names)(null? vals))**

**—«fill-in 18»—**

**—«fill-in 19»—)**

השתמשו בפונקציה שכתבתם ובשתי הפונקציות הבאות בכדי להשלים את הקוד החסר ל-eval.

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

**(define (NumV->number v)**

**(cases v**

**[(NumV n) n]**

**[else (error 'NumV->number "expects a number, got: ~s" v)]))**

**(: eval\* : (Listof FLANG) ENV -> (Listof VAL))**

**;; evaluates a list of FLANG expressions by**

**;; reducing it to a list of values**

**(define (eval\* l env)**

**(: eval-straight : FLANG -> VAL)**

**(define (eval-straight exp)**

**(eval exp env))**

**(map eval-straight l))**

**(: eval : FLANG ENV -> VAL)**

**;; evaluates FLANG expressions by reducing them to values**

**(define (eval expr env)**

**(cases expr**

**[(Num n) (NumV n)]**

**[(Add args) (NumV —«fill-in 20»—)]**

**[(Sub fst args) —«fill-in 21»—)]**

**[—«fill-in 22»— ]**

**[—«fill-in 23»— ]**

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

**(eval bound-body**

**(Extend bound-id (eval named-expr env) env))]**

**[(Id name) —«fill-in 24»— ]**

**[(Fun bound-ids bound-body) —«fill-in 25»— ]**

**[(Call fun-expr arg-exprs)**

**(let ([fval (eval fun-expr env)])**

**(cases fval**

**[(FunV bound-ids bound-body f-env)**

**(if —«fill-in 26»—**

**—«fill-in 27»—**

**(error 'eval "wrong number of arguments to function ~s" fval))]**

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

**fval)]))]))**

הדרכה: השתמשו בפונקציות [map](http://docs.racket-lang.org/reference/pairs.html?q=map&q=exp&q=foldl#%28def._%28%28lib._racket%2Fprivate%2Fmap..rkt%29._map%29%29) ו-[foldl](http://docs.racket-lang.org/reference/pairs.html?q=map&q=exp&q=foldl#%28def._%28%28lib._racket%2Fprivate%2Flist..rkt%29._foldl%29%29)  של RACKET המתוארות מטה.

אין צורך לטפל בחלוקה באפס.

תוספות:

הפונקציה map:

קלט: פרוצדורה proc ורשימה lst

פלט: רשימה שמכילה אותו מספר איברים כמו ב- lst – שנוצרה ע"י הפעלת הפרוצדורה proc על כל אחד מאיברי הרשימה lst. (ההסבר הבא הוא כללי יותר – כי למעשה הפונקציה map יכולה לטפל במספר רשימות – לצורך השאלה הנתונה לא תזדקקו לשימוש כזה)

|  |
| --- |
| ([map](http://docs.racket-lang.org/reference/pairs.html?q=map&q=exp&q=foldl" \l "%28def._%28%28lib._racket%2Fprivate%2Fmap..rkt%29._map%29%29) proc lst ...+) → [list?](http://docs.racket-lang.org/reference/pairs.html?q=map&q=exp&q=foldl#%28def._%28%28quote._%7E23%7E25kernel%29._list%7E3f%29%29) |
| proc : [procedure?](http://docs.racket-lang.org/reference/procedures.html?q=map&q=exp&q=foldl#%28def._%28%28quote._%7E23%7E25kernel%29._procedure%7E3f%29%29) |
| lst : [list?](http://docs.racket-lang.org/reference/pairs.html?q=map&q=exp&q=foldl#%28def._%28%28quote._%7E23%7E25kernel%29._list%7E3f%29%29) |

Applies proc to the elements of the lsts from the first elements to the last. The proc argument must accept the same number of arguments as the number of supplied lsts, and all lsts must have the same number of elements. The result is a list containing each result of proc in order.

דוגמאות:

> (map add1 (list 1 2 3 4))

'(2 3 4 5)

> (map (lambda (x) (list x))

'(sym1 sym2 33))

'((sym1) (sym2) (33))

הפונקציה foldl:

קלט: פרוצדורה proc, ערך התחלתי init ורשימה lst

פלט: ערך סופי (מאותו טיפוס שמחזירה הפרוצדורה proc) שנוצר ע"י הפעלת הפרוצדורה proc על כל אחד מאיברי הרשימה lst תוך שימוש במשתנה ששומר את הערך שחושב עד כה – משתנה זה מקבל כערך התחלתי את הערך של init. (ההסבר הבא הוא כללי יותר – כי למעשה הפונקציה foldl יכולה לטפל במספר רשימות – לצורך השאלה הנתונה לא תזדקקו לשימוש כזה)

|  |
| --- |
| ([foldl](http://docs.racket-lang.org/reference/pairs.html?q=map&q=exp&q=foldl" \l "%28def._%28%28lib._racket%2Fprivate%2Flist..rkt%29._foldl%29%29) proc init lst ...+) → [any/c](http://docs.racket-lang.org/reference/data-structure-contracts.html?q=map&q=exp&q=foldl#%28def._%28%28lib._racket%2Fcontract%2Fprivate%2Fmisc..rkt%29._any%2Fc%29%29) |
| proc : [procedure?](http://docs.racket-lang.org/reference/procedures.html?q=map&q=exp&q=foldl#%28def._%28%28quote._%7E23%7E25kernel%29._procedure%7E3f%29%29) |
| init : [any/c](http://docs.racket-lang.org/reference/data-structure-contracts.html?q=map&q=exp&q=foldl#%28def._%28%28lib._racket%2Fcontract%2Fprivate%2Fmisc..rkt%29._any%2Fc%29%29) |
| lst : [list?](http://docs.racket-lang.org/reference/pairs.html?q=map&q=exp&q=foldl#%28def._%28%28quote._%7E23%7E25kernel%29._list%7E3f%29%29) |

Like [map](http://docs.racket-lang.org/reference/pairs.html?q=map&q=exp&q=foldl#%28def._%28%28lib._racket%2Fprivate%2Fmap..rkt%29._map%29%29), [foldl](http://docs.racket-lang.org/reference/pairs.html?q=map&q=exp&q=foldl#%28def._%28%28lib._racket%2Fprivate%2Flist..rkt%29._foldl%29%29) applies a procedure to the elements of one or more lists. Whereas [map](http://docs.racket-lang.org/reference/pairs.html?q=map&q=exp&q=foldl#%28def._%28%28lib._racket%2Fprivate%2Fmap..rkt%29._map%29%29) combines the return values into a list, [foldl](http://docs.racket-lang.org/reference/pairs.html?q=map&q=exp&q=foldl#%28def._%28%28lib._racket%2Fprivate%2Flist..rkt%29._foldl%29%29) combines the return values in an arbitrary way that is determined by proc.

דוגמאות:

> (foldl + 0 '(1 2 3 4))

10

> (foldl cons '() '(1 2 3 4))

'(4 3 2 1)

**---<<<FLANG>>>-------------------------------------------------------**

**;; The Flang interpreter (substitution model)**

**#lang pl**

**#|**

**The grammar:**

**<FLANG> ::= <num>**

**| { + <FLANG> <FLANG> }**

**| { - <FLANG> <FLANG> }**

**| { \* <FLANG> <FLANG> }**

**| { / <FLANG> <FLANG> }**

**| { with { <id> <FLANG> } <FLANG> }**

**| <id>**

**| { fun { <id> } <FLANG> }**

**| { call <FLANG> <FLANG> }**

**Evaluation rules:**

**subst:**

**N[v/x] = N**

**{+ E1 E2}[v/x] = {+ E1[v/x] E2[v/x]}**

**{- E1 E2}[v/x] = {- E1[v/x] E2[v/x]}**

**{\* E1 E2}[v/x] = {\* E1[v/x] E2[v/x]}**

**{/ E1 E2}[v/x] = {/ E1[v/x] E2[v/x]}**

**y[v/x] = y**

**x[v/x] = v**

**{with {y E1} E2}[v/x] = {with {y E1[v/x]} E2[v/x]} ; if y =/= x**

**{with {x E1} E2}[v/x] = {with {x E1[v/x]} E2}**

**{call E1 E2}[v/x] = {call E1[v/x] E2[v/x]}**

**{fun {y} E}[v/x] = {fun {y} E[v/x]} ; if y =/= x**

**{fun {x} E}[v/x] = {fun {x} E}**

**eval:**

**eval(N) = N**

**eval({+ E1 E2}) = eval(E1) + eval(E2) \ if both E1 and E2**

**eval({- E1 E2}) = eval(E1) - eval(E2) \ evaluate to numbers**

**eval({\* E1 E2}) = eval(E1) \* eval(E2) / otherwise error!**

**eval({/ E1 E2}) = eval(E1) / eval(E2) /**

**eval(id) = error!**

**eval({with {x E1} E2}) = eval(E2[eval(E1)/x])**

**eval(FUN) = FUN ; assuming FUN is a function expression**

**eval({call E1 E2}) = eval(Ef[eval(E2)/x]) if eval(E1)={fun {x} Ef}**

**= error! otherwise**

**|#**

**(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])**

**(: parse-sexpr : Sexpr -> FLANG)**

**;; to convert s-expressions into FLANGs**

**(define (parse-sexpr sexpr)**

**(match sexpr**

**[(number: n) (Num n)]**

**[(symbol: name) (Id name)]**

**[(cons 'with more)**

**(match sexpr**

**[(list 'with (list (symbol: name) named) body)**

**(With name (parse-sexpr named) (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 '+ lhs rhs) (Add (parse-sexpr lhs) (parse-sexpr rhs))]**

**[(list '- lhs rhs) (Sub (parse-sexpr lhs) (parse-sexpr rhs))]**

**[(list '\* lhs rhs) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]**

**[(list '/ lhs rhs) (Div (parse-sexpr lhs) (parse-sexpr rhs))]**

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

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

**(: parse : String -> FLANG)**

**;; parses a string containing a FLANG expression to a FLANG AST**

**(define (parse str)**

**(parse-sexpr (string->sexpr str)))**

**(: subst : FLANG Symbol FLANG -> FLANG)**

**;; substitutes the second argument with the third argument in the**

**;; first argument, as per the rules of substitution; the resulting**

**;; expression contains no free instances of the second argument**

**(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**

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

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

**;; gets a Racket numeric binary operator, and uses it within a FLANG**

**;; `Num' wrapper**

**(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))))**

**(: eval : FLANG -> FLANG)**

**;; evaluates FLANG expressions by reducing them to \*expressions\***

**(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)))]**

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

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

**[(Call fun-expr arg-expr)**

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

**(cases fval**

**[(Fun bound-id bound-body)**

**(eval (subst bound-body**

**bound-id**

**(eval arg-expr)))]**

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

**fval)]))]))**

**(: run : String -> Number)**

**;; evaluate a FLANG program contained in a string**

**(define (run str)**

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

**(cases result**

**[(Num n) n]**

**[else (error 'run**

**"evaluation returned a non-number: ~s" result)])))**

**;; tests**

**(test (run "{call {fun {x} {+ x 1}} 4}")**

**=> 5)**

**(test (run "{with {add3 {fun {x} {+ x 3}}}**

**{call add3 1}}")**

**=> 4)**

**--<<<FLANG-ENV>>>----------------------------------------------------**

**;; The Flang interpreter, using environments**

**#lang pl**

**#|**

**The grammar:**

**<FLANG> ::= <num>**

**| { + <FLANG> <FLANG> }**

**| { - <FLANG> <FLANG> }**

**| { \* <FLANG> <FLANG> }**

**| { / <FLANG> <FLANG> }**

**| { with { <id> <FLANG> } <FLANG> }**

**| <id>**

**| { fun { <id> } <FLANG> }**

**| { call <FLANG> <FLANG> }**

**Evaluation rules:**

**eval(N,env) = N**

**eval({+ E1 E2},env) = eval(E1,env) + eval(E2,env)**

**eval({- E1 E2},env) = eval(E1,env) - eval(E2,env)**

**eval({\* E1 E2},env) = eval(E1,env) \* eval(E2,env)**

**eval({/ E1 E2},env) = eval(E1,env) / eval(E2,env)**

**eval(x,env) = lookup(x,env)**

**eval({with {x E1} E2},env) = eval(E2,extend(x,eval(E1,env),env))**

**eval({fun {x} E},env) = <{fun {x} E}, env>**

**eval({call E1 E2},env1)**

**= eval(Ef,extend(x,eval(E2,env1),env2))**

**if eval(E1,env1) = <{fun {x} Ef}, env2>**

**= error! otherwise**

**|#**

**(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])**

**(: parse-sexpr : Sexpr -> FLANG)**

**;; to convert s-expressions into FLANGs**

**(define (parse-sexpr sexpr)**

**(match sexpr**

**[(number: n) (Num n)]**

**[(symbol: name) (Id name)]**

**[(cons 'with more)**

**(match sexpr**

**[(list 'with (list (symbol: name) named) body)**

**(With name (parse-sexpr named) (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 '+ lhs rhs) (Add (parse-sexpr lhs) (parse-sexpr rhs))]**

**[(list '- lhs rhs) (Sub (parse-sexpr lhs) (parse-sexpr rhs))]**

**[(list '\* lhs rhs) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]**

**[(list '/ lhs rhs) (Div (parse-sexpr lhs) (parse-sexpr rhs))]**

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

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

**(: parse : String -> FLANG)**

**;; parses a string containing a FLANG expression to a FLANG AST**

**(define (parse str)**

**(parse-sexpr (string->sexpr str)))**

**;; Types for environments, values, and a lookup function**

**(define-type ENV**

**[EmptyEnv]**

**[Extend Symbol VAL ENV])**

**(define-type VAL**

**[NumV Number]**

**[FunV Symbol FLANG ENV])**

**(: lookup : Symbol ENV -> VAL)**

**(define (lookup name env)**

**(cases env**

**[(EmptyEnv) (error 'lookup "no binding for ~s" name)]**

**[(Extend id val rest-env)**

**(if (eq? id name) val (lookup name rest-env))]))**

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

**;; gets a Racket numeric binary operator, and uses it within a NumV**

**;; wrapper**

**(define (arith-op op val1 val2)**

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

**(define (NumV->number v)**

**(cases v**

**[(NumV n) n]**

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

**(NumV (op (NumV->number val1) (NumV->number val2))))**

**(: eval : FLANG ENV -> VAL)**

**;; evaluates FLANG expressions by reducing them to values**

**(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 bound-id named-expr bound-body)**

**(eval bound-body**

**(Extend bound-id (eval named-expr env) env))]**

**[(Id name) (lookup name env)]**

**[(Fun bound-id bound-body)**

**(FunV bound-id bound-body env)]**

**[(Call fun-expr arg-expr)**

**(let ([fval (eval fun-expr env)])**

**(cases fval**

**[(FunV bound-id bound-body f-env)**

**(eval bound-body**

**(Extend bound-id (eval arg-expr env) f-env))]**

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

**fval)]))]))**

**(: run : String -> Number)**

**;; evaluate a FLANG program contained in a string**

**(define (run str)**

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

**(cases result**

**[(NumV n) n]**

**[else (error 'run**

**"evaluation returned a non-number: ~s" result)])))**

**--<<<FLANG-Substitution-cache>>>----------------------------------------------------**

**;; The Flang interpreter, using Substitution-cache**

**#lang pl**

**(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])**

**(: parse-sexpr : Sexpr -> FLANG)**

**;; to convert s-expressions into FLANGs**

**(define (parse-sexpr sexpr)**

**(match sexpr**

**[(number: n) (Num n)]**

**[(symbol: name) (Id name)]**

**[(cons 'with more)**

**(match sexpr**

**[(list 'with (list (symbol: name) named) body)**

**(With name (parse-sexpr named) (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 '+ lhs rhs) (Add (parse-sexpr lhs) (parse-sexpr rhs))]**

**[(list '- lhs rhs) (Sub (parse-sexpr lhs) (parse-sexpr rhs))]**

**[(list '\* lhs rhs) (Mul (parse-sexpr lhs) (parse-sexpr rhs))]**

**[(list '/ lhs rhs) (Div (parse-sexpr lhs) (parse-sexpr rhs))]**

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

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

**(: parse : String -> FLANG)**

**;; parses a string containing a FLANG expression to a FLANG AST**

**(define (parse str)**

**(parse-sexpr (string->sexpr str)))**

**;; a type for substitution caches:**

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

**(: empty-subst : SubstCache)**

**(define empty-subst null)**

**(: extend : Symbol FLANG SubstCache -> SubstCache)**

**(define (extend name val sc)**

**(cons (list name val) 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))))**

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

**;; gets a Racket numeric binary operator, and uses it within a FLANG**

**;; `Num' wrapper**

**(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))))**

**(: eval : FLANG SubstCache -> FLANG)**

**;; evaluates FLANG expressions by reducing them to expressions**

**(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 : String -> Number)**

**;; evaluate a FLANG program contained in a string**

**(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)])))**