גירסה 1.01 – 3.11.2004

LISP

ניר אדר

Nir Adar
Email: underwar@hotmail.com
Home Page: http://underwar.livedns.co.il

אנא שלחו תיקונים והערות אל המחבר.

גירסאות

גירסה 1.01 – 3.11.2004	– הוספה רשימת המקורות לסוף המסמך.
גירסה 1.00 – 23.05.2004	– גירסה ציבורית ראשונה.

תוכן עניינים

LISP

הקדמה

למה ללמוד LISP? התשובה היא בדרך כלל: אין סיבה. הסיבה היחידה שמסמך זה נכתב, ושכנראה הביאה אותך לקרוא בו, היא שהגעת לקורס באוניברסיטה שצריך להשתמש בו ב-LISP. מלבד באוניברסיטאות, אין מקומות רבים בהם משתמשים בשפה זו.
LISP הומצאה אי שם בשנות החמישים. מקור השם LISP הינו List Processor. כפי שהשם מרמז, רשימות מהוות חלק משמעותי ביותר במרכיבי השפה. פעולות על רשימות תוכננו כך שיתבצעו במהירות. LISP נחשבת אחת השפות העקריות לתכנות של סוכני בינה מלאכותית ומערכות לומדות.

מאפייני השפה

- שפה פונקציונלית
- תמיכה מובנת ברשימות
- ניהול זיכרון אוטומטי – garbage collector
- תומך בסוגי משתנים דינמיים וסטטיים – ניתן להגדיר משתנים בלי לציין את סוגם, וכן ניתן לציין את סוגם לצורך יעילות.
- פונקציות כערכים ממחלקה ראשונה: פונקציות ב-LISP הן אובייקטים לכל דבר. ניתן לשמור פונקציה בתוך משתנה, ניתן להעביר פונקציה כפרמטר לפונקציה אחרת ולהתייחס אליה בצורה מילולית.
- שפת Prefix - האופרטורים מופיעים לפני האופרנדים.
- כל ה-symbols של השפה מתוארים על ידי אותיות אנגליות גדולות (אותיות קטנות יחשבו באופן אוטומטי כגדולות).

רשימות

מקור השם LISP הינו List Processor. כאמור רשימות מהוות חלק משמעותי ביותר במרכיבי השפה.

תחביר רשימות: $(\text{name arg}_1 \ldots \text{arg}_n)$ כאשר רשימה יכולה להתפרש כפונקציה, ואז name הוא שם הפונקציה ו- $\text{arg}_1 \ldots \text{arg}_n$ הינם הפרמטרים לפונקציה, או לחילופין רשימה יכולה להתפרש בצורות מיוחדות לפי ה-name, ואז משמעות הארגומנטים תלוייה בפירוש.

פיענוח פונקציות: פונקציות מפורשות על ידי חוק הפיענוח הבא:
1. הארגומנטים מפורשים משמאל לימין.
2. ערכי הארגומנטים מועברים לפונקציה ששמה name.
3. אם ארגומנט כלשהו הוא פונקציה בעצמו, הוא מפוענח בעזרת אותו כלל.

חריגים: ישנם אופרטורים מיוחדים שלא מצייתים לחוק הפירוש הכולל, כגון if, quote.
דוגמא לפונקציה: (* 7 8)

quote: חוק הפירוש של quote הוא: "אל תעשה כלום". אופרטור quote לוקח ארגומנט בודד ומחזיר אותו כפי שהוא (באופן מילולי). דוגמא: (quote (+ 3 5)) ((+ 3 5)). לצורך הנוחות, ניתן להשתמש בסימן ' במקום במילה quote, למשל: '(+ 3 5)'.

בנית רשימה

שם הפונקציה:	cons
תחביר:	(**cons** *object list*)
פעולה:	הפונקציה מקבלת אובייקט ורשימה ומחזירה רשימה חדשה שהאיבר הראשון שלה הוא object וזנבה הרשימה הוא list.
דוגמאות:	(cons 'a (cons 'b nil)) nil זוהי רשימה ריקה. כאשר נכתוב למשל (cons 3 nil) נקבל רשימה בה האיבר היחיד הוא 3.

שם הפונקציה:	list
תחביר:	(**list** $object_1$ $object_2 \ldots object_n$)
פעולה:	הפונקציה מקבלת מספר אובייקטים ומחזירה רשימה חדשה שאבריה הם אובייקים אלו.
דוגמאות:	(list 'a 'b) → (A B) (list '(+1 2) (+ 1 2)) → ((+ 1 2) 3) '(list (+ 1 2) (+ 1 2)) → (list (+ 1 2) (+ 1 2))

שם הפונקציה:	append
תחביר:	(**append** $list_1$ $list_2 \ldots list_n$)
פעולה:	הפונקציה מקבלת מספר רשימות ומחזירה רשימה חדשה שאיבריה הם שרשור האלמנטים של הרשימות, לפי הסדר בו הרשימות נכתבו.
דוגמאות:	(append '(a b) '(c d) '(e)) → (A B C D E) (append) → nil

גישה לרשימה

שם הפונקציה:	car / first
תחביר:	(**car** *list*) (**first** *list*)
פעולה:	הפונקציה מקבלת רשימה ומחזירה את האיבר הראשון ברשימה. איבר זה יכול להיות אטום ויכול להיות רשימה בעצמו.
דוגמאות:	(car '(Hello World)) → HELLO (first '((A B) C)) → (A B) (car ()) → NIL

שם הפונקציה:	cdr / rest
תחביר:	(**cdr** *list*) (**rest** *list*)
פעולה:	הפונקציה מקבלת רשימה ומחזירה את כל איברי הרשימה פרט לאיבר הראשון. הפונקציה תמיד תחזיר רשימה.
דוגמאות:	(cdr '(Hello World)) → (WORLD)

שם הפונקציה:	first, second, third, …, tenth
תחביר:	(**first** *list*) / (**second** *list*) / …
פעולה:	הפונקציה מקבלת רשימה ומחזירה את האיבר הראשון/שני/... שבה.
דוגמאות:	(first '(1 2 3)) → 1

שם הפונקציה:	nth
תחביר:	(**nth** *n list*)
פעולה:	הפונקציה מקבלת אינדקס n ורשימה list ומחזירה את האיבר ה-n ברשימה (האיבר הראשון נחשב איבר מספר 0).
דוגמאות:	(nth 2 '(1 2 3)) → 3

שם הפונקציה:	nthcdr
תחביר:	(**nthcdr** *n list*)
פעולה:	אקוויוולנטי לקריאה n פעמים רצופות לפונקציה cdr.
דוגמאות:	(nthcdr 1 '(1 2 3)) → (2 3) (nthcdr 0 '(1 2 3)) → (1 2 3)

השמות

הפונקציות setq ו-setf משמשות להשמת ערכים בסמלים (symbols) או במיקומים (locations).

תחביר:

(**setq** $symbol_1$ $value_1$ … $symbol_n$ $value_n$)
(**setf** $place_1$ $value_1$ … $place_n$ $value_n$)

נקודות:

- setq נותן לכל משתנה $symbol_i$ את הערך של הביטוי $value_i$.
- setf שומר את התוצאה של $value_i$ במקום המתאם ל-$place_i$, כאשר מקום חוקי יכול להיות משתנה, קריאה לפונקציה וכדו'.
- שתי הפונקציות מחזירות את הערך של ה-value האחרון.

דוגמאות:

1. (setf x 7 y 9)
   ב-x יושם 7, ב-y יושם 9 והפונקציה תחזיר 9.
2. (setf x '(1 2))
   ב-x תושם רשימה.
3. (setq y '(1 2))
   ב-y תושם רשימה.
4. (setq y '(1 2 3))
   (setq (first y) 0)
   שגיאה!
5. (setq y '(1 2 3))
   (setf (first y) 0)
   y שווה ל-(0 2 3).

setf מכלילה למעשה את פעולת setq. אם נרצה, נוכל להשתמש ב-setf תמיד במקום ב-setq.

פונקציות

הגדרת פונקציות

הגדרת פונקציות נעשית על ידי הפקודה defun.

תחביר:

(**defun** *name* ($param_1$ $param_2$ … $param_3$) *function-body*)

דוגמא:

```
(defun first-and-third (l)
      (list (first l) (third l))
)

(setf my_var '(this is a sentence))
(first-and-third my_var)
```

תוצאה:

```
(THIS A)
```

קריאה לפונקציה

ישנן מספר צורות לקרוא לפונקציה.

1. את האפשרות הראשונה כבר ראינו: רשימה שהאיבר הראשון בה הוא שם הפונקציה ולאחריו הפרמטרים.

(**f-name** arg_1 … arg_n)

דוגמא: (+ 1 2 3 4) יחושב ל 10

2. שימוש ב-funcall:

(**funcall** f-name arg_1 … arg_n)

דוגמא: (funcall #'+ 1 2 3 4) יחושב ל-10.

'# זהו יוצר מעין "מצביע" לפונקציה +, כדי ש-funcall יוכל לקבל אותה.

3. apply – הפעלת פונקציה על רשימה.

(**apply** f-name argument-list)

דוגמא: (apply #'+ (1 2 3 4)) יחושב ל-10.

התניות

שם ההתניה:	if
תחביר:	(**if** *test then [else]*)
פעולה:	מפענח את ערך test. אם מוחזר אמת (t או ערך שאינו nil) אז הביטוי then מבוצע וערכו מוחזר. אחרת הערך של else מבוצע וערכו מוחזר. אם אין else, מוחזר nil.
דוגמאות:	(setf x 11) (if (> x 10) (- x 10) x) → 1 (if (> x 100) x) → NIL

שם ההתניה:	when
תחביר:	(**when** *test expression**)
פעולה:	מפענח את ערך test. אם מוחזר אמת יבוצע הביטוי/ביטויים שאחריו, ויוחזר ערכו של הביטוי האחרון. אם test הינו שקר, יוחזר NIL.

שם ההתניה:	unless
תחביר:	(**unless** *test expression**)
פעולה:	מפענח את ערך test. אם מוחזר שקר יבוצע הביטוי/ביטויים שאחריו, ויוחזר ערכו של הביטוי האחרון. אם test הינו אמת, יוחזר NIL. פעולה זו הינה ההופכית ל-when.

שם ההתניה:	cond
תחביר:	(**cond** (*clause*$_1$)...(*clause*$_n$)) clause = *test*$_i$ expression$_{i1}$...expression$_{in}$...
פעולה:	זהה ל-switch ב-C. ה-test נבדק עבור כל ביטוי אחד אחרי השני. אם אחד מהתנאים הוא אמת, הביטויים שאחריו מבוצעים.

דוגמאות לתנאים:
התנאי listp מקבל פרמטר ומחזיר אמת אם הפרמטר הוא רשימה.
התנאי null מקבל פרמטר ומחזיר T אם הפרמטר הוא רשימה ריקה או NIL.

פונקציות בוליאניות

בשפת LISP הביטוי NIL הינו שקר, וכל ביטוי אחר הינו אמת.
הביטויים מפוענחים באופן דומה לפיענוח בשפת C: and עוצר מיידית אם הוא הגיע לביטוי שהוא NIL, ו-or עוצר כאשר הוא מגיע לביטוי שאינו NIL.
ערך האמת מוגדר בשפת להיות הסמל T. (אם כי כל ביטוי שאינו NIL הוא אמת).

שם הפונקציה:	and
תחביר:	(**and** expression$_1$ ... expresion$_n$)
פעולה:	הפונקציה מבצעת and בין הפרמטרים השונים.

שם הפונקציה:	or
תחביר:	(**or** expression$_1$ ... expresion$_n$)
פעולה:	הפונקציה מבצעת or בין הפרמטרים השונים.

שם הפונקציה:	not
תחביר:	(**not** expression)
פעולה:	הפונקציה מבצעת not לוגי על הפרמטר אותו היא מקבלת.

דוגמא

נציג דוגמא לשימוש ב-cond ובשפת LISP בכלל. כידוע, nand הינה מערכת שלמה.
נציג כעת מימוש של nand על ידי cond, ולאחר מכן נממש את האופרטורים הבוליאניים השונים בעזרת nand:

```
(defun my-nand (a b)
        (cond ((null a) t) ((null b) t) (t nil))
)


(defun my-not (a)
        (my-nand a a)
)


(defun my-and (a b)
        (my-not (my-nand a b))
)

(defun my-or (a b)
        (my-not (my-and (my-not a) (my-not b)))
)
```

```
(defun my-nor (a b)
        (my-and (my-not a) (my-not b))
)

(defun my-xor (a b)
        (my-or
                (my-and a (my-not b))
                (my-and (my-not a) b)
        )
)
```

קלט & פלט

קלט בסיסי

- read – קליטת ביטויים.
- read-char – קליטת תו.
- read-line – קריאת שורה עד סופה.

פלט בסיסי:

- prin1 מציגה אובייקטים המיועדים לקריאה על ידי בני אדם.
- princ מציגה אובייקטים המיועדים לקריאה על ידי תוכנות.
- print פועלת בדומה ל-prin1 אבל מוסיפה מעבר שורה לפני ההדפסה.

ערוצים:
ערוצים ב-LISP הם אובייקטים המציינים מקורות או יעד לתווים. כדי לקרוא או לכתוב מקבצים אנו פותחים ערוץ כתיבה אליהם. בדומה ל-C, אנחנו מתייחסים אל הקלט ואל הפלט הסטנדרטיים כערוצים.
שמם ב-LISP: *standard-output*, *standard-input*.

פלט מעוצב:
הפקודה format משמשת להדפסת פלט מעוצב אל ערוץ. הפקודה פועלת בדומה לפונקצית printf הקיימת בשפות נורמליות.
הפרמטר הראשון שלו הוא ערוץ (או NIL או T), ויש לו אפס או יותר פרמטרים נוספים עבור הדפסה – מחרוזת בקרה ואובייקטים.
כאשר format מקבלת T, הפלט הוא לקלט הסטנדרטי. כאשר היא מקבלת NIL הפלט מוחזר בלבד כמחרוזת. בכל שאר המקרים, הערך המוחזר על ידי הפונקציה הוא NIL.
מספר הנחיות לשימוש:

- ~f עבור מספרים ממשיים.
- ~% כדי לעבור לשורה חדשה
- ~a הדפסת כל דבר – דומה ל-princ
- ~s הדפסת כל דבר, בדומה ל-prin1

דוגמא:

```
(format t "~a plus ~s is ~f" "two" "two" 4)
two plus "two" is 4.0
```

משתנים

משתנים לוקליים
נגדיר משתנים לוקליים בעזרת LET, לדוגמא:

```
(let
      ((var1 val1) ... (varn valn))
      let-body
)
```

המשתנים המוגדרים קיימים מהתחלת הצהרת let עד סופה.

משתנים מיוחדים
שני סוגי משתנים מיוחדים:

1. (**defvar** variable [init-value])

נהוג לסמן משתנים מיוחדים עם כוכביות, לדוגמא:

```
(defvar *my-var* 17)
```

2. (**defparameter** variable init-value)

הגדרת פרמטר שנשאר קבוע ולא משתנה במהלך ריצת התוכנית.

ביטויי למבדה

הרעיון: בשפת LISP אנחנו יכולים להתייחס לפונקציות כאל משתנים לכל דבר. בין היתר אנחנו יכולים ליצור פונקציות חדשות תוך כדי ריצה, להחזיר פונקציות ולשמור אותן בתוך משתנה וכמובן גם לקבל פונקציות.
בעזרת LAMBDA אנו יוצרים פונקציות חדשות. תחביר:
(**lambda** parameters-list body)

דוגמאות:

```
((lambda(x) (+ x 2)) 4) → 6

(setf my_func (lambda(x) (+ x 2)))
(funcall my_func 3) → 5
```

ניתן בעזרת ביטויי למבדה ליצור פונקציות בזמן ריצה, לדוגמא:

```
(defun add-constant (constant)
      #'(lambda (number)
            (+ number constant)
      )
)

(setf add-5 (add-constant 5))
(funcall add-5 10) → 15
```

מבנים איטרטיביים

DOTIMES

מבנה זה מזכיר את for בשפת C.

תחביר:

```
(DOTIMES (var integer) loop-body)
```

לולאה: כל עוד var מתחת ל-integer בצע את loop-body.

לדוגמא:
הפקודה הבאה תגדיר משתנה חדש x שירוץ בין הערכים 0 ל-9 ותדפיס את ערכי x על המסך:

```
(dotimes (x 10) (print x) )
```

DOLIST

מבנה זה מזכיר את הלולאה foreach בשפות בהן היא קיימת. הלולאה מתבצעת על כל איברי הרשימה הנתונה.

תחביר:

```
(DOLIST (var list) loop-body)
```

לולאה: עבור כל משתנה var ב-list בצע את loop-body.

לדוגמא:

```
(dolist (var '(a b c d)) (print var) )
```

ייצוג פנימי של רשימות

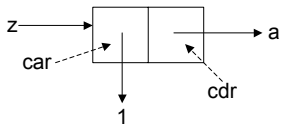
מבוא

נציג כעת כיצד LISP שומרת את הרשימות שאנו יוצרים. ידיעה זו תגרום לנו להיות מסוגלים לבצע פעולות בצורה יעילה יותר. כמו כן, נבין יותר את התנהגות השפה במקרים שונים.

כאשר אנו משתמשים ב-cons, נוצר למעשה רכיב מידע בשם cons (או "**זוג מנוקד**") בעל שני שדות המכונים car ו-cdr. כל אחד משדות אלו יכול להיות כל רכיב חוקי של שפת LISP.

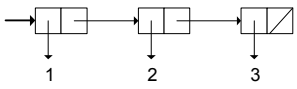
לדוגמא:

```
(setf z (cons 1 'a)) → (1 . A)
(car z) → 1
(cdr z) → A
```



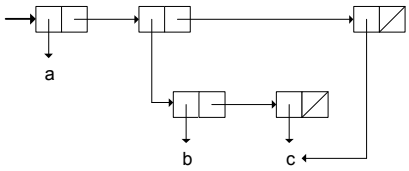
רשימה בשפת LISP היא למעשה שרשרת של cons המקושרים ביניהם על ידי שדה ה-cdr. האיבר האחרון ברשימה מקושר בד"כ ל-nil.

```
(list 1 2 3) ≡ (cons 1 (cons 2 (cons 3 nil)))
```



נביט כיצד מיוצגת הרשימה הבאה בזכרון:

```
(list 'a '(b c) 'c)
```

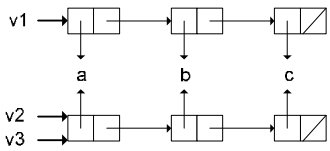


נשים לב כי שמות סימבוליים משותפים בשפה. כאשר יש שתי רשימות המכילות את C, שתיהן למעשה מצביעות אל אותו איבר.

דוגמא:

```
> (setf v1 '(a b c))
(A B C)
> (setf v2 '(a b c))
(A B C)
> (setf v3 v2)
(A B C)
```

בזכרון v1, v2, v3 יראו כך:



כעת נשתמש בפונקציות eq ו-equal על מנת להשוות בין הרשימות, ונראה את ההבדלים:

```
(equal v1 v2) → T
(equal v2 v3) → T

(eq v1 v2) → NIL
(eq v2 v3) → T
```

eq מבצעת השוואה בין אובייקטים. מוחזר שוויון רק אם מדובר על אותו אובייקט ממש.
equal תחזיר אמת גם במקרה בו הרשימות שונות, וזאת בתנאי שאיברי הרשימות הינם זהים.

נציג כעת את האופרטורים השונים של השוואה ביתר פירוט.

בדיקת שיוויון

שפת LISP מכילה מספר אופרטורים שונים לצורך בדיקת שוויון הנבדלים בהתנהגותם.

אופרטור	משמעות	דוגמא
=	השוואה בין מספרים בלבד	(= 3 3.0) → T (= 'a 'a) → ERROR
eq	השוואה בין אובייקטים שוויון רק אם מדובר על אותו אובייקט.	(eq 'a 'A) → T
eql	eq או מספרים/תווים זהים. מספרים עשרוניים ושלמים לא ניתנים להשוואה.	(eql '(1) '(1)) → NIL (eql 17 17) → T (eql 17 17.0) →NIL
equal	eql או במקרה שמדובר ברשימות – אם כל איברי הרשימות זהים על פי eql	(equal '(1) '(1)) → T (equal (11 "a") (11.0 "a")) → NIL
equalp	פועל כמו equal אך בנוסף מחשיב אותיות גדולות וקטנות לזהות, וכן ממיר בין מספרים מסוגים שונים	(equalp '(1 "xY") '(1.0 "Xy")) → T

Destructive Functions

פונקציות מתמטיות רגילות אינן משנות את הפרמטרים שלהן. פונקציות אשר משנות את הפרמטרים שלהן מכונות destructive functions. שפת LISP מכילה מספר פונקציות כאלו. השימוש העיקרי שלהן הוא לשיפור היעילות של התוכנות שאנו כותבים.

שם הפונקציה:	rplaca
תחביר:	(**RPLACA** cons object)
פעולה שקולה:	(**RPLACA** x y) ≡ (**setf** (**car** x) y)
פעולה:	משנה את ה-car של ה-cons להיות object ומחזיר את ה-cons לאחר השינוי.
דוגמא:	> **(setf sentence '(small money big problems))** (SMALL MONEY BIG PROBLEMS) > **(rplaca sentence 'big)** (BIG MONEY BIG PROBLEMS) > **sentence** (BIG MONEY BIG PROBLEMS)

שם הפונקציה:	rplacd
תחביר:	(**RPLACD** cons object)
פעולה שקולה:	(**RPLACD** x y) ≡ (**setf** (**cdr** x) y)
פעולה:	משנה את ה-cdr של ה-cons להיות object ומחזיר את ה-cons לאחר השינוי.

הפונקציה NCONC

פונקציה זו הינה גירסת destructive function של הפונקציה append.

נניח כי הרצנו את הפקודות הבאות:

```
> (setf abc '(a b c))
(A B C)
> (setf xyz '(x y z))
(X Y Z)
```

נפעיל את nconc ונראה את התוצאה, ואת ערכי abc ו-xyz לאחר הקריאה:

```
> (nconc abc xyz)
(A B C X Y Z)

> xyz
(X Y Z)

> abc
(A B C X Y Z)
```

abc השתנה כדי להכיל את תוצאת השרשור.

מה שבעצם nconc ביצעה זה שינוי ה-cdr האחרון של abc כדי שיצביע על xyz.

להשוואה, נביט בתוצאת append על ערכים זהים:

```
> (append abc xyz)
(A B C X Y Z)

> xyz
(X Y Z)

> abc
(A B C)
```

append החזירה רשימה חדשה, וערכי abc, xyz נשארו ללא שינוי.

שימוש בפונקציות אשר משנות את הפרמטרים שלהן הוא מסוכן, אבל יעיל. שימוש ב-append כל פעם יוצר עותק של הרשימה. שימוש ב-nconc בצורה נכונה יכול לשפר את יעילות התוכנית בסדר גודל.

רקורסית זנב

שפת LISP היא שפה בה אנו מסתמכים במידה רבה על שימוש ברקורסיות. כידוע משפות אחרות שימוש ברקורסיה לרוב פוגע בביצועי התוכנית. נרצה לגרום לכך שלמרות שאנו משתמשים ברקורסיה התוכנית שלנו תהיה יעילה.
המהדר, כאשר הוא יכול, הופך את הרקורסיות שאנחנו כותבים ללולאות כאשר הוא ממש אותן, ובכך אנו משיגים את השיפור שאנו מעוניינים בו.

כדי שלמהדר יהיה קל לעשות זאת, אנו נעדיף כשאפשר להשתמש ברקורסיה הנקראת רקורסית זנב.
הרעיון של רקורסיה זו: כאשר אנחנו מגיעים לתנאי העצירה של הרקורסיה, יש בידינו את תוצאת החישוב, וכל מה שעלינו לעשות זה להחזיר את התוצאה.

לדוגמא, כך תראה פונקציה לחישוב עצרת ללא רקורסית זנב:

```
(defun azeret (n)
      (if (= n 0)
            1
            (* n ( azeret (- n 1)))
      )
)
```

כאשר אנחנו מגיעים לתנאי העצירה, אנחנו מתחילים לעלות כדי להחזיר את התוצאה הסופית.

עם רקורסית זנב, הפונקציה תראה כך:

```
(defun azeret-aux(n res)
      (if (= n 0)
            res
            (azeret-aux (- n 1) (* res n))
      )
)

(defun azeret (n)
      (azeret-aux n 1)
)
```

הגדרנו פונקצית עזר, המקבלת כפרמטר נוסף את התוצאה עד כה. כאשר אנו מגיעים לסוף הרקורסיה, אנו פשוט מחזירים אותו.

מבנים

יצירת מבנה:

(**defstruct** name $slot_1$ $slot_2$ ... $slot_n$)

באופן אוטומטי נוצרים:
constructor בשם **make**-name.
פונקציה מזהה בשם **p**-name.
פונקצית השמה וקבלה של כל אחד מהפרמטרים:

name-$slot_1$, name-$slot_2$, …, name-$slot_n$

שינוי שדות המנה נעשה על ידי setf.

לדוגמא:
הגדרת מבנה ופונקציה המשתמשת בו:

```
; Struct the holds a state with a list of the states that came
before.
(defstruct Path state prev_states depth)

; Checks if the current state is the goal
(defun is_goal (the_path)
      (eql (Path-state the_path) 0)
)
```

דוגמא לאתחול מבנה:

```
(setf n (make-Path :state '0 :prev_states nil :depth 0))
```

מערכים

יצירת מערך נעשית על ידי הפונקציה make-array.

דוגמא ליצירת מערך חד ממדי:

```
(setf numarray (make-array 10))
```

הפונקציה תיצור מערך חד ממדי שכל איבריו מאותחלים להיות NIL.

יצירת מערך דו ממדי שאבריו מאותחלים להיות NIL:

```
(setf twodimension (make-array '(4 4)))
```

יצירת מערך חד ממדי שכל האיברים בו הם 20:

```
(setf initarray (make-array 10 :initial-element 20))
```

גישה למערת בעזרת הפקודה aref:

```
(aref array subscripts)
```

שינוי איבר במערך, בעזרת setf.

דרך נוספת להגדיר מערך ולאתחל לתוכו ערכים היא בעזרת הפונקציה vector:

```
(vector 1 2 3 4 5)
```

הפונקציה תיצור מערך עם 5 תאים מאותחלים.

Hash Tables

יצירת hash-table בעזרת הפונקציה make-hash-table.
לדוגמא:

```
(setf a (make-hash-table))
(setf (gethash 'color a) 'brown)
(gethash 'color a) → brown
```

תחביר:

(**make-hash-table** :test :size :rehash-size)

לדוגמא:

```
(make-hash-table :test #'equal :size 100))
```

פונקציה לבדיקה האם אוביקט הוא hash table:

(**hash-table-p** object)

גישה ל-hash:

(**gethash** key hash-table)

דוגמא:

```
(setf (gethash "Moshe Cohen" students) '(100 100))
(gethash "Moshe Cohen" students) → (100, 100)
```

מחיקת כניסה מה-hash:

(**remhash** key hash-table)

מחיקת כל איברי ה-hash:

(**clrhash** hash-table)

מקורות

1. סאהר אסמיר (2004), תרגולים בבינה מלאכותית.

2. Simon Fraser University (2004), Imperative vs Functional Programming in LISP.

3. University Mozart Mozarteum Salzburg (2002), Iteration and the loop Macro

4. http://ice.dju.ac.kr (1998), Common LISP Hints

5. Guy Steele (1998), Common LISP