Lisp

Ali Zargar

goldsmith.ir

aizrcom@gmail.com

فهرست

مقدمه

جان مک کارتی و دانشجویانش اولین بار در سال 1958 برای پیاده سازی لیسپ شروع به کار کردند.

بعد از زبان فرترن زبان لیسپ یکی از قدیمی ترین زبانهاست که هنوز هم در حال استفاده است آنچه بیشتر قابل توجه است این است که آن هنوز جلودار تکنولوژی زبان های برنامه سازی می باشد.

برنامه نویسانی که با زبان لیسپ آشنا هستند به شما خواهند گفت که این زبان به صورت مجزا ایجاد شده است.

چیزی که زبان لیسپ را متمایز و مشخص می کند این است که برای رشد و تکامل طراحی شده . شما می توانید از لیسپ برای تعیین اپراتورهای جدید لیسپ استفاده کنید. که به عنوان انتزاعی معروف شده (به عنوان مثال برنامه نوسی شی گرایی) که پیاده سازی آن در لیسپ آسان می باشد.

۱-۱- ابزارهای جدید

چرا یادگیری زبان لیسپ؟ زیرا آن به شما اجازه و امکان انجام کارهایی که در زبانهای دیگر نمی توان انجام داد را می دهد. اگر شما فقط بخواهید یک تابع حاصل جمع اعداد کمتر از n را بنویسد آن برنامه شبیه به زبان c خواهد بود.

```
; Lisp
(defun sum (n)
  (let ((s 0))
    (dotimes (i n s)
      (incf s i))))
```

```
/* C */
int sum(int n){
  int i, s = 0;
  for(i = 0; i < n; i++)
    s += i;
  return(s);
}
```

اگر شما تنها نیاز به انجام چنین کار ساده ای داشته باشید مهم نیست که شما از چه زبانی استفاده می کنید.

ویژگی منحصر به فرد زبان لیسپ تعریف برنامه های لیسپ به عنوان ساختمان داده می باشد یعنی شما می توانید برنامه هایی برای نوشتن برنامه های دیگر بنویسید که به آنها ماکرو گفته می شود و برنامه نویسان با تجربه بیشتر زمانها از آنها استفاده می کنند.

۲-۱- تکنیک های جدید

همانطور که گفته شد لیسپ به شما امکاناتی می دهد که دیگر زبانها انجام نمی دهند. به طور جداگانه چیزهای جدیدی که همراه با لیسپ آمد از قبیل مدیریت اتوماتیک حافظه ، اعلان تایپ ، بستارها (closure) و غیره نوشتن هر برنامه ای را راحتر می کند.

لیسپ برای توسعه پذیر بودن طراحی شده این یعنی به شما اجازه می دهد که اپراتورهای جدید خود را تعریف کنید. این کار شدنی است چرا که خود زبان لیسپ از توابع و ماکروها ، همانند برنامه های ما ساخته شده است. بنابراین مشکلی برای توسعه زبان لیسپ برای نوشتن برنامه های ما وجود ندارد. به این روش برنامه نویسی پایین - بالا (down - top) گفته می شود. یک برنامه پایین – بالا (down – top) میتواند به عنوان یک سری از لایه ها نوشته شود که هر لایه وظیفه مرتب سازی زبان برنامه سازی لایه ی بالایی را دارد. کلا این زبان بصورت لایه ای کار می کند و شما می توانید بر روی لایه های قبلی لایه ای جدید بسازید.

۳-۱- یک رویکرد جدید

سبک لیسپ از سال 1960 تا به حال در حال تحول است. برنامه نویسی در حال حاضر تحت تاثیر تغییرات است ، یک نمونه ی جدید که " شی گرایی پویا " نامیده می شود در لیسپ خلاصه می شود. این به این معنی نیست که تمام نرم افزارها ظرف چند سال آینده با لیسپ نوشته می شوند.

لیسپ در حال حاضر در دانشگاهها ، آزمایشگا های تحقیقاتی وچند شرکت پیشرو در این زمینه استفاده می شود.

۱-۲- فرم (Form)

هر سیتم لیسپ شامل یک روش محاوره ی جلو – عقب (front - end) خواهد بود که به آن سطح بالا (top level) گفته می شود. یعنی شما عبارات را در سطح بالا وارد می کنید و سیتم مقدار آنها را نشان می دهد.

لیسپ معمولا پرامپتی را نشان می دهد که به شما می گوید منتظر وارد شدن عبارات می باشد. اکثر پیاده سازی های زبان لیسپ از > برای نشان دادن پرامپت سطح بالا استفاده می کنند.

یک نمونه ی ساده از عبارت لیسپ که یک مقدار عدد صحیح است در زیر نشان داده شده است اگر ما در پرامپت مقدار 1 را وارد کنیم:

```
> 1
1
>
```

سیستم مقدار آن را چاپ خواهد کرد و در ادامه نمایش پرامپت بیانگر آماده بودن برای ورود عبارات دیگر است.

ارزیابی و مقداریابی عددی مانند 1 خود آن عدد می باشد.

اگر ما بخواهیم دو عدد را با هم جمع کنیم:

```
> (+ 2 3)
5
```

در عبارت (3 2 +) به + عملگر (operator) و به اعداد 2 و 3 آرگومانها (arguments)گفته می شود.

در زندگی روزمره ما عمل جمع را به صورت 3 + 2 نشان می دهیم اما در لیسپ اپراتور + ابتدا و آرگومانها بعد از آن می آیند ، تمام عبارات در یک جفت پرانتز قرار می گیرند. که به این روش نشانه گذاری prefix گفته می شود چرا که اپراتور ها در ابتدا قرار گرفته اند.

این روش شاید در ابتدا عجیب به نظر برسد اما در حقیقت این نوع نشانه گذاری یکی از بهترین قابلیت های زبان لیسپ است.

برای مثال اگر ما بخواهیم دو عدد را با هم جمع کنیم ، در روش نشانه گذاری معمولی از اپراتور + دو برار استفاده می کنیم:

2 + 3 + 4

در لیسپ فقط آرگومانها را با یکدیگر جمع می کنیم:

(+ 2 3 4)

در روش متداول استفاده از + فقط برای دو آرگومان استفاده می شود یکی سمت چپ و یکی سمت راست.

نشانه گذاری prefix انعطاف پذیر است یعنی در لیسپ به اپراتور + می توان هر تعداد آرگومان داد.

```
> (+)
0
> (+ 2)
2
> (+ 2 3)
5
```

> (+ 2 3 4)
9
> (+ 2 3 4 5)

14

بدلیل اینکه اپراتورها تعداد متفاوتی از آرگومانها را می توانند بگیرند ما باید ابتدا و انتهای آنها را با پرانتز مشخص کنیم.

عبارات می توانند تو در تو باشند. یعنی آرگومانهای یک عبارت خود یک عبارت باشند.

> (/ (- 7 1) (- 4 2))

3

که به این عبارت 1-7 تقسیم بر 2-4 گفته می شود.

یکی دیگر از زیبایی های نشانه گذاری لیسپ : این است که تمام عبارات یا اتم ها (atoms) مانند عدد 1 هستند یا لیست ها (lists) مانند صفر یا تعداد زیادی عبارت داخل پرانتز. در اینجا چند عبارت معتبر آورده شده:

2 **(+ 2 3)** **(+ 2 3 4)** **(/ (- 7 1) (- 4 2))**

ما می بینیم که تمام کدهای لیسپ به این شکل هستند . یک زبانی مانند C دارای شکل نحوی پیچیده ای می باشند : عبارات ریاضی از نشانه گذاری infix استفاده می کنند ، فراخوانی توابعی که از نشانه گذاری prefix برای مرتب سازی با تعیین آرگومانها بوسیله ویرگول استفاده می کنند ، عباراتی که با سمی

کالون محدود می شوند و بلوک هایی که با براکت مشخص می شوند. در لیسپ ما از یک نشانه گذاری ساده برای بیان تمام این مقصود ها استفاده می کنیم.

۲-۲ - ارزیابی (Evaluation)

در قسمت قبل ما عبارات را در سطح بالا وارد کردیم و لیسپ مقدار آنها را نشان داد در این قسمت ما به چگونگی ارزیابی عبارات می پردازیم.

در لیسپ + یک تابع است و عباراتی مانند (+ 2 3) یک فراخوانی تابع است.

زمانی که لیسپ یک فراخوانی تابع را ارزیابی می کند این کار در دو مرحله انجام می شود:

1- ابتدا آرگومانها از چپ به راست ارزیابی می شوند . در این مورد آرگومانها به مقدار خود ارزیابی می شوند یعنی مقدار آرگومانها به ترتیب 2 و 3 است.

2- مقادیر آرگومانها ، بعد از عبور از تابع که به آن اپراتور گفته می شود. در مثال بالا تابع + مقدار 5 را بر می گرداند.

اگر هر یک از آرگومانها تابع خود را فراخوانی کنند آنها مطابق با همین قوانین ارزیابی می شوند. بنابراین زمانی که ((2 4-) (1 7-) /) ارزیابی شد ، اتفاقاتی که رخ می دهد شامل:

1- لیسپ (1 7 -) را ارزیابی می کند: 7 به 7 ارزیابی شده و 1 به 1 ارزیابی می شود. این مقادیر بعد از عبور از تابع – مقدار 6 را برمی گرداند.

2- لیسپ (2 4 -) را ارزیابی می کند: 4 به 4 ارزیابی شده و 2 به 2 ارزیابی می شود.این مقادیر بعد از عبور از تابع – مقدار 2 را بر می گرداند.

3- مقدار 6 و 2 به تابع / فرستاده شده و مقدار 3 برگردانده می شود.

تمام اپراتور ها در زبان لیسپ تابع نیستند ولی اکثر آنها تابع می باشند. فراخوانی توابع معمولا به این صورت ارزیابی می شوند ، آرگومانها از چپ به راست ارزیابی شده و مقادیر آنها از تابع عبور کرده در پایان مقادیر هر عبارت بازگردانده می شود که به این روش قانون ارزیابی در common lisp گفته می شود.

اپراتوری که از قانون ارزیابی لیسپ پیروی نمی کند quote می باشد. اپراتور quote یک اپراتور ویژه است یعنی یک قانون ارزیابی متمایزی از آن خود دارد و آن قانون : انجام هیچ چیز است. در مثال زیر اپراتور یک آرگومان گرفته و آن را کلمه به کلمه برمی گرداند.

```
> (quote (+ 3 5))
(+3 5)
```

برای راحتی لیسپ ' را به عنوان مخفف quote تعریف کرده ، شما می توانید با قرار دادی ' جلوی هر عبارت quote را فراخوانی کنید.

```
> '(+ 3 5)
(+ 3 5)
```

استفاده از علامت اختصاری بجای نوشتن عبارت quote بیشتر رایج است.

لیسپ quote را به عنوان راهی برای حفاظت عبارات از ارزیابی فراهم کرده است. در بخش های بعدی توضیح داده خواهد شد که این محافظت کردن می تواند مفید باشد.

۳-۲- داده ها (Data)

لیسپ همه نوع های داده ای که ما می توانیم در اکثر زبانها مشاهده کنیم را ارائه می دهد. یک نوع داده ای که ما با نوشتن یک سری از اعداد مانند 256 استفاده می کنیم integer است. یک نوع داده ای دیگر در لیسپ همانند دیگر زبانها وجود دارد string می باشد که یکسری از کاراکتر ها را همراه با double-quotes (دابل کوتیشن) نشان داده می شود مانند : "ora et labora".

Integer و string هر دو به مقدار خود ارزیابی می شوند. دو نوع داده ای لیسپ که ما نمی توانیم در دیگر زبانها پیدا کنیم سمبول ها (Symbols) و لیست ها (Lists) هستند. سمبول ها کلمات هستند.

> 'Artichoke

ARTICHOKE

صرف نظر از اینکه شما چگونه آنها را تایپ می کنید معمولا آنها به حروف بزرگ تبدیل می شوند.

سمبول ها معمولا به مقدار خود ارزیابی نمی شوند بنابراین اگر شما بخواهید از یک سمبول استفاده کنید باید قبل از آن از quote استفاده کنید.

لیست ها نشان دهنده ی هیچ و یا تعداد زیادی عنصر (element) در میان پرانتز ها هستند. عنصر ها می توانند از هر نوعی از جمله لیست ها باشند.

> '(my 3 "Sons")
(MY 3 "Sons")
> '(the list (a b c) has 3 elements)

(THE LIST (A B C) HAS 3 ELEMENTS)

توجه کنید که یک quote از عباراتی که بعد از آن می آید محافظت می کند.

شما می توانید با فراخوانی list لیستهایی را ایجاد کنید. از آنجایی که لیست یک تابع می باشد آرگومانهای آن ارزیابی می شوند. در اینجا ما فراخوانی + را داخل فراخوانی به یک لیست می بینیم.

> (list 'my (+ 2 1) "Sons")

(MY 3 "Sons")

اگر قبل از لیست quote آمده باشد ارزیابی کننده خود لیست را برمی گرداند ، اما اگر quote وجود نداشته باشد ازریابی کننده مقدار لیست را برمی گرداند.

> (list '(+ 2 1) (+ 2 1))

((+ 2 1) 3)

در اینجا اولین آرگومان دارای quote است و بنابراین خروجی یک لیست است. آرگومان دوم quote ندارد و بعد از فراخوانی تابع یک عدد بر گردانده می شود.

در لیست دو راه برای نشان دادن لیست های خالی وجود دارد . شما می توانید بوسیله یک جفت پرانتز که چیزی بین آنها نیست یک لیست خالی را نمایش دهید و یا شما می توانید از سمبول nil استفاده کنید. مهم نیست که شما از کدام روش برای نوشتن یک لیست خالی استفاده می کنید اما این نشان دهنده ی تهی خواهد بود.

> ( )
NIL

>nil

NIL

شما quote nil ندارید چرا که مقدار nil خود آن خواهد بود.

۴-۲- اپراتورهای لیست (List Operations)

ساخت یک لیست توسط تابع cons . اگر آرگومان دوم یک لیست باشد ، این تابع یک لیست جدید که از اضافه کردن آرگومان اول به بقیه عناصر ایجاد می کند.

> (cons 'a ' (b c d))

(A B C D)

ما می توانیم لیست هایی توسط عناصر جدید cons برروی یک لیست خالی بسازیم.

> (cons 'a (cons ‘b nil))

(A B)
> (list'a'b)
(A B)

توابع اولیه برای استخراج عناصر از داخل لیستها car و cdr هستند.
car اولین عنصر یک لیست و cdr هر چیزی که بعد از عنصر اول باشد را بر می گرداند.

```
> (car '(a b c))
A
> (cdr '(a b c))
(B C)
```

شما می توانید با ترکیب car و cdr به هریک از عناصر یک لیست دسترسی پیدا کنید. اگر شما بخواهید که عنصر سوم را بدست آورید می توان گفت:

```
> (car (cdr (cdr '(a b c d))))
C
```

با این حال شما می توانید این کار را توسط فراخوانی third بصورت ساده تر انجام دهید.

```
> (third '(a b c d))
C
```

۵-۲- درستی (Truth)

در Common Lisp سمبول t بطور پیش فرض برای نمایش درستی بکار می رود. مانند nil، t به مقدار خودش ارزیابی می شود.

اگر آرگومان تابع listp یک لیست باشد مقدار درست برگردانده می شود.

```
>(listp '(a b c ))
T
```

تابعی که مقادیر را بصورت درست یا غلط برمی گرداند predicate نامیده می شود. Predicate های زبان لیسپ اغلب دارای نامهایی که آخر آنها p است می باشند.

غلط بودن در Common Lisp بوسیله nil، لیست خالی نشان داده می شود. اگر ما به تابع listp یک آرگومانی که لیست نیست تحویل بدهیم مقدار nil برگردانده می شود.

```
> (listp 27)
 NIL
```

چرا که nil در Common Lisp دو نقش را ایفا می کند ، تابع null در صورتیکه لیست خالی باشد مقدار درست بر می گرداند.

```
> (null nil)
T
```

و تابع not مقدار درست برمی گرداند اگر آرگومان آن غلط باشد.

```
> (not nil)
T
```

یک شرط ساده در لیسپ if می باشد. این معولا سه آرگومان می گیرد: عبارت test، عبارت then و عبارت else . عبارت test ارزیابی شده در صورتی که مقدار درست برگرداند عبارت then ارزیابی می شود. در صورتیکه عبارت test مقدار false برگرداند سپس عبارت else ارزیابی شده و مقدار آن برگردانده می شود.

```
> (if (listp '(a b c))
        (+ 1 2)
        (+ 5 6))
3
> (if (listp 27)
        (+ 1 2)
         (+ 5 6))
11
```

مانند quote، if یک اپراتور ویژه است. این اپراتور نمی تواند بصورت یک تابع اجرا شود چرا که آرگومانها در فراخوانی تابع معمولا ارزیابی می شوند ، در حالی که در if تنها یکی از دو آرگومان انتهایی ارزیابی می شوند. آخرین آرگومان if بصورت انتخابی می باشد. اگر شما آنرا حذف کنید این بطور پیش فرض nil خواهد شد.

```
> (if (listp 27)
      (+ 2 3))
 NIL
```

اگرچه t بصورت پیش فرض برای نشان دادن درستی بکار می رود ، هر چیزی به جز nil به عنوان شمارش درست در چارچوب منطقی نیز بکار می رود.

```
> (if 27 1 2)
1
```

اپراتورهای منطقی and و or به شرطی ها شباهت دارند.هر دوی آنها تعدادی آرگومان می گیرند ولی فقط تعدادی از آنها یی که مقدار برگرداند ارزیابی می شوند. اگر همه ی آرگومانها درست باشند (یعنی nil نباشد) سپس and مقدار آخرین آنرا برمی گرداند.

```
> (and t(+ 1 2))
3
```

اما اگر معلوم شود یکی از آرگومانها اشتباه است هیچ یک از آرگومانها بعد از آن ارزیابی نمی شود. به همین ترتیب برای or با وجود درست بودن یکی از آرگومانها انجام می شود.

۶-۲- توابع (Functions)

شما می توانید تابع جدیدی با defun تعریف کنید. این معمولا سه و یا تعداد بیشتری آرگومان می گیرد: یک نام ، یک لیست از پارامترها و یک یا تعداد بیشتری از عبارات که بدنه ی تابع را خواهند ساخت. در اینجا توسط تابع ، بدست آوردن سومین عنصر را تعریف می کنیم:

> (defun our-third (x) (car (cdr (cdr x))))
OUR-THIRD

اولین آرگومان می گوید که اسم این تابع our-third خواهد بود. دومین آرگومان که لیست (x) است می گوید که تابع دقیقا یک آرگومان می گیرد. X یک سمبول است که به عنوان یک نگهدارنده ی مکانی که در اینجا متغیر نامیده می شود می باشد. زمانیکه متغیری برای نمایش یک آرگومان یک تابع بکار می رود مانند x آنرا پارامتر می نامیم.
دنباله ی تعریف (car (cdr (cdr x) به عنوان بدنه ی تابع شناخته شده است. این قسمت کاری که لسیت برای محاسبه ی مقدار برگشتی تابع انجام می دهد را مشخص می کند. بنابراین فراخوانی our-third برگرداندن (car (cdr (cdr x) برای هر مقدار x که به عنوان آرگومان به آن می دهیم است.

> (our-third ‘ (a b c d))
C

حالا که متغیرها را دیدیم درک سمبول ها راحت تر شده است.
به آنهایی متغیر گفته می شود که در قسمت راست خود یک شی وجود داشته باشد. به همین دلیل سمبول مانند لیست ها باید quote داشته باشند. یک لیست باید quote داشته باشند در غیر اینصورت به عنوان قسمتی از برنامه تعبیر می شود. و یک سمبول باید quote داشته باشد در غیر اینصورت به عنوان متغیر تعبیر می شود.
شما با تعریف توابع می توانید به تعمیم دادن عبارات نسخه های لیسپ فکر کنید.
در عبارات زیر بررسی می شود که آیا جمع 1 و 4 بزرگتر از 3 است:

```
>(> (+ 1 4) 3)
T
```

با جایگزین کردن این اعداد با متغیر ها ما می توانیم تابعی بنویسیم که بررسی کند جمع دو عدد بزرگتر از سومین عدد است:

```
> (defun sum-greater (x y z)
      (> (+ x y) z))
SUM-GREATER
> (sum-greater  1 4 3)
T
```

لیسپ هیچ فرقی بین یک برنامه ، یک زیربرنامه و یک تابع نمی داند . تابع ها همه چیز را انجام می دهند. اگر شما بخواهید بدانید که کدام یک از تابع ها به عنوان تابع اصلی می باشد ، شما می توانید این را انجام دهید اما بطور معمول باید هر تابع را در سطح بالا (top level) فراخوانی کنید. در میان چیزهای دیگر این بدان معنی است که شما با تکه تکه نوشتن برنامه ، برنامه خود را تست کنید.

۷-۲- بازگشتی (Recursion)

توابعی که ما در بخش قبل تعریف کردیم توابع دیگر را برای انجام تعدادی از کارهای خودشان فرخوانی می کردند. برای مثال sum-greater توابع + و > را فراخوانی می کرد. یک تابع می تواند هر تابعی را که درون خود دارد را فراخوانی کند.

یک تابعی که می تواند خود را فراخوانی کند بازگشتی نام دارد. زبان لیسپ تابع را تست می کند که آیا عنصر یک لیست می باشد یا خیر . در اینجا یک نمونه ساده از تعریف یک تابع بازگشتی وجود دارد:

```
(defun our-member (obj lst)
      (if (null lst)
            nil
            (if (eql (car lst) obj)
```

```
Lst
(our-member obj (cdr lst)))))
```

eql تساوی دو آرگومان باهم را تست می کند ، گذشته از آن همه چیزی که در این تعریف وجود دارد قبلا مشاهده شده است. در عمل به این صورت است:

```
> (our-member 'b '(a b c))
(B C)
> (our-member 'z'(a b c))
NIL
```

در تعریف our-member برار بودن حروف بررسی می شود. برای تست اینکه هر شی obj یک عضو لیست lst می باشد:

1- ابتدا بررسی می شود که lst خالی است. اگر آن خالی است آنگاه به وضوح معلوم است که obj یک عضو آن نمی باشد و هیچ کاری انجام نمی شود.

2- در غیر اینصورت اگر obj عنصر اول lst باشد. آن یک عضو است .

3- در غیر اینصورت obj تنها یک عضوی از lst است اگر آن یک عضو از بقیه ی lst باشد.

۸-۲- ورودی و خروجی (Input and Output)

تاکنون ما ورودی و خروجی را بصورت ضمنی با بهره گیری از سطح بالا انجام دادیم این روش برای برنامه های محاوره ای واقعی به احتمال زیاد کافی نمی باشد. در این بخش ما نگاهی به چند تابع ورودی و خروجی خواهیم انداخت.

format عمومی ترین تابع خروجی در زبان لیسپ می باشد. این تابع دو یا تعداد بیشتری آرگومان می گیرد: اولی نشان می دهد که خروجی کجا باید چاپ شود ، دومین آرگومان نشان دهنده ی قالب رشته می باشد و آرگومانهای باقیمانده معمولا اشیایی هستند که در جاهای مشخص شده در داخل قالب چاپ می شوند. دراینجا یک مثال آورده شده:

> (format t "~A plus ~A equals ~A~%"2 3 (+ 2 3))
2 plus 3 equals 5
NIL

توجه داشته باشید که در اینجا دو مورد نشان داده شده ، اولین خط توسط format نشان داده شده. دومین خط مقدار بازگشتی توسط فراخوانی تابع format می باشد که به طور معمول در سطح بالا نمایش داده می شود.معمولا تابعی مانند format به طور مستقیم از سطح بالا فراخوانی نمی شود ، اما داخل برنامه ها استفاده می شود.

t اولین آرگومان format نشان می دهد که خروجی درجای پیش فرض فرستاده شود. به طور معمول این در سطح بالا خواهد بود. آرگومان دوم یک رشته است که به عنوان یک قالب برای خروجی بکار می رود. در بین این رشته هر A~ نشان دهنده ی مکانی است که باید پر شود و %~ نشان دهنده ی یک خط جدید می باشد. این مکانها توسط مقادیر آرگومانهای باقی مانده به ترتیب پر خواهد شد.

read تابع استاندارد برای ورودی می باشد. زمانی که هیچ آرگومانی داده نمی شود این تابع از مکان پیش فرض می خواند ، که معمولا سطح بالا می باشد. در اینجا تابعی وجود دارد که از پرامپت ، ورودی کاربر را گرفته و هر آنچه را که گرفته بر می گرداند.

```
(defun askem (string)
      (format  t  "~A"  string)
      (read))
```

اجرای آن به صورت زیر می باشد:

> (askem "How old are you? ")
How old are you? 29
29

۹-۲- متغیرها (Variables)

یکی از اپراتورهایی که معمولا در لسیپ استفاده می شود let است ، که به شما اجازه تعریف متغیرهای محلی جدید را می دهد.

```
> (let ((x 1) (y 2))
(+ x y))
3
```

یک عبارت let دو بخش دارد. ابتدا لیستی از دستورالعمل ها برای ایجاد متغیرها برای هر نوع فرمی می آید. هر متغیر در ابتدا توسط عبارت نظیر خود مقدار دهی می شوند. همانطور که در مثال بالا ما دو متغیر x و y ایجاد کردیم که به ترتیب با 1 و 2 مقداردهی اولیه می شوند. این متغیر ها در بین بدنه let معتبر می باشند.
بعد از متغیرها و مقادیر بدنه عبارت می آید که براساس قوانین ارزیابی می شوند ، که در مورد مثال قبل تنها یکی وجود دارد و آن فراخوانی + است . مقدار آخرین عبارت به عنوان مقدار let بازگردانده می شود. در اینجا یک مثالی از چند انتخاب که شکل دیگری از تابع askem می باشد آورده شده :

```
(defun ask-number ()
        (format t "Please enter a number. ")
        (let ((val (read)))
         (if (numberp val)
val
(ask-number))))
```

این تابع متغیری به اسم val برای نگهداشتن شی ای که توسط read برگدانده می شود ایجاد می کند. بنابراین شاید شما حدس زده باشید که numberp یک predicate باشد برای تست اینکه این آرگومان یک عدد است بکار می رود. اگر مقداری که توسط کاربر وارد می شود یک عدد نباشد ask-number خود را فراخوانی می کند. مثال زیر تابعی است که تاکید به گرفتن اعداد دارد:

```
> (ask-number)
```

Please enter a number. a
Please enter a number.(ho hum)
Please enter a number. 52
52

متغیرهایی که تاکنون دیده ایم متغیرهای محلی نامیده می شود. آنها فقط در یک محدوده ی مشخص معتبر می باشند. در اینجا نوع دیگری از متغیرها وجود دارد که متغیر سراسری نامیده می شود که می تواند در هر کجایی قابل دسترسی باشد.
شما با دادن یک سمبول و یک مقدار به defparameter می توانید یک متغیر سراسری ایجاد کنید.

> (defparameter *glob* 99)
GLOB

چنین متغیری در هر جایی قابل دسترسی خواهد بود به جز در عباراتی که متغیر های محلی جدیدی با همین نام ایجاد شده باشد. برای جلوگیری از این اتفاق که بصورت تصادفی رخ می دهد ابتدا و انتهای نام متغیرهای سراسری را با علامت * نشان می دهند. نام متغیری که ما ایجاد کردیم"star-glob-star"تلفظ می شود .
شما می توانید با فراخوانی defconstant ثابت های سراسری تعریف کنید

(defconstant limit (+ *glob* 1))

در اینجا نیازی نیست که به ثابتها نام متمایز داده شود چرا که هرکسی که بخواهد از این نام برای تعریف متغیرها استفاده کند خطا ایجاد خواهد کرد. اگر شما بخواهید بررسی کنید که هر سمبول یک متغیر سراسری و یا یک ثابت است از boundp استفاده می کنیم.

> (boundp '*glob*)
T

۱۰-۲- انتساب (Assignment)

در زبان لیسپ به طور کلی از اپراتور setf برای انتساب استفاده می شود. ما می توانیم برای انتساب هر نوع متغیری از آن استفاده کنیم.

```
> (setf *glob* 98)
98
> (let ((n 10))
    (setf n 2)
     n)
2
```

زمانی که اولین آرگومان به setf داده می شود یک سمبولی که نام آن یک متغیر محلی نمی باشد به عنوان یک متغیر سراسری گرفته می شود.

```
> (setf x (list 'a 'b 'c))
(A B C)
```

این بدان معناست که شما می توانید متغیرهای سراسری را تنها بوسیله انتساب مقدار آنها بصورت ضمنی ایجاد کنید. در فایل های منبع حداقل این روش بهتراز استفاده صریح از defparameter ها می باشد.
شما می توانید بیش از انتساب یک مقدار به متغیر انجام دهید. آرگومان اول برای setf می تواند یک عبارت همانند نام یک متغیر باشد. در این مورد مقدار آرگومان دوم را بوسیله جایی که ابتدای آن مشخص شده است وارد می شود.

```
> (setf (car x) 'n)
N
>x
(N B C)
```

آرگومان اول setf جایی در هر عبارتی می تواند باشد که مکان مخصوص A را مشخص می کند.
شما می توانید به setf هر عدد (زوج) آرگومان بدهید . یک عبارت نمونه به اینصورت می باشد:

```
(setf a b
c d
e f)
```

با سه بار فراخوانی جدا setf به ترتیب برابراست:

```
(setf a b)
(setf c d)
(setf e f)
```

۱۱-۲- برنامه نویسی تابعی (Functional Programming)

برنامه نویسی تابعی به معنی نوشتن برنامه هایی که توسط بازگرداندن مقادیر به جای تغییر دادن چیزها کار می کنند. این روش متداول در لیسپ می باشد. اکثر توابع ساخته شده در لیسپ بیشتر برای بازگرداندن مقادیر آنها فراخوانی می شوند نه برای تاثیرات جانبی در آنها .
برای مثال تابع remove یک شئ و یک لیست می گیرد و لیست جدیدی که شامل هر چیزی به جز آن شی می باشد را بر می گرداند.

```
> (setf lst '(c a r a t))
(CARAT)
> (remove 'a lst)
(C RT)
```

چرا نمی توان گفت که تابع remove یک شی را از یک لیست حذف نمی کند؟ این کار را انجام نمی دهد ، چرا که لیست اصلی بعد از آن دست نخورده باقی می ماند.

```
>lst
(C A R A T)
```

پس اگر شما واقعا بخواهید چیزی را از لیست حذف کنید باید چه کار کنید ؟ در لیسپ معمولا شما با قرار دادن یک لیست به عنوان آرگومان یک تابع که آن استفاده از setf بهمراه بازگرداندن مقدار است ، این کار را انجام می دهید. برای حذف a از لیست x این صورت انجام می گیرد:

```
(setf x (remove 'a x))
```

برنامه نویسی تابعی یعنی اینکه ، واقعا از setf و چیزهایی که این را دوست دارند اجتناب کرد. در نگاه اول شاید تصور آن مشکل باشد که چگونه این ممکن است چه برسد که این روش مطلوب باشد.
چگونه می توان یک برنامه ای فقط برای برگرداندن مقدار ساخت؟
این امر ناخوشایند را می توان بدون تاثیرات جانبی به طور کامل انجام داد. یکی از مزایای برنامه نویسی تابعی این است که اجازه می دهد امتحان کردن برنامه بصورت تعاملی انجام شود.

۱۲-۲- تکرار (Iteration)

زمانی که ما می خواهیم بعضی از کارها را بصورت تکراری انجام دهیم گاهی اوقات استفاده از تکرار به جای بازگشت مؤثرتر است. یک صورت متداول برای تکرار ایجاد جدول مرتب سازی است. این تابع :

```
(defun show-squares (start end)
      (do ((i start (+ i 1)))
((> i end) 'done)
(format t "~A ~A ~%" i (* i i))))
```

چاپ مربع اعداد صحیح از شروع تا پایان :

```
> (show-squares 2 5)
2 4
3 9
4 16
5 25
DONE
```

ماکروی do اساس اپراتور تکرار در زبان لیسپ می باشد. همانند let، do متغیرهایی را ایجاد کند و آرگومان اول لیستی از تعریف متغیرهاست. هر عنصر این لیست می توان به این شکل باشد:

(variable initial update)

که در آن variable یک سمبول ، initial و update عبارات می باشند. در ابتدا variable با مقدار آغازین مربوط به خود مقدار دهی می شود ، در هر تکرار مقدار این متغیر با مقدار update مقدار دهی خواهد شد. do در show-squares یک متغیر ایجاد می کند . در ابتدای تکرار i با مقدار start مقدار دهی می شود و در هر تکرار مقدار آن یک واحد اضافه خواهد می شود.

دومین آرگومان do باید یک لیست که شامل یک یا تعداد بیشتری عبارت است باشد. عبارت اول جایی که تکرار باید متوقف شود را تست می کند ، در مورد بالا عبارت تست (end i >) است. عبارت باقی مانده در این لیست بر اساس قواعد ارزیابی می کند ، زمانی که تکرار متوقف شده آخرین مقداری که بازگردانده شده به عنوان مقدار بازگردانده شده از do خواهد بود.

آرگومان های باقی مانده بدنه ی حلقه را تشکیل می دهند . آنها در هر تکرار طبق قوانین ارزیابی می شوند. در هر تکرار متغیر ها مقدار جدید می گیرند سپس شرط پایان ارزیابی می شود و اگر شرط برقرار نباشد بدنه ارزیابی می شود.

برای مقایسه در اینجا نمونه بازگشتی show-squares آورده شده است:

```
(defun show-squares (i end)
      (if (> i end)
            'done
            (progn
```

```
(format t  " ~A ~A~%" i (* i i))
(show-squares (+ i 1) end))))
```

تنها چیزجدیدی که در این تابع وجود دارد (progn) است ، آن تعدادی از عبارات را می گیرد و آنها را براساس قوانین ارزیابی می کند و آخرین مقدار را بر می گرداند.

زبان لسیپ اپراتور تکرار ساده ای برای استفاده در موارد خاص دارد . برای تکرار از طریق عناصر یک لسیت ، آنچه شما بیشتر می خواهید استفاده از dolist است. در اینجا تابعی که طول یک لیست را برمی گرداند آورده شده است :

```
(defun our-length ( lst)
      (let ((len 0))
            (dolist (obj lst)
                  (setf len (+ len 1)))
        len))
```

در اینجا dolist یک آرگومان به شکل (variable expression) و بدنبال آن بدنه ی عبارت دارد. بدنه ی تکرار توسط ملزم کردن متغیر ها در هر تکرار با باز گرداندن عناصرلیست توسط عبارات ارزیابی خواهد شد. بنابراین در حلقه ی بالا برای هر obj در lst یک واحد به len اضافه خواهد کرد.

شکل بازگشتی این تابع به صورت زیر می باشد:

```
(defun our-length ( lst)
      ( if (null lst)
      0
(+ (our-length (cdr lst)) 1)))
```

اگر لیست خالی باشد طول آن صفر خواهد بود. در غیر اینصورت طول cdr به اضافه 1 خواهد بود. این شکل از تابع our-length تمیز تر است اما کارآمد تر نیست.

۱۳-۲- توابع به عنوان اشیاء (Functions as Objects)

در لیسپ توابع همانند سمبول ها یا رشته ها یا لیستها ، اشیای با قاعده ای هستند . اگر ما نام یک تابع را به function دهیم ، آن یک شی مجتمع شده را برمی گرداند.
function مانند quote یک اپراتور خاص می باشد بنابراین ما برای آرگومان آن علامت quote نداریم:

```
> (function +)
#<Compiled-Function + 17BA4E>
```

مقدار بازگشتی که از طریق function در پیاده سازی های متداول لیسپ نمایش داده می شود ممکن است عجیب به نظر برسد.
تاکنون که ما با اشیاء سروکار داشتیم بطور قراردادی زمانی لیسپ آنها را نمایش می دهد که ما آنها را تایپ کنیم. این قرارداد برای توابع صدق نمی کند. ذاتا ترکیب یک تابعی مانند + همانند یک تکه کد به زبان ماشین می باشد. همانطور که می توان از ' برای کوته سازی quote بکار برد ما می توانیم از #' برای کوته سازی function استفاده کنیم:

```
> #' +
#<Compiled-Function + 17BA4E>
```

این کوته سازی با sharp-quote شناخته می شود.
مانند هر نوع دیگری از اشیاء ما می توانیم function را به عنوان آرگومان ارسال کنیم. یک تابعی که تابع را به عنوان آرگومان می گیرد apply نام دارد. آن یک تابع و لیستی از آرگومانهای آن را می گیرد و نتیجه ی اجرای تابع بر روی آرگومانها را بر می گرداند:

```
> (apply #'+ '(1 2 3))
6
> (+ 1 2 3)
6
```

این می تواند هر تعداد آرگومان که آخرین آن یک لیست باشد را بگیرد:

```
> (apply #'+ 1 2 '(3 4 5))
15
```

تابع funcall نیز به همین صورت است اما نیازی به بسته بندی آرگومانها داخل یک لسیت نیست:

```
> (funcall  #'+ 1 2 3)
6
```

LAMBDA چیست؟

lambda در یک عبارت لامبدا ، یک اپراتور نمی باشد ، بلکه تنها یک سمبول است.
در نسخه های ابتدایی لیسپ هدفی که آن داشت : توابع داخلی به عنوان لیست حضور داشتند و تنها راهی که نشان می داد یک تابع از یک لیست معمولی می باشد بررسی اینکه عنصر اول lambda باشد.
در لسیپ شما می توانید توابع را به عنوان لیستها بیان کنید ، اما آنها نمایندگان داخلی به عنوان اشیای تابع مجزا می باشند. بنابراین lambda ضروری می باشد ، در اینجا هیچ تناقضی که به عنوان توابع مشخص شوند وجود نخواهد داشت.

```
((x) (+ x 100))
```

به جای

```
(lambda (x) (+ x 100))
```

اما برنامه نویسان لیسپ استفاده از توابع را با سمبول lambda شروع کردند. به همین دلیل common lisp این رسم را حفظ کرده . ماکروی defun یک تابع ایجاد کرده و نامی را به آن اختصاص می دهد. اما نیازی نیست که تابع ، نام داشته باشد و ما نیازی به defun برای تعریف آنها نداریم. همانند گونه های دیگر اشیاء در لیسپ ما می توانیم به صورت متناظر به توابع اشاره کنیم.

برای اشاره به یک عدد صحیح ما از یک سری عدد استفاده می کنیم ، برای اشاره به یک تابع ما آنچه که عبارت لامبدا نامیده می شود استفاده می کنیم. یک عبارت لامبدا شامل یک سمبول lambda و دنبال آن لیستی از پارامترها و بعد از آن بدنه که شامل صفر و یا تعداد زیادی عبارت است می آید. در اینجا یک عبارت لامبدا وجود دارد که نشان دهنده ی یک تابعی است که دو عدد را گرفته و حاصل جمع آنها را برمی گرداند :

```
(lambda (x y)
(+ x y))
```

(x y) پارامترهای لیست بوده و بعد از آن بدنه ی تابع آمده است.
یک عبارت لامبدا می تواند به عنوان نام یک تابع در نظر گرفته شود. همانند نام تابع یک عبارت لامبدا می تواند عنصر ابتدایی فراخوانی تابع باشد.

```
> ((lambda (x) (+ x 100)) 1)
101
```

و با اضافه کردن sharp-quote به یک عبارت لامبدا ، ما تابع متناظر آن را بدست می آوریم.

```
> (funcall #'(lambda (x) (+ x 100))
            1)
101
```

در میان چیزهای دیگر ، این روش نشانه گذاری به ما اجازه می دهد که بدون نامگذاری توابع از آنها استفاده کنیم.

۱۴-۲- نوع ها (Types)

لیسپ دارای یک رویکرد انعطاف پذیر غیر متداول به نوع ها دارد. در بسیاری از زبانها متغیر ها نوع های متفاوتی دارند و شما نمی توانید از یک متغیر بدون تعریف نوع آن استفاده کنید.

در common lisp مقادیر نوع هایی دارند که متغیر نیستند. شما می توانید تصور کنید که هر شی که یک برچسب به آن متصل شده ، تعریف نوع آن می باشد. این رویکرد تعیین نوع به صورت آشکار می باشد. لازم نیست شما نوع متغیر ها را اعلام کنید زیرا هر متغیر می تواند اشیاء از هر نوعی را نگه دارد.

اگرچه اعلام نوع ضروری نمی باشد ، شما شاید به دلیل کارایی ایجاد می کنید. نوع ها در ترکیب common lisp دارای سلسله مراتب subtypes و supertypes هستند. یک شی معمولا بیشتر از یک نوع دارد . به عنوان مثال عدد 27 دارای نوع های fixnum، integer ، rational ، real ، number ، atom و t می باشد. نوع t یک supertype برای همه نوع ها می باشد. بنابراین همه چیز از نوع t می باشد.

تابع typep یک شی و یک نوع مشخص گرفته و اگر شی از این نوع باشد مقدار درست برمی گرداند.

```
> (typep 27 'integer)
T
```

منابع

Paul Graham.1996. ANSI Common Lisp