פקולטה: מדעי הטבע

מחלקה: מדעי המחשב ומתמטיקה

שם הקורס: שפות תכנות

קוד הקורס: 2-7036010

תאריך מטלה: _05/07/2020_ סמ' _ב'___ מועד __א'__

משך הזמן המוקצה למטלה: 6:30—21:00

שם המרצה: ערן עמרי

חומר עזר: מותר להשתמש בכל חומר שנלמד בקורס, באתר של DrRacket Documentation. אין לחפש פתרונות כתובים בשום מקור ואין להתייעץ עם חברים.

הוראות כלליות:

- **קראו היטב את הוראות המטלה.**
- **המטלה לעבודה ביחידים בלבד.**
- **הגישו קובץ rkt. יחיד בלבד!!!**
- בכל מקום הדורש תכנות, מותר להגדיר פרוצדורות עזר (אלא אם נאמר אחרת במפורש).
- ניתן להשיג עד 110 נקודות במטלה. השתמשו בלינק הבא לאינטרפרטר החסר הנתון.
- בתחתית מסמך זה יש הפניה לפונקציות שימושיות בשפה ראקט (pl).
- עשו מאמץ להגיש קובץ שרץ ללא תקלות. מטלה שאינה רצה תבדק, אך תהיה הפחתת ניקוד אוטומטית על מצב זה.
- אם לא פתרתם את כל המטלה, אין זה אומר שלא תקבלו ציון עובר. עשו כמיטב יכולתכם.
- זכרו כי כל העתקה, זיוף או חריגה מכללי האתיקה מקרינים עליכם כבני אדם, בעוד הצלחה או כשלון בקורס אינם מגדירים אתכם. זכרו כי הבחירה היא שלכם, אך העתקה עושה עוול לכל מי שבוחר להשאר הגון. **השארו הגונים והוגנים.**

מבוא:

במטלה זאת נבנה שפה לטיפול בקבוצות, נקרא לה SOL. אנחנו נחשוב על קבוצה כסדרת מספרים ממויינת ללא חזרות. אנחנו נאפשר בשפה שלנו שלוש פעולות פרימיטיביות: 1. איחוד, 2. חיתוך, 3. הכפלה בסקלר. בנוסף, בדומה לשפה FLANG נאפשר שמות מזהים (ביטויי with), הגדרת פונקציות (ביטויי fun – אולם, כאן נטפל בפונקציות שתמיד מקבלות שני פרמטרים).

סוגי קריאה לפונקציה – אנחנו נאפשר בשפה שלנו שני סוגי קריאה לפונקציה:

א. על-פי static-scoping – ראו ביטויי call-static – בקריאה כזאת, הפונקציה רצה בסביבה המרחיבה את סביבת ה-closure.
ב. על-פי dynamic-scoping – ראו ביטויי call-dynamic – בקריאה כזאת, הפונקציה רצה בסביבה המרחיבה את הסביבה בה מתבצעת הקריאה.

לבסוף, נממש קוד בשפה SOL עבור יצירת אובייקט זוג. את המימוש שלו נכניס לסביבה גלובאלית שתאותחל עם כל הרצה של האינטרפרטר שאנחנו בונים. נתון לכם קוד חלקי (למעשה, רוב הקוד נתון לכם, עליכם להשלים חלקים מסויימים).

הערות: חשוב מאד להכניס הערות משלושה סוגים: 1. הסברים מה כל פונקציה עושה, 2. הערות המנמקות את ההשלמות שלכם, 3. הערות המספרות על תהליך הפתרון שלכם.
*****כדי לקבל ציון עובר במטלה, חובה להוסיף הערות מספקות**.

טסטים – עליכם לכתוב טסטים רבים ומגוונים למטלה שלכם. חלק ניכר מהערכת הפתרון שלכם יסתמך על הטסטים שכתבתם. הטסטים צריכים להיות מגוונים ולכסות את כל מקרי הקצה האפשריים.

השתמשו בלינק הבא לאינטרפרטר החסר הנתון.

שאלה 1 — BNF — (5 נקודות):

השלימו את כללי הדקדוק החסרים ואת הגדרת עץ התחביר האבסטרקטי SOL על פי ההגדרות במבוא מעלה ועל-פי דוגמאות ההרצה בתוך קוד המטלה הנתון לכם.

שאלה 2 — SET operations — (15 נקודות):

השלימו את הקוד החסר וכתבו הערות עבור פרוצדורות העזר לטיפול בקבוצות (SET).
השתמשו בפרוצדורות של pl המוצעות בתחתית מסמך זה.
הכניסו הערות משלושה סוגים: 1. הסברים מה כל פונקציה עושה, 2. הערות המנמקות את ההשלמות שלכם, 3. הערות המספרות על תהליך הפתרון שלכם.
הגדרות:

א. הפרוצדורה ?ismember נתונה לכם. היא לוקחת מספר ורשימת מספרים ומחזירה #t אם המספר הוא איבר ברשימה ו-#f אחרת.
ב. הפרוצדורה remove-duplicates נתונה לכם חלקית. היא לוקחת רשימת מספרים ומחזירה את הרשימה החלקית לה ללא חזרות – באופן שרירותי ניקח לכל מספר תמיד את המופע הימני ביותר של מספר זה ברשימה. לדוגמה:

```
(test (remove-duplicates '(3 4 5)) => '(3 4 5))
(test (remove-duplicates '( 3 2 3 5 6)) => '(2 3 5 6))
(test (remove-duplicates '(3 4 5 1 3 4)) => '(5 1 3 4))
(test (remove-duplicates '(1)) => '(1))
```

ג. הפרוצדורה create-sorted-set נתונה לכם חלקית. היא לוקחת רשימת מספרים ומחזירה את הרשימה החלקית לה ללא חזרות – ממויינת על פי היחס >.

ד. הפרוצדורה set-union נתונה לכם חלקית. היא לוקחת שתי רשימות מספרים ומחזירה את רשימת כל המספרים השייכים לפחות לאחת מן השתיים – ממויינת וללא חזרות.

ה. הפרוצדורה set-intersection נתונה לכם חלקית. היא לוקחת שתי רשימות מספרים ומחזירה את רשימת כל השייכים לכל אחת מן השתיים – ממויינת וללא חזרות. אין להניח שהקלט הן סדרות ממויינות וללא חזרות.

שאלה 3 — השלמת קוד ה-parse — (15 נקודות):

השלימו את הקוד החסר בפרוצדורה parse-sexpr **על פי ההגדרות מעלה והדוגמאות בתוך הקוד, בהמשך להגדרת הטיפוס** SOL.

א. טיפול בביטויים בסיסיים – קבוצה – תוחזק כסדרה ממויינת ללא חזרות של מספרים.
ב. טיפול בביטויי with – במטלה זאת, אנו נטפל בביטויי with ללא בנאי מיוחד. לחילופין, אנו נשתמש בבנאים קיימים (כלומר, נתייחס לביטויים אלה כ-syntactic sugar עבור ביטוי אחר). שימו לב לשני העניינים הבאים:
   a. ביטוי with מאפשר פרמטר יחיד, בעוד ביטוי fun דורש שני פרמטרים. חישבו כיצד להתגבר על קושי טכני זה. שימו לב שהגבלות ואיסורים בבחירת שמות הפרמטרים החלים על המתכנת – אינם חלים עליכם ככותבי האינטרפרטר (כאן מדובר על עניין טכני נוסף, אשר אינכם מחוייבים להבין).
   b. ישנם שני סוגי ביטויי call. חישבו באיזה מהם כדאי להשתמש כאן. בחלק השאלות התאורטי, הנכם מתבקשים להסביר את בחירתכם.
ג. טפלו בביטויי call-static וגם call-dynamic.
ד. שימו לב להחזיר הודעות שגיאה התואמות את הטסטים הקיימים.

הכניסו הערות משלושה סוגים: 1. הסברים מה כל פונקציה עושה, 2. הערות המנמקות את ההשלמות שלכם, 3. הערות המספרות על תהליך הפתרון שלכם.

שאלה 4 — השלמת קוד ה-eval ופרוצדורות עזר — (25 נקודות):

א. השלימו את החלקים החסרים בהגדרות הפורמליות של eval (נתונות כהערה).
ב. השלימו את הקוד החסר בפרוצדורות העזר ל-eval.
ג. השלימו את הקוד החסר בפרוצדורה eval.
   a. טפלו בבנאים העוסקים בפעולות על SET בעזרת פרוצדורות העזר.
   b. טפלו בשני סוגי הבנאים של קריאה לפונקציה. זכרו להרחיב את הסביבה המתאימה.

הכניסו הערות משלושה סוגים: 1. הסברים מה כל פונקציה עושה, 2. הערות המנמקות את ההשלמות שלכם, 3. הערות המספרות על תהליך הפתרון שלכם.

שאלה 5 — יצירת סביבה גלובלית וטיפול ב-pairs — (20 נקודות):

עתה נרצה להרחיב את השפה כך שתכיר את הפונקציות cons, first, second.
בפרט נרצה שהטסט הבא יעבוד:

```
(test (run "{with {p {call-static cons {1 2 3} {4 2 3}}}
        {with {S {intersect {call-static first p {}}
                    {call-static second p {}}}}
          {call-static {fun {x y} {union x S}}
                {scalar-mult 3 S}
                {4 5 7 6 9 8 8 8}}}}")
      => '(2 3 6 9))
```

הדרכה:

א. ראשית נזכיר כי הרעיון הוא שאנחנו מממשים pair כפונקציה שזוכרת את הערכים שאיתם אותחל ה-pair בזמן יצירתו. בפרט, הפונקציה cons תחזיר פונקציה. הפונקציות first ו-second יקבלו pair ויפעילו אותו על פונקציה שמהווה סלקטור.
כך, למשל, מישהו שמכיר את המימוש שלנו מבפנים, יוכל גם להריץ את הטסט הבא (לאחר שנסיים את התהליך):

```
(test (run "{with {p {call-static cons {1 2 3} {4 2 3}}}
        {with {foo {fun {x y} {intersect x y}}}
          {call-static p foo {}}}}")
      => '(2 3))
```

ב. כדי להתחיל בתהליך המימוש מומלץ לכם: להשלים את החלקים החסרים בשלוש שורות הקוד האפשריות הבאות (אלו שורות קוד שלא יהיו חלק מהאינטרפרטר הסופי, אבל צריכות לרוץ אם השלמתם את כל הסעיפים הקודמים).

```
(run "{with {cons {fun {f s} < -- fill in -- >}}
        cons}")
(run "{with {first {fun {p spare-param}
          {call-< -- fill in -- >}}}
      first}")
(run "{with {second < -- fill in -- >
          {< -- fill in -- > p {fun {a b} b} {}}}}
      second}")
```

שימו לב – לעיתים יתכן ותצטרכו פונקציה הפועלת על פרמטר יחיד. פתרון פשוט הוא להפוך את הפרמטר השני לפרמטר פיקטיבי שאינו בשימוש ובקריאות לפונקציה, לשלוח פשוט ערך כלשהו (לדוגמה, {}).

ג. הריצו את שלוש שורות הקוד שהשלמתם באינטרפרטר כפי שכבר כתבתם אותו. השתמשו בערכים המוחזרים כדי להשלים את הקוד:

```
(: createGlobalEnv : -> ENV)
(define (createGlobalEnv)
  (Extend 'second <-- fill in -->
      (Extend <-- fill in -->
          (Extend <-- fill in -->
              (EmptyEnv)))))
```

ד. השלימו את הקוד החסר בפרוצדורה run.

שאלה 6 — שאלות תאורטיות לגבי הפתרון שלכם — (30 נקודות):

ענו על השאלות הבאות באופן תמציתי וברור. הוסיפו את תשובותיכם כהערות בתחתית המטלה (ההגשה הינה כקובץ rkt. יחיד).

1. כתבו האם הצלחתם לפתור את המטלה ללא עזרה מאף מקור חיצוני. אם לא, פרטו.
2. מהם הטיפוסים הקיימים בשפה שהגדרנו? הסבירו את תשובתכם.
3. בשאלה 3, למעלה הראיתם כיצד ניתן להחליף ביטוי with בביטוי המתאר קריאה לפונקציה, על מנת שלא להשתמש בבנאי נפרד לביטויים אלו. כיצד התגברתם על השוני במספר הפרמטרים? האם השתמשתם ב-call-static או ב-call-dynamic? מהי החשיבות של בחירתכם?
4. אילו מן הפרוצדורות שטיפלתם בהן בשאלה 2 הן רקורסיביות ומשתמשות בקריאות זנב בלבד? מה היתרון של שימוש ברקורסית זנב בראקט?
5. הסבירו היכן בפתרון שאלה 5 השתמשתם ב-call-static והיכן ב-call-dynamic. מהי החשיבות של בחירותיכם להצלחת הפתרון? האם יכולתם להשתמש באפשרות האחרת?
6. מה היה קורה אם בטסט שניתן בשאלה 5, אם היינו מחליפים את אחד מהשימושים ב--call static לשימוש ב-call-dynamic? הסבירו את תשובתכם.

בהצלחה!!!!

---

procedure

```
(sort  Lst
       less-than?
      [#:key extract-key
       #:cache-keys? cache-keys?]) → list?
  lst : list?
  less-than? : (any/c any/c . -> . any/c)
  extract-key : (any/c . -> . any/c) = (lambda (x) x)
```

```
  cache-keys? : boolean? = #f
```

Returns a list sorted according to the *less-than?* procedure, which takes two elements of *lst* and returns a true value if the first is less (i.e., should be sorted earlier) than the second.

The sort is stable; if two elements of *lst* are "equal" (i.e., *less-than?* does not return a true value when given the pair in either order), then the elements preserve their relative order from *lst* in the output list. To preserve this guarantee, use `sort` with a strict comparison functions (e.g., `<` or `string<?`; not `<=` or `string<=?`).

The `#:key` argument *extract-key* is used to extract a key value for comparison from each list element. That is, the full comparison procedure is essentially

```
(lambda (x y)
  (less-than? (extract-key x) (extract-key y)))
```

By default, *extract-key* is applied to two list elements for every comparison, but if *cache-keys?* is true, then the *extract-key* function is used exactly once for each list item. Supply a true value for *cache-keys?* when *extract-key* is an expensive operation; for example, if `file-or-directory-modify-seconds` is used to extract a timestamp for every file in a list, then *cache-keys?* should be `#t` to minimize file-system calls, but if *extract-key* is `car`, then *cache-keys?* should be `#f`. As another example, providing *extract-key* as `(lambda (x) (random))` and `#t` for *cache-keys?* effectively shuffles the list.

Examples:

```
> (sort '(1 3 4 2) <)
'(1 2 3 4)
> (sort '("aardvark" "dingo" "cow" "bear") string<?)
'("aardvark" "bear" "cow" "dingo")
> (sort '(("aardvark") ("dingo") ("cow") ("bear"))
        #:key car string<?)
'(("aardvark") ("bear") ("cow") ("dingo"))
```

```
(map proc lst ...+) → list?
  proc : procedure?
  lst : list?
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.
```

```
Examples:
> (map (lambda (number)
         (+ 1 number))
       '(1 2 3 4))
'(2 3 4 5)
> (map (lambda (number1 number2)
         (+ number1 number2))
       '(1 2 3 4)
       '(10 100 1000 10000))
'(11 102 1003 10004)
```

---

procedure

```
(filter pred lst) → list?
  pred : procedure?
  lst : list?
```

Returns a list with the elements of *lst* for which *pred* produces a true value. The *pred* procedure is applied to each element from first to last.

Example:

```
> (filter positive? '(1 -2 3 4 -5))
'(1 3 4)
```

---

procedure

```
(append lst ...) → list?
  lst : list?
(append lst ... v) → any/c
  lst : list?
  v : any/c
```

When given all list arguments, the result is a list that contains all of the elements of the given lists in order. The last argument is used directly in the tail of the result. The last argument need not be a list, in which case the result is an “improper list.”

Examples:

```
> (append (list 1 2) (list 3 4))
'(1 2 3 4)
> (append (list 1 2) (list 3 4) (list 5 6) (list 7 8))
'(1 2 3 4 5 6 7 8)
```