بسم الله الرحمن الرحیم

آشنایی با Lisp

زبان lisp در در اواخر دهه ۵۰ توسط John McCarthy در MIT به وجود آمد. بنابراین زبان از قدیمی ترین زبانهای برنامه سازی (سطح بالا) است. هدف عمده طراحی آن انجام محاسبات نمادین ((symbolic computation بود. چون بیشتر محاسبات کامپیوتری در آن زمان به صورت عددی بود و این برای برخی از شاخه های علوم کامپیوتر (مانند هوش مصنوعی) کافی نبود.

باید اعتراف کنیم که زبان lisp زبان بسیار جالبی است. یعنی کسی که با lisp و زبانهای نظیر آن آشنایی ندارد، در این زبان ویژگی هایی را خواهد یافت که در زبانهایی که تا کنون با آنها آشنا شده، چنین ویژگیهایی را ندیده است. یکی از بارز ترین این ویژگیها یکنواختی فوق العاده ای است که در آن به چشم می خورد و موجب می شود این زبان بسیار انعطاف پذیر باشد. با مطالعه مطالبی که در ادامه خواهد آمد به این مطلب پی می برید.

هر زبان برنامه سازی از ترکیب تعدادی واحد اولیه تشکیل شده. مثلا زبانی مانند C از عبارات (جمع، ضرب، انتساب و...)، دستورات کنترلی مثل شرط ها و حلقه ها، بلاک ها، توابع و... تشکیل شده. همچنین دارای تعدادی داده(data) است. مثل اعداد صحیح، اعداد اعشاری، کاراکترها، رشته ها، structure ها و.... ترکیب تمام این اجزا یک برنامه C را به وجود می آورند.

شاید اولین ویژگی که در زبانlisp به چشم می خورد این است که تمام اجزای سازنده آن از یک نوع می باشند. به این اجزا، Symbolic Expression یا به اختصار s-expression می گویند. یعنی تمام برنامه های lisp چه دستورات آن و چه داده ها) از نوع s-expression هستند. نتیجه جالبی که می شود از این موضوع گرفت این است که یک داده در lisp می تواند خودش یک برنامه lispباشد. فرض کنید در زبان C یک متغیر یا structure از نوع Program بتوانیم تعریف کنیم که خود آن متغیر یک برنامه C باشد. بعد به آن چند تا دستور if و for هم به آن اضافه کنیم و بعد با یک تابع آن را اجرا کنیم و همه این کارها هم در زمان اجرای برنامه (runtime) صورت بگیرد. درlisp چنین کاری را می توان انجام داد.

حالا ببینیم s-expression چیست. می شود گفت s-expression به دو صورت می تواند باشد. یا یک atom است و یا یک list. پس بهتر است ابتدا اتم را تعریف کنیم:

کلا اتم به معنای چیزی است که به اجزای کوچکتری تجزیه نمی شود. یک اتم در زبان lisp می تواند یک نماد یا یک عدد باشد. هر اتم می تواند رشته ای از کاراکترها باشد. این کاراکترها شامل ارقام 0-9 و حروف a-z, A-Z و همچنین کاراکترهای+-/*@$%^&_~?<> =- . می باشند. در زیر نمونه هایی از اتم ها را مشاهده می کنید:

```
abc
+
100
12.2
nil
hello-world
vector+
```

حال می توانیم یک list را تعریف کنیم.

لیست ها در لیسپ دنباله ای از s-expression ها هستند. یعنی اعضای یک list می تواند هم یک اتم باشد هم یک لیست دیگر. در مثال زیر نمونه ای از لیست ها را مشاهده می کنید:

```
(1  2  3)
(I  am  a  human)
(a  b  c  (2  3)  (3  4))
( ( (a  b (c  d) )  (a) ) () )
()
```

مورد آخر در مثال بالا یک list خالی است که آن را با نماد nil هم نشان می دهند. (یعنیnil هم اتم است و هم لیست).
داده ها در lisp به همان صورت های بالا (لیست و اتم) هستند. حالا ببینم برنامه های lisp به چه صورتی هستند. همانطور که گفته شد برنامه های lisp هم به صورت s-expression می باشند. با یک مثال شروع میکنیم. مثلا عبارت زیر را در نظر بگیرید:

```
(+ 1 2)
```

این عبارت یک s-expression و از نوع لیست است. یعنی یک لیست سه عضوی. اگر به آن به عنوان یک عبارت محاسباتی نگاه کنیم معنای آن همان 1+2 است. شاید در درس مبانی برنامه سازی دیده باشید که عبارات را به چند صورت می شود نشان داد از جمله به صورت های infix، postfix و prefix. در lisp عبارات به صورت prefix می باشند. یعنی اول عملگر می آید و بعد از آن عملوندها نوشته می شوند. اگر عبارت بالا را به interpreter لیسپ[1] بدهیم. عملگر + را روی دو عملوند آن اعمال (apply)می کند و نتیجه را گزارش می دهد:

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

حال عبارت (7+3)*(7-3) را حساب می کنیم:

```
>>> (* (+ 7 3) (- 7 3))
40
```

تمام برنامه های lispبه همین صورت است. یعنی یک عبارت به interpreter می دهیم و عبارت محاسبه می شود. به همین سادگی! ممکن است این سوال پیش بیاید که lisp عبارات را مثل تمام زبانهای برنامه سازی دیگر محاسبه می کند. اما زبانهای دیگر کارهای دیگری هم غیر از محاسبه عبارات دارند. مثلا کنترل کردن روند برنامه بوسیله شرط ها یا حلقه ها و یا تعریف توابع. آیا لسیپ این کارها را هم انجام می دهد؟ یا اینکه به قدرتمندی زبانهای دیگر برنامه سازی نیست؟
باید پاسخ دهیم که lisp تمام این کارها را انجام می دهد ولی در قالب محاسبه عبارات. یک مثال ساده می زنیم:
یک ویژگی که زبان C نسبت به برخی زبانهای دیگر مثل Pascal دارد آن است که انتساب (assignment) در آن یک عبارت (expression) است نه یک دستور (statement). مثلا عبارت:

```
a * (c = d) + 1
```

در C شامل یک انتساب است. وقتی که این عبارت محاسبه می شود، انتساب هم انجام می شود. حال فرض کنید در زبان C، ساختارهای if، for، switch، while و تعریف توابع هم یک عبارت بودند. آنوقت این زبان زبانی فوق العاده یکنواخت و همچنین انعطاف پذیر می شد. زبان lisp اینچنین است.
وقتی که یک s-expression به lispبه عنوان برنامه داده می شود تا آن را اجرا کند. lisp تنها کاری که می کند این است که آن را محاسبه می کند. اجازه بدهید به جای محاسبه از کلمه مقدار یابی استفاده کنیم که رسا تر است. عبارت ها در lisp طبق قانون زیر مقدار یابی می شوند:

1. اگر عبارت (s-expression) یک اتم باشد:
   a.اگر عبارت یک عدد باشد حاصل مقداریابی خود همان عدد است.
   b.اگر عبارت یک symbol باشد حاصل evaluation مقداری است که به آن نسبت داده شده.
   در lisp به هر symbol می تواند یک مقدار نسبت داده شود. مانند زبانهای برنامه سازی دیگر که در آنها به هر متغیر یک مقدار نسبت داده می شود. مقدار هر symbol می تواند یک عدد، یک symbolدیگر، یک لیست یا حتی یک تابع (function) باشد.

2- اگر عبارت یک لیست باشد، اولین عضو آن را به عنوان یک تابع در نظر می گیرد و سعی می کند آن را روی بقیه اعضای list اعمال (apply) کند. البته معمولا قبل از اینکه اینکار را انجام دهد تمام اعضای دیگر لیست را مقدار یابی می کند. (به همین روشهایی که گفته شد) پس

[1] برنامه ای که عبارات نوشته شده در زبان لیسپ را مقداریابی و اجرا می کند.

عمل evaluation یک عمل بازگشتی(recursive) است.

بنابر این اگر به interpreter لیستی به صورت (f x y z) بدهیم مثل فراخوانی تابع f روی x, y, z است که در زبانهای دیگر به صورت "f(x, y, z)" نمایش داده می شود. پس می توان گفت در lisp توابع و عملگرها، دو چیز متفاوت نیستند. مثلا در عبارت (+ 1 2 3) می توان گفت تابع '+' روی 1,2,3 اعمال شده. گفتیم که ممکن است به برخی از symbol ها مقداری انتساب داده شود. در اینجا هم + خودش یک symbol معمولی است که به آن یک تابع انتساب داده شده.

چند مثال می زنیم:

```
>>> 1.2
1.2
```

در این مثال یک atom داریم که عدد است. بنابراین حاصل آن خودش می شود.

```
>>> (+ 1 2 3 a)
10
```

در مثال بالا تابع '+' روی آرگومانهای خود اعمال شده و عدد 10 به عنوان حاصل برگردانده شده. اما قبل از آن تمام آرگومانهای آن مقدار یابی می شوند. حدث میزنید که مقدار نماد a چقدر است؟

```
>>> (1 2 3 4)
Error
```

در مثال بالا interpreter لیسپ سعی میکند تابع '1' را روی سه آرگومان دیگر اعمال کند ولی چنین تابعی وجود ندارد. پس یک پیغام Error می دهد.

(توجه کنید که این بدین معنا نیست که داده ای به صورت (1 2 3 4) نمی توانیم داشته باشیم)

```
>>> (* (+ 1 2) (- 3 4) 5)
-15
```

در این مثال قبل از اعمال تابع * باید آرگومانهای آن مقدار یابی شوند. پس اول حاصل (+ 1 2) محاسبه می شود و بعد حاصل (- 3 4) و بعد 5 (که حاصل آن برابر خودش است) و سپس تابع * روی آرگومانهای 3, -1, 5 اعمال می شود که حاصل -15 است.

طرح مساله

حال فرض کنیم تابعی به جای آنکه آرگومانهایی که لازم دارد عدد باشد روی آرگومانهایی از نوع دیگر کار کند. مثلا فرض کنیم تابعی به نام print داریم که یک symbol را می گیرد و آن را چاپ می کند. ما می خواهیم با استفاده از این تابع یک salam در اول برنامه خود چاپ کنیم.

اگر آن را به این صورت استفاده کنیم:

```
>>> (print salam)
```

به نظر شما چه اتفاقی می افتد ؟

قبل از آنکه تابع print روی آرگومانش اجرا شود، salam باید مقدار یابی شود. پس تابع print روی مقدار منتسب به salam اعمال می شود نه خود salam. حال اگر مقدار salam برای print معتبر نباشد یا اصلا مقداری به salam منتسب نشده باشد برنامه با خطا مواجه می شود.

یک راه حل آن است که (به نحوی) نماد salam را به یک نماد دیگر مثل p انتساب دهیم و بعد p را به عنوان آرگومان به print بدهیم.

```
>>> (print p)
```

اما آیا راه حل ساده تری هم وجود دارند ؟

یک مساله دیگر:

فرض کنید تابعی به نام length ساخته ایم که یک list را به عنوان آرگومان می گیرد و طول list را برمی گرداند. می خواهیم آن را روی لیست (**1 2 3 4**) امتحان کنیم اگر آن را به این صورت به کار ببریم:

```
>>>  (length (1 2 3 4))
```

نتیجه چه خواهد شد ؟

قبل از آنکه تابع روی (1 2 3 4) اجرا شود، این list باید مقدار یابی شود. پس lisp سعی میکند تابع 1 را روی 2,3,4 اعمال کند که چون چنین تابعی نداریم با خطا مواجه می شویم.

یک راه حل آن است که تابعی داشته باشیم که تعدادی آرگومان بگیرد و به عنوان مقدار لیستی از آرگومانهای خود برگرداند. فرض کنید اسم این تابع list است:

```
>>> (list 1 2 3 4)
(1 2 3 4)
```

آنگاه تابع length را می توان به این صورت به کار برد:

```
>>> (length (list 1 2 3 4))
4
```

اما آیا در اینگونه موارد راه حل ساده تری هم وجود دارد ؟

راه حل ساده تر:

مثالهایی که در بالا موجب خطا شد را در نظر بگیرید:

```
>>> (print salam)
>>> (length (1 2 3 4))
```

در هر دو مورد اگر می شد کاری کرد که lisp قبل از اعمال تابع روی آرگومان خود، آن را مقدار یابی نکند، به منظور خود می رسیدیم. این کار در lisp با قرار دادن یک ' (single quote) در ابتدای عبارات انجام می شود و به interpreter می گوید که این عبارت را مقدار یابی نکند. (در نتیجه اگر عبارت یک list باشد وارد آن نمی شود و اعضای آن را هم مقدار یابی نمی کند.)

برای روشن تر شدن مطلب به مثال زیر توجه کنید:

```
>>> (+ 1 2)
3
>>> '(+ 1 2)
(+ 1 2)
```

در مثال اول (+ 1 2) محاسبه شده و حاصل کل عبارت برابر 3 می شود ولی در مثال دوم علامت ' مانع از مقدار یابی شدن عبارت می گردد و حاصل لیست سه عضوی (+ 1 2) می شود.

پس در مورد مسایلی که مطرح شد کافی است قبل از آرگومانها یک ' قرار دهیم:

```
>>> (print 'salam)
salam
>>> (length '(1 2 3 4))
4
```

طرح مساله

تابع list را که در بالا مطرح کردیم به یاد دارید. بد نیست بدانید خود lisp چنین تابعی دارد. این تابع تعداد دلخواهی آرگومان می گیرد و به عنوان مقدار برگشتی، لیستی از آرگومانهای خود را بر می گرداند.

شاید به نظر برسد با وجود عملگر ' که در بالا مطرح شد، به این تابع نیازی نباشد. مثالهای زیر را در نظر بگیرید:

```
>>> (list 1 2 3 4)
(1 2 3 4)
```

```
>>> '(1 2 3 4)
(1 2 3 4)
```

می بینید که نتیجه هر دو یکی است

آیا این حرف درست است که با وجود (single quote) ' نیازی به تابع list نیست؟

جواب آن را به خودتان وا می گذاریم.

کار با لیست ها:

نام lisp از کلمات LISt Processing گرفته شده. از همینجا می توان به اهمیت کار روی لیست ها در lisp پی برد. در lisp تعدادی تابع اولیه وجود دارد که کار روی لیست ها را انجام می دهند. (در اینجا منظور ما از تابع اولیه این است که این توابع را نمی توانیم خودمان بوسیله دستورات lisp بسازیم، بلکه بوسیله سخت افزار و زبان اسمبلی هر کامپیوتر پیاده سازی می شوند). برای فهمیدن مفهوم دقیق این توابع، لازم است با طرز پیاده سازی این زبان بیشتر آشنا شویم و برای مفاهیمی مثل s_expression و list تعریف دقیقتری ارائه دهیم. اما چون در این مقاله نمی خواهیم وارد جزئیات شویم، به یک تعریف سطحی از این توابع بسنده می کنیم. (به همین علت ممکن است مثلا نام برخی توابع نامربوط به نظر برسد.)

CAR: این تابع یک آرگومان از نوع لیست می گیرد و اولین عضو آن را بر می گرداند:

```
>>>  (car '(1 2 3 4))
1
>>>  (car '(+ 2 3))
+
>>>  (car '( (1 2)  2  (3) ) )
(1 2)
>>>  (car '())
ERROR
```

توجه کنید که همانطور که گفته شد، قرار دادن (single quote) '، قبل از یک عبارت باعث می شود آن عبارت محاسبه و مقداریابی نشود.

CDR: این تابع یک آرگومان از نوع لیست می گیرد و یک لیست شامل اعضای دوم به بعد لیست بر می گرداند.

```
>>> (cdr '(1 2 3 4))
(2 3 4)
>>> (cdr '(+ 2 3))
(2 3)
>>> (cdr '( (1 2)  2  (3) ) )
(2 (3))
>>> (cdr '())
ERROR
```

توجه کنید که نامهای CAR و CDR مربوط به ساختار کامپیوتری است که LISP اولین بار روی آن پیاده سازی شد. ممکن است در برخی از نسخه های لیسپ به جای آنها نامهای FIRST و REST ویا HEAD و TAIL را به کار ببرند.

CONS: این تابع یک s-expression و یک list را به عنوان آرگومان می گیرد و یک LIST برمی گرداند که عضو اول آن همان s_expression (آرگومان اول) و بقیه اعضای آن همان اعضای لیست (آرگومان دوم) است. به عبارت دیگر عبارت (CONS X L) یک لیست است که CAR آن X و CDR آن L است.

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

```
>>> (cons  +  '(2 3))
(+ 2 3)
>>> (cons   '(1 2)   '(2  (3))   )
((1 2)  2  (3))
```

به مثالهای زیر توجه کنید:

```
>>> (car (cdr '(1 2 3 4)))
2
>>> (cons 'a  '())
(a)
>>> (cons 'a  'nil)
(a)
>>> (cons 'a  nil)
(a)
```

توجه کنید که نماد nil همان لیست تهی است. در مثال آخر، چون nil یک symbol است و جلو آن quote هم گذاشته نشده، پس باید اول مقدار آن محاسبه شود و تابع cons روی مقدار منتسب به آن اعمال شود. ولی می بینیم که حاصل این مثال با مثال بالایی یکی است. دلیل آن این است که، در lisp، مقدار منتسب به نماد nil، خود نماد nil است. به اصطلاح حاصل مقداریابی nil خودش است.(nil evaluates to itself)

از عملکرد توابع car، cdr و cons می توان حدس زد که پیاده سازی لیست ها در LISP به صورت لیست پیوندی (linked list) است. لیست پیوندی از تعدادی جزء مانند هم تشکیل شده، که هر کدام از این اجزاء شامل یک قسمت داده و یک قسمت اشاره گر است که به جزء بعدی اشاره می کند. شاید در مقالات بعدی که ساختارهای زبان lisp را دقیقتر بررسی بکنیم، به این بحث بیشتر بپردازیم.

داده ها و رکورد ها را در lisp می توان به صورت لیست نگهداری کرد. فرض کنید می خواهیم تعدادی رکورد را که هر کدام شامل شماره دانشجویی، نام و نام خانوادگی دانشجویان است، در یک لیست نگه داری کنیم. آن لیست چیزی شبیه این می شود:

```
((800111 Ali Mohseni)
 (800112   Reza  Nazari)
 (800113    Maryam   Karimi))
```

حال فرض می کنیم این لیست به یک symbol به نام L منتسب شود.

حال اگر بخواهیم نام خانوادگی مربوط به رکورد دوم را بدست آوریم باید بنویسیم:

```
>>> (car (cdr (cdr (car (cdr L) ) ) ) )
Nazari
```

برای نوشتن عبارت فوق یک روش ساده تر هم وجود دارد:

```
>>> (caddadr L)
Nazari
```

می توانید ساختاری که در مثال بالا آمده است را مانند یک Macro در نظر بگیرید که قبل از تفسیر به همان فرم car (cdr (cdr (car (cdr L))))) تبدیل می شود. (مانند عمل preprocessing در زبان C)

مقادیر منطقی و predicate ها:

قبل از آنکه با ساختارهای کنترلی lisp آشنا شویم، باید حتما بدانیم که مقادیر منطقی در lisp چگونه نشان داده می شوند. گفته شد که نماد nil در lisp یک نماد خاص است. از این نماد برای نشان دادن یک لیست خالی استفاده می شود. مورد استفاده دیگر nil، نشان دادن مقدار منطقی نادرست (FALSE) است. همانطور که گفتیم مقدار منتسب به nil، خود nil است. بنابر این لازم نیست در عبارات قبل از آن quote قرار دهیم.

برای نشان دادن مقدار صحیح در lisp معمولا از نماد T استفاده می کنیم. در حقیقت برای lisp هر عبارتی غیر از nil، مقدار منطقی درست دارد. اما T یک نماد است که همیشه می توانیم مطمئن باشیم که مقدار آن صحیح است. چون (مانند نماد nil) مقدار منتسب به T نیز خود نماد T است. (T evaluates to T)

برای مثال تابع ATOM در lisp را در نظر بگیرید. این تابع جزء توابع پایه lisp است و عملکرد آن بدین صورت است که یک آرگومان می گیرد و مقداری منطقی را برمی گرداند که نشان می دهد آیا آرگومان یک اتم است یا نه.

```
>>> (atom   'x)
T
>>> (atom   '(1 2 3))
nil
>>> (atom   nil)
T
```

در lisp به توابعی که مقدار برگشتی آنها یک مقدار منطقی (درست یا نادرست) است، Predicate می گوییم. مثلا تابعatom که در بالا ذکر شد، یک predicate است. گفته شد که lisp هر چیزی غیر از nil راtrue در نظر می گیرد. پس مفهوم predicate کاملا وابسته به منظوری است که از تابع برداشت می شود.

یکی دیگر از توابع اساسی lisp، تابع EQ است. این تابع دو آرگومان می گیرد و می گوید آیا این دو آرگومان، مربوط به یک نقطه از حافظه اند یا نه. در مورد symbol ها، lisp طوری پیاده سازی شده که اگر دو symbol مساوی باشند، در یک آدرس حافظه قرار دارند. lisp این symbol ها را در جایی به نام *atom table* نگه می دارد[2]. بنابر این اگر لازم باشد یک symbol به atom table اضافه شود، فقط در صورتی اضافه می شود که در atom table وجود نداشته باشد. نتیجه اینکه در مورد اتم ها از این تابع می توان به عنوان تابع تساوی استفاده کرد:

```
>>> (eq   'x   'x)
T
>>> (eq   'x   'y)
nil
>>> (eq   'x   y)
T
```

در مثال آخر، می توانید حدث بزنید که مقدار منتسب به y همان x است.

```
>>> (eq   '(1 2)   '(1 2))
nil
```

در مثال بالا دو لیست با هم مساوی هستند، ولی چون آدرس آنها در حافظه یکسان نیست، حاصل تابع (EQ، nil false) شده است.

دستورات کنترلی:

گفته شد که جریان کنترلی (control flow) در lisp، بوسیله محاسبه (evaluation) عبارات صورت می پذیرد. محاسبه در lisp هم به صورت احضار توابع و اجرای هر تابع روی آرگومانهای خودش است. برای اینکه بهتر متوجه شویم که چگونه دستورات کنترلی (مثل شرط یا حلقه) می توانند بوسیله محاسبه عبارات پیاده سازی شوند، یک مثال از زبان C می آوریم.

جالب است بدانید که اگر در زبان C دستورات شرطی (if, switch) و حلقه (while, do, for) وجود نداشت، چیزی از توانایی های آن کم نمی شد. (البته اگر محدودیت ها و هزینه های زمانی و سخت افزاری را در نظر نگیریم). زیرا دستورات شرطی را می توان با عملگر? : و حلقه ها را هم می توان بوسیله توابع بازگشتی پیاده سازی کرد. در این صورت تمام اعمال این به صورت محاسبه عبارات می شد. (گفته شد که انتساب در C یک عبارت است)

مثلا برنامه زیر را در C در نظر بگیرید:

```
int fact(int x) {
```

[2] این مفهوم نظیر مفهوم جدول نمادها (symbol table) یا جدول متغیر ها در برخی زبانهای برنامه سازی است.

```
        int ans = 1;
        while (x > 1) {
             ans = ans * x;
             x = x - 1;
        }
        return ans;
}
```

برنامه فوق را می توان به این صورت هم نوشت:

```
int fact(int x) {
        return x <= 1 ? 1 : x * fact(x-1);
}
```

دستور شرط در lisp:

عملیات شرطی در lisp، بوسیله تابع cond انجام می شود. آرگومانهای این تابع شامل تعدادی لیست دو عضوی است:

```
(COND  (condition1  exp1)  (condition2  exp2)...  (conditionN  expN))
```

عضو اول هر کدام از این لیست ها، یک عبارت منطقی است که حاصل آن می تواند درست یا نادرست باشد. عضو دوم هر لیست یک عبارت است که حاصل آن ممکن است به عنوان حاصل کل تابع cond برگردد. lisp، لسیت های دوتایی را به ترتیب امتحان می کند. یعنی عضو اول آنها را محاسبه می کند. اگر حاصل عضو اول مقداری صحیح بود، حاصل عضو دوم را به عنوان جواب کل تابع بر می گرداند. وگرنه به سراغ عضو بعدی می رود و همین عمل را تکرار می کند. توجه کنید که طرز کار تابع cond با همه توابعی که تا کنون دیده اید فرق می کند. این تابع، قبل از اجرا شدن آرگومانهای خود را محاسبه و مقداریابی نمی کند. حتی وقتی عضو اول یک آرگومان صحیح بود، دیگر به اعضای بعدی کاری ندارد.

برای مثال فرض کنید مقدار منتسب به x، c باشد:

```
>>>  (cond  ((eq  x 'a) 'aa)
             ((eq  x 'b) 'bb)
             ((eq  x 'c) 'cc)
             ( T         'nn)
             )
        )
cc
```

مشاهده می شود که در مثال بالا چون حاصل عبارت (eq x 'c) صحیح بوده، مقدار 'cc به عنوان حاصل فراخوانی تابع cond برگردانده شده.

در برنامه های lisp، در خیلی از دستورات شرطی، عضو آخر تابع cond به صورت (T exp) است. (به همان مثال بالا توجه کنید)، با توجه به اینکه مقدار T همیشه صحیح است، به نظر شما این عبارت در آخر شرط چه کاری انجام می دهد؟

توابع AND و OR:

این توابع تعداد دلخواهی آرگومان می گیرند و and یا or آنها را برمی گردانند. ولی تفاوت آنها با توابع عادی در آن است که ابتدا همه آرگومانهای خود را مقداریابی نمی کنند (مثل cond) بلکه آنها را به ترتیب از راست به چپ محاسبه می کنند تا جایی که حاصل and یا or معلوم شود. و بعد از آن دیگر آرگومانهای بعدی را محاسبه نمی کنند.

تعریف توابع در lisp:

اگر به مطالبی که تا بحال در باره لیسپ گفته شد، توجه کنید، خواهید دید که با آنها حتی برنامه های ساده هم نمی شود نوشت. چون حداقل به دستواتی مانند حلقه و انتساب نیاز دارید. گفته شد که به برخی از symbol ها ممکن است مقداری منتسب شده باشد. ولی نگفتیم که این انتساب چگونه صورت می گیرد. همچنین باید بگوییم که چگونه می توانیم تکرار (iteration) را در lisp پیاده سازی کنیم. شاید پاسخ تمام این

پرسش ها را در تعریف توابع لیسپ بیابیم.

باید بگوییم که یک برنامه لیسپ با تعریف توابع ساخته می شود. یعنی اصلی ترین قسمت یک برنامه lisp، تعریف توابع است. در لیسپ حلقه وجود ندارد (حداقل در نسخه های اول لیسپ)، اگر در برنامه به تکرار نیاز باشد، می توان آن را با توابع بازگشتی (recursive) پیاده سازی کرد. در یک تابع می توان دنباله ای از دستورات را نوشت. (توجه کنید که تمام دستوراتی که تا به حال دیده ایم به صورت فراخوانی توابع بوده). درون یک تابع حتی می شود یک تابع دیگر تعریف کرد. تا بحال از توابع زیادی استفاده کرده ایم و طرز فراخوانی توابع را یاد گرفته ایم. حال می خواهیم طریقه تعریف یک تابع را هم یاد بگیریم.

ابتدا باید دستور lambda را تعریف کنیم. این دستور به عنوان مقدار *یک تابع* برمیگرداند. فرض کنید تابعی داریم که دو عدد را می گیرد و مجموع آنها را برمی گرداند. به دستور زیر توجه کنید.

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

آرگومان دوم lambda لیستی از آرگومانهای تابع مورد نظر است و آرگومان بعدی هم مقدار بازگشتی تابع را با توجه به آرگومانها نشان می دهد. باز هم تکرار می کنیم که lambda، تعریف تابع نیست، بلکه حاصل آن یک تابع است. به مثال زیر توجه کنید.

```
>>> ((lambda (x  y)  (+  x  y) )    2    3)
5
```

در این مثال تابع (lambda (x y) (+ x y)) روی آرگومانهای 2 و 3 اعمال شده و 5 به عنوان حاصل کل عبارت برگردانده شده.

مسلما در اکثر موارد ما نمی خواهیم از توابع اینطور استفاده کنیم. ما می خواهیم تابع را یک بار تعریف کنیم و به دفعات استفاده کنیم. برای اینکار کافی است که یک تابع (حاصل lambda) را به یک symbol انتساب دهیم. برای اینکار از define استفاده می کنیم. شکل کلی define به این صورت است:

```
(define  '(
        (sym1  (lambda ...))
        (sym2  (lambda ...))
        ...
         ))
```

(به علامت quote که قبل از آرگومان define آمده، توجه کنید)

توجه کنید که بوسیله define می توان چند تابع تعریف کرد.

حال دو تابع pls و mns را تعریف می کنیم که به ترتیب، مجموع و تفاضل دو عدد را برمی گرداند.

```
>>> (define '(
                (pls   (lambda   (a   b)    (+   a   b)))
                (mns   (lambda   (a   b)    (-   a   b)))
            )
    )
>>>  (pls    2   5)
7
>>>  (mns    (pls  3   6)    (mns    6     9))
12
```

همینطور که مشاهده می کنید وقتی یک تابع را صدا می زنیم، مقدارهایی که به عنوان آرگومان به تابع داده می شوند، در حقیقت به آرگومانهایی که در تعریف تابع وجود دارد منتسب می شود. این یکی از راههای انتساب یک مقدار به یک symbol است.

تا اینجای مقاله سعی شده که حداقل امکانات زبان lisp، بیان شود. یعنی امکاناتی که در اینجا توضیح داده شد، اکثرا مربوط به نسخه های اول این زبان بود. امروز زبان lisp خیلی تغییر کرده و امکانات زیادی به آن اضافه شده. در ضمن نسخه های متعددی هم از آن وجود دارد. مثلا ممکن است از این مقاله اینطور برداشت شود که در lisp حلقه یا دستور برای انتساب مستقیم متغیر وجود ندارد ولی در نسخه های موجود lisp این موارد

را ببینید. یا اینکه دستور شرط را به گونه ای دیگر در lisp مشاهده کنید. اینها مربوط به امکاناتی است که بعدها به زبان lisp اضافه شد.

نکته ای که باید در باره تعریف توابع گفته شود، این است که ممکن است در برخی نسخه های موجود از زبان lisp، تعریف توابع به صورت بالا (با استفاده از define) قانونی نباشد. (البته خود lambda در تمام نسخه ها وجود دارد). یک دستور دیگر برای تعریف توابع که در اکثر نسخه ها می توانید استفاده کنید (define function) **defun** است. که به صورت زیر استفاده می شود:

```
(defun  <function-name>
        <list-of-arguments>
        <return-value>)
```

برای مثال تابعی را تعریف می کنیم که جمع دو عدد را حساب کند:

```
>>> (defun   pls  (x  y)  (+ x y) )
```

با lisp برنامه بنویسید !

با اطلاعاتی که تا کنون در باره lisp بدست آوردیم، می توانیم اکثر برنامه ها را بوسیله زبان lisp پیاده سازی کنیم. ممکن است فقط نیاز به دستوراتی داشته باشیم که کارهای ساده ریاضی یا کارهای خاص دیگری را انجام می دهند. که مسلما این دستورات یکسری تابع هستند که آنها را می توانید در فهرست توابعی که هر نسخه lisp در اختیار شما می گزارد، پیدا کنید.

حال برای آشنایی بیشتر شما چند برنامه ساده می نویسیم. شاید حدس بزنید که این برنامه ها به صورت توابع lisp می باشند.

برنامه ای بنویسید که عدد n را بگیرد و $n!$ را برگرداند.

```
>>> (defun   fact   (n)
          (cond
                ((<   n    2)    1)
                (T       (*   n   (fact   (- n 1))))
           )
    )
```

این برنامه اگر $n < 2$ بود، مقدار 1 را برمی گرداند در غیر اینصورت مقدار `'(n*fact(n-1'` را برمی گرداند. ملاحظه می کنید که تابع fact خودش را صدا می زند.

تابع if:

این تابع در اکثر نسخه های جدید lisp وجود دارد. و در خیلی جاها کار را راحت تر می کند. فرمت کلی آن به این صورت است:

```
(if  <condition>  <exp1>  <exp2> )
```

این دستور *<condition>* را محاسبه می کند. اگر درست بود، *<exp1>* را مقدار یابی کرده و به عنوان حاصل تابع بر می گرداند در غیر این صورت *<exp2>* را برمی گرداند. توجه کنید که if هم مثل cond اول همه آرگومانهای خود را مقدار یابی نمی کند. بلکه آرگومان اول را محاسبه کرده و براساس جواب آن یکی از آرگومانهای دوم یا سوم خود را محاسبه می کند.

حال همان تابعfact را با استفاده از if می نویسیم:

```
>>> (defun  fact  (n)
                (if    (<  n   2)
                       1
                       (*  n  (fact  (- n 1)))
                 )
    )
```

یک مثال دیگر:

بیایید تابعی بنویسیم که یک list را به عنوان آرگومان بگیرد و طول آن را برگرداند.

تنها دستوراتی که بلدیم car و cdr است. حلقه هم که نداریم. پس باید یک رابطه بازگشتی (recursive) برای طول list پیدا کنیم:

```
If  L= ()  then   Length(L) = 0
Else  Length(L) = 1+ Length(CDR(L))
```

حال این رابطه را تبدیل به برنامه می کنیم:

```
(defun  length  (L)
                (if   (eq   L   NIL)
                      0
                      (+   1  (length   (cdr   L)))
                )
)
```

توجه کنید که اتم nil معادل لیست خالی است.

حال تابعی می نویسیم که لیست L و عدد N را بگیرد و n امین عضو L را برگرداند. و اگر n از محدوده L خارج است، مقدار nil را برگرداند.

سعی کنی اول خودتان این تابع را بنویسید. اگر نتوانستید راهنمایی زیر را بخوانید.

N امین عضو L همان N-1 امین عضو cdr(N) است. حالا شروع کنید به نوشتن تابع. بعد هم کد خود را با کد زیر مقایسه کنید.

```
(defun   getitem   (L  n)
          (cond    ((eq   L   nil)    nil)
                   ((eq   n   0)  (car   L))
                   (T             (getitem  (cdr L)  (-  n   1))
           )
)
```

تابع eval:

در ابتدای مقاله گفته شد که یک داده در lisp می تواند خودش یک برنامه باشد. مثلا داده ای به شکل (+ 1 2)، یک لیست سه عضوی است که عضو اول آن سیمبول + و اعضای دیگر آن اعداد 1 و 2 می باشند. اما می توان آن را به صورت یک برنامه در نظر گرفت.

به همین شکل لیست (defun add1 (x) (+ 1 x)) یک لیست است که دو عضو اول آنها اتم های defun و add1 و دو عضو دیگر آن هم دو لیست هستند.

حال فرض کنید داده هایی به این شکل داریم و می خواهیم آنها را به interpreter لیسپ بدهیم تا اجرا کند. مثل اینکه آنها را در خط فرمان lisp نوشته باشیم. این کار با تابع eval انجام می پذیرد. این تابع آرگومان خود را محاسبه و مقدار یابی می کند:

```
>>> (eval   '(+ 1 2))
3
```

می توان گفت کار eval برعکس کار ' است.

ناگفته ها:

در نسخه های جدید lisp دستور هایی برای انتساب متغیر وجود دارد، همچنین دستور حلقه. ولی ما آنها را به شما معرفی نمی کنیم و پیشنهاد می کنیم حداقل در ابتدای برنامه نویسی با lisp از آنها استفاده نکنید. (چون ممکن است شما را تنبل کنند!). سعی کنید تمرین های پایان این مقاله را خودتان حل کنید تا کمی بیشتر به برنامه نویسی با lisp عادت کنید.

ممکن است اول برنامه نویسی با lisp سخت به نظر بیاید ولی عادت می کنید. حتی برخی برنامه ها با lisp خیلی ساده تر می شود.

وقتی برخی از نسخه های جدید lisp را نگاه می کنیم، متوجه می شویم که خیلی جاها یکنواخت بودن زبان lisp را بهم زده اند. مثلا بین داده های معمولی و توابع فرق گذاشته اند. در صورتیکه می شد این تفاوت ها را برداشت. اگر علاقه دارید با یک زبان برنامه بنویسید که یکنواخت تر از lisp باشد، ما زبان SCHEME را پیشنهاد می کنیم. این زبان در حقیقت یکی از فرزندان زبان lisp است. در خیلی از جاها هم شبیه lisp

می باشد. ولی امکانات دیگری هم دارد.

نصب lisp و scheme:

نسخه های مختلف lisp را می توانید از طریق internet دریافت و نصب کنید. کافی است در یک جستجو گر مثل google کلمات " common lisp" + download را وارد کنید و در سایت مربوطه نسخه مربوط به سیستم عامل خودتان را downlad کنید.

اگر از سیستم عامل linux استفاده می کنید، معمولا یک نسخه از scheme به نام umb-scheme، در سیستم شما نصب شده. (کافی است در خط فرمان umb-scheme را وارد کنید) اگر می خواهید با یک نسخه کاملتر همراه با توابع کتابخانه ای بیشتر کار کنید، می توانید mit-scheme را download و نصب کنید.

Lisp و Emacs:

اکثر کاربران linux با ادیتور emacs آشنا هستند. (مخصوصا کاربران قدیمی تر) شاید امکانات این ادیتور برای زمان خودش خیلی زیاد بود! این ادیتور امکانات زیادی دارد و طوری طراحی شده که به راحتی می توان آن را توسعه داد و امکانات جدید به آن اضافه کرد. بسیاری از کاربران کنونی linux هنوز هم ادیتور emacs را به خیلی از محیط های گرافیکی برنامه سازی ترجیه می دهند. جالب است بدانید این ادیتور با زبان lisp نوشته شده است.

این editor امکاناتی برای کار با lisp (و scheme) دارد. کافی است کلید های Alt + x را فشار دهید و بنویسید run-lisp. با اینکار یک خط فرمان lisp درون خود emacs می آید که می توانید با آن کار کنید.

در واقع با این کارemacs دستور lisp را در shell یونیکس اجرا می کند. پس باید اول یک نسخه از lisp در سیستم شما نصب باشد و اجرای آن هم با دستور lisp انجام شود. (یعنی نام فایل اجرایی مربوطه lisp باشد.)

اگر فایل اجرایی lisp در سیستم شما با نام دیگری (مثلا clisp) موجود باشد، می توانید به شکل زیر عمل کنید:

اول محل آن را در filesystem پیدا کنید:

```
>>> which clisp
/usr/bin/clisp
```

بعد یک symbolic link به نام lisp از آن در یک مسیر شناخته شده بسازید:

```
>>> ln –s /usr/bin/clisp /usr/bin/lisp
```

حال می توانید با دستور run-lisp در emacs، آن را اجرا کنید.

چنین دستوری هم در emacs برای scheme وجود دارد

(run-scheme)

تمرینات:

1- تابعی بنویسید که n را بگیرد و n امین عدد از سری فیبوناچی را برگرداند.

2- تابعی بنویسید که یک لیست غیر تهی را بگیرد و عضو آخر آن را برگرداند.

3- تابعی بنویسید که لیست L و آرگومان n را بگیرد و محل اولین وقوع n در L را برگرداند.

4- تابعی بنویسید که لیست L و آرگومان n را بگیرد و لیستی را برگرداند که از حذف تمام n ها از L ساخته شده.

5- با استفاده از تابع eq تابعی به نام eq2 بنویسید که در مورد تساوی لیست ها هم درست عمل کند.

6- تابعی بنویسید که عدد n را بگیرد و لیستی شامل اعضای 1 تا n برگرداند (برای سادگی می توانید فرض کنید که لسیت از بزرگ به کوچک باشد)

7- تابعی بنویسید که یک list را بگیرد و معکوس آن را برگرداند.

8- تابعی بنویسید که دو لیست را بگیرد و لیستی را که از اتصال آنها به وجود می آید برگرداند.

9- سعی کنید برنامه هایی را که تا بحال در دروس مختلف نوشته اید با lisp یا scheme بنویسید.

پایان:

در این مقاله سعی شد تا به خواننده آشنایی مقدماتی با زبان lisp داده شود. و حداقل دانشی که برای برنامه نویسی با این زبان لازم است در اختیار خواننده قرار گیرد. مسلما برای یادگیری بیشتر این زبان و مشاهده مزیت ها و کاربردهای آن خود خواننده باید دست به کار شود و شروع به کار و تحقیق کند.

مسلما این مقاله چه از نظر تایپ و چه از نظر خود مطالب علمی، اشکالاتی دارد. اگر پیشنهاد یا انتقاد یا مطلب تازه ای در باره این موضوع دارید، می توانید به آدرس پست الکترونیکی نویسنده بفرستید:

nasihatkon {AT} cse.shirazu.ac.ir

References:

- Programming Languages Structures (Organick, Forsythe, Plummer) Academic Press 1978
- Artificial Intelligence (Luger) 2002
- Introduction to Scheme (?) Springer