به نام خدا

زبان‌های برنامه‌نویسی

آرش شفیعی

برنامه نویسی تابعی

برنامه نویسی تابعی

- در این فصل به معرفی برنامه نویسی تابعی و چند زبان برنامه نویسی تابعی می‌پردازیم.
- ابتدا با مفاهیم برنامه نویسی تابعی و حساب لامبدا که پایه و اساس زبان‌های تابعی است آشنا می‌شویم.
- سپس زبان‌های برنامه نویسی تابعی لیسپ، اسکیم، ام‌ال و هسکل را به اختصار معرفی می کنیم.
- در پایان به معرفی چندین تکنیک برنامه نویسی تابعی که در زبان‌های رویه‌ای و شیءگرا مانند پایتون می‌توانند مورد استفاده قرار بگیرند می‌پردازیم.

برنامه نویسی تابعی

- یکی از تفاوت‌های بنیادین زبان‌های تابعی و زبان‌های دستوری به شرح زیر است:
- در زبان‌های دستوری در هر لحظه در حین اجرا، برنامه دارای حالت است بدین معنی که تعدادی متغیر وجود دارند که مقادیر آنها مشخص است و نتیجه دستورات برنامه و همچنین نتیجه توابع به مقادیر این متغیرها بستگی پیدا می‌کند. برای مثال نتیجه یک تابع فقط وابسته به ورودی‌های آن نیست بلکه وابسته به حالت برنامه نیز هست. یک متغیر عمومی می‌تواند حالت برنامه را تغییر دهد. می‌گوییم زبان‌های دستوری دارای حالت [1] هستند.
- در زبان‌های تابعی برنامه‌ها بدون حالت هستند بدین معنی که متغیری وجود ندارد که حالت برنامه را تغییر دهد و نتیجه دستورات و همچنین توابع تنها وابسته به ورودی آنهاست. زبان‌های تابعی شباهت زیادی به زبان ریاضی دارند چرا که در زبان‌های تابعی همچون زبان ریاضی، محاسبات نتیجه اعمال چندین تابع بر ورودی است و نتیجه هر یک از توابع تنها به ورودی آن تابع بستگی دارد. می‌گوییم زبان‌های تابعی بدون حالت [2] هستند.

[1] stateful
[2] stateless

برنامه نویسی تابعی

- زبان‌های تابعی بر پایه حساب لامبدا به وجود آمده‌اند که توسط آلونزو چرچ ابداع شده است، در صورتی که زبان‌های دستوری (رویه‌ای و شیءگرا) بر اساس ماشین تورینگ که توسط آلن تورینگ ابداع شده است به وجود آمده‌اند.
- اساس ماشین تورینگ یک نوار است که در زبان‌های برنامه نویسی به متغیرها ترجمه می‌شود و یک کنترل‌کننده که در زبان‌های برنامه‌نویسی دستورات شرطی و حلقهٔ تکرار است.
- اساس حساب لامبدای چرچ تنها مفهوم تابع و اعمال توابع است.

برنامه نویسی تابعی

- زبان‌های برنامه نویسی تابعی مطمئن‌تر از زبان‌های دستوری هستند چرا که برای بررسی درستی برنامه تنها باید بررسی کنیم که هر یک از توابع نتیجه درست باز می‌گردانند.
- لیسپ یکی از زبان‌های برنامه نویسی تابعی بود که در ابتدا تنها توسط مفاهیم برنامه نویسی تابعی پیاده سازی شد و به عبارت دیگر یک زبان برنامه نویسی خالص بود ولی به مرور زمان مفاهیمی را از برنامه نویسی دستوری وام گرفت. این زبان هنوز بسیار مورد توجه و پر استفاده است.
- هسکل یک زبان برنامه نویسی تابعی دیگر است که یک زبان تابعی خالص باقی مانده است. هنوز هم در بسیاری از کاربردهای محاسبات ریاضی آماری این زبان به همراه زبان‌های دیگر تابعی مورد استفاده قرار می‌گیرند.

حساب لامبدا

- زبان‌های برنامه نویسی تابعی بر اساس حساب لامبدا به وجود آمده‌اند که در اینجا به معرفی آن می‌پردازیم.
- حساب لامبدا [1] در واقع یک دستگاه صوری [2] است برای توصیف محاسبات بر اساس توابع انتزاعی.
- یک دستگاه صوری یک ساختار انتزاعی (یک زبان قراردادی) است برای بیان اصول و قضایا در یک نظریه بر اساس تعدادی قوانین منطقی.

[1] lambda calculus
[2] formal system

حساب لامبدا

- حساب لامبدا تشکیل شده است از تعدادی متغیر و عباراتی که از متغیرها تشکیل شده‌اند، یک روش علامت‌گذاری [1] برای تعریف توابع، و مجموعه‌ای از قوانین برای اعمال یک تابع بر روی یک عبارت که قوانین کاهش [2] نامیده می‌شوند.
- در حساب لامبدا، توابع با حرف لامبدای یونانی (λ) تعریف می‌شوند و به همین دلیل اینگونه نام گرفته است.
- در حساب لامبدای ساده متغیرها بدون نوع هستند ولی حساب لامبدای نوع‌دار [3] نیز وجود دارد که در آن متغیرها نوع‌دار هستند.

[1] notation
[2] reduction rules
[3] typed lambda calculus

حساب لامبدا

- یک تابع در واقع یک قانون است که براساس مقادیر ورودی که آرگومان یا پارامتر نیز نامیده می‌شود مقادیر خروجی را تعیین می‌کند.
- برای مثال توابع $f(x) = x^2 + 3$ و $g(x, y) = \sqrt{x^2 + y^2}$ در ریاضی مورد مطالعه قرار می‌گیرند.

حساب لامبدا

- در حساب لامبدای خالص هیچ عملگری مانند جمع و تفریق وجود ندارد و تنها عملیات ممکن تعریف تابع و اعمال تابع است.
- بنابراین می‌توان محاسباتی را به صورت $h(x) = f(g(x))$ تعریف کرد.
- خواهیم دید که عملگرهای ریاضی را می‌توان با استفاده از توابع تعریف کرد.
- پس تنها دو ساختار در حساب لامبدا وجود دارند : انتزاع لامبدا [1] که برای تعریف تابع به کار می‌رود و عملیات اعمال [2] که برای اعمال تابع بر روی یک عبارت به کار می‌رود.

[1] lambda abstraction
[2] application

حساب لامبدا

- اگر M یک عبارت باشد، آنگاه $\lambda x.M$ تابعی است که x را دریافت می‌کند و M را به عنوان تابعی از x بازمی‌گرداند.
- برای مثال $\lambda x.x$ یک انتزاع لامبدا است که x را دریافت کرده و x را بازمی‌گرداند. به عبارت دیگر تابع همانی $I(x) = x$ توسط این انتزاع لامبدا تعریف می‌شود.
- در تعریف ریاضی یک تابع، همیشه باید نامی برای تابع در نظر بگیریم ولی در حساب لامبدا یک تابع بدون نام تعریف می‌شود.

حساب لامبدا

- برای اعمال یک تابع بر روی یک عبارت، تابع لامبدا را در یک پرانتز قرار می‌دهیم و عبارتی را که می‌خواهیم تابع بر روی آن انجام شود را در مقابل آن می‌نویسیم.
- برای مثال $(\lambda x.x)M$ ، تابع $\lambda x.x$ را بر روی عبارت M اعمال می‌کند و به دست می‌دهد :
$(\lambda x.x)M = M$
- عبارت M در اینجا می‌تواند هر عبارت دلخواهی تشکیل شده از تعدادی متغیر باشد.
- برای مثال $(\lambda x.x)(wyz)$ ، تابع $\lambda x.x$ را بر روی عبارت wyz اعمال می‌کند و به دست می‌دهد :
$(\lambda x.x)(wyz) = wyz$
- همچنین عبارت M می‌تواند عبارتی باشد که یک تابع را تعریف می‌کند.
- برای مثال $(\lambda x.x)(\lambda y.yy)$ ، تابع $\lambda x.x$ را بر روی عبارت $\lambda y.yy$ اعمال می‌کند و به دست می‌دهد :
$(\lambda x.x)(\lambda y.yy) = \lambda y.yy$

حساب لامبدا

- یک زبان برنامه نویسی می‌تواند توسط حساب لامبدا مدلسازی شود با این تفاوت که در زبان‌های برنامه نویسی نوع‌های داده‌ای وجود دارند. در واقع یک زبان برنامه نویسی معادل حساب لامبدای نوع‌دار است. می‌توانیم حساب لامبدای خالص را تعمیم دهیم به طوری که متغیرهای آن دارای نوع باشند.
- حساب لامبدا برای مدلسازی زبان‌های غیر تابعی نیز می‌تواند به کار رود چرا که حالت سیستم می‌تواند به عنوان یک ورودی به تابع لامبدا تعریف شود.

حساب لامبدا

- با استفاده از گرامر مستقل از متن می‌توانیم ساختار نحوی حساب لامبدا را به عنوان یک زبان برنامه نویسی سادهٔ بدون نوع تعریف کنیم.
- فرض می‌کنیم یک مجموعه نامحدود V از متغیرها داریم که معمولاً آنها را با x و y و z و غیره نشان می‌دهیم.
- گرامر حساب لامبدا به صورت زیر است :

$$M \rightarrow x \mid MM \mid \lambda x.M$$

- به طوری که x یک متغیر از مجموعهٔ V است.
- عبارت $\lambda x.M$ انتزاع لامبدا یا تعریف تابع و عبارت M_1M_2 اعمال تابع نامیده می‌شوند.
- در واقع $\lambda x.M$ تعریف تابعی است که x را به عنوان ورودی دریافت می‌کند و عبارت M را باز می‌گرداند و M_1M_2 اعمال تابع M_1 بر روی عبارت M_2 است.

حساب لامبدا

- برای مثال $\lambda x.(f(gx))$ تعریف تابعی است که به عنوان ورودی x را دریافت می‌کند و به عنوان خروجی، عبارت g را بر روی x و عبارت f را بر gx اعمال می‌کند.
- عملیات اعمال $(\lambda x.x)5$ تابع همانی را تعریف کرده و آن را بر روی 5 اعمال می‌کند.
- عبارت $f(gx)$ با عبارت $(fg)x$ متفاوت است. در عبارت اول ابتدا g بر روی x اعمال می‌شود و سپس f بر روی gx اعمال می‌شود، اما در عبارت $(fg)x$ ابتدا f بر روی g اعمال می‌شود و تابع به دست آمده بر روی x اعمال می‌شود.
- عبارت fgx در واقع به معنی $(fg)x$ است.
- اعمال تابع اولویت بالاتری نسبت به تعریف تابع دارد.
- بنابراین $\lambda x.MN$ به معنی $\lambda x.(MN)$ است، نه به معنی $(\lambda x.M)N$

حساب لامبدا

- دو عبارت $\lambda x.x$ و $\lambda y.y$ معادل هستند زیرا تابعی یکسان را تعریف می‌کنند و تنها اسامی ورودی آنها متفاوت است. دو عبارت یکسان که فقط در اسامی متغیرها متفاوت هستند را معادل آلفا [1] می‌نامیم. بنابراین می‌نویسیم :

$$\lambda x.x =_\alpha \lambda y.y$$

- در تابع $\lambda x.M$ ، عبارت M حوزهٔ تعریف [2] انقیاد λx نامیده می‌شود.
- همچنین در تابع $\lambda x.M$ ، عبارت M را بدنهٔ تابع و x را متغیر ورودی تابع می‌نامیم.

[1] α-equivalent
[2] scope

حساب لامبدا

- در یک عبارت، یک متغیر می‌تواند آزاد [1] یا مقید [2] باشد.
- یک متغیر آزاد متغیری است که تعریف نشده است و مقداری به آن انتساب داده نشده است. برای مثال در حساب ریاضی عبارت $(x + 3)$ غیر قابل محاسبه است زیرا x یک متغیر آزاد و تعریف نشده است. اما متغیر x در عبارت $f(x) = x + 3$ مقید است، زیرا تعریف شده است و می‌توان آن را مقداردهی کرد.
- نماد لامبدا، عملگر انقیاد [3] نیز نامیده می‌شود، زیرا یک متغیر را در یک عبارت تعریف می‌کند. متغیر x در عبارت $\lambda x.M$ مقید شده است، زیرا می‌توان به جای x هر مقداری را قرار داد و عبارت M را محاسبه کرد.

[1] free
[2] bound
[3] binding operator

حساب لامبدا

- متغیر x در بدنهٔ یک تابع مقید است اگر با استفاده از یک عملگر انقیاد مقید شده باشد.
- می‌توانیم تابع FV را به صورت زیر تعریف کنیم که متغیرهای آزاد یک عبارت را محاسبه می‌کند :

$$FV(x) = \{x\}$$
$$FV(MN) = FV(M) \cup FV(N)$$
$$FV(\lambda x.M) = FV(M) - \{x\}$$

- برای مثال

$$FV(\lambda f.\lambda x.(f(g(x)))) = \{g\}$$

حساب لامبدا

- در یک عبارت لامبدا، یک متغیر مقید یک بار به عنوان مقید کننده [1] و یک بار به عنوان مقید شده [2] به کار می‌رود.
- در عبارت $\lambda x.(\lambda y.xy)y$ اولین وقوع y مقید کننده، دومین وقوع y مقید شده و سومین وقوع y به عنوان متغیر آزاد است.
- معمولا از آنجایی که تکرار یک متغیر در یک عبارت وقتی حوزهٔ تعریف آن متفاوت باشد، می‌تواند گیج کننده باشد، نام متغیرها را می‌توانیم به نحوی تغییر دهیم که دو متغیر متفاوت با نام یکسان در یک عبارت وجود نداشته باشد، برای مثال عبارت پیشین را به صورت $\lambda x.(\lambda z.xz)y$ بازنویسی می‌کنیم.

[1] binding
[2] bound

حساب لامبدا

- زبان لیسپ شبیه حساب لامبدا طراحی شده است. تابع لامبدا در لیسپ را به صورت زیر می‌نویسیم.

```
(lambda (x)   function-body)
```

حساب لامبدا

- در یک عبارت در حساب لامبدا می‌توانیم یک متغیر را با یک متغیر دیگر جایگزین کنیم. برای مثال $[y/x]M$ بدین معناست که همه متغیرهای x در M با متغیر y جایگزین شوند، البته y نباید در M از قبل وجود داشته باشد.
- بنابراین می‌توانیم بنویسیم :

$$\lambda x.M = \lambda y.[y/x]M$$

حساب لامبدا

- در عبارت $(\lambda x.M)N$ در واقع $\lambda x.M$ عبارت M را به عنوان تابعی از x تعریف می‌کند. سپس این تابع را بر روی N را اعمال می‌کنیم. پس نتیجهٔ عبارت $(\lambda x.M)N$ عبارت M است که در آن همه متغیرهای x را با N جایگزین شده باشد. پس می‌توانیم بنویسیم :

$$(\lambda x.M)N = [N/x]M$$

- با استفاده از این قانون مقدار عبارت زیر را بدست می‌آوریم :

$$(\lambda f.fx)(\lambda y.y) = (\lambda y.y)x = x$$

حساب لامبدا

- از آنجایی که نام‌های یکسان در یک عبارت می‌توانند پیچیدگی‌های بسیاری ایجاد کنند، در اولین قدم برای ساده کردن یک عبارت، متغیرهای همنام که حوزهٔ تعریف آنها متفاوت است را تغییر نام می‌دهیم.
- در مثال زیر قبل از شروع محاسبات در پرانتز اول x را به z تبدیل می‌کنیم :

$$\begin{aligned}(\lambda f.\lambda x.f(fx))(\lambda y.y + x) &= (\lambda f.\lambda z.f(fz))(\lambda y.y + x) \\ &= \lambda z.((\lambda y.y + x)((\lambda y.y + x)z)) \\ &= \lambda z.((\lambda y.y + x)(z + x)) \\ &= \lambda z.(z + x + x)\end{aligned}$$

حساب لامبدا

- می‌توانیم قوانین جایگزینی را برای عبارت‌های متفاوت از تعریف جایگزینی به صورت زیر بدست آوریم :

$$[N/x]x = N$$
$$[N/x]y = y$$
$$[N/x](M_1M_2) = ([N/x]M_1)([N/x]M_2)$$
$$[N/x](\lambda x.M) = \lambda x.M$$
$$[N/x](\lambda y.M) = \lambda y.([N/x]M)$$

حساب لامبدا

- گفتیم با استفاده از انتزاع لامبدا می‌توانیم عبارت M را به عنوان تابعی از متغیر x به صورت $\lambda x.M$ بیان کنیم.
- اما چگونه می‌توانیم عبارت M را به عنوان تابعی از x و y در نظر بگیریم ؟
- می‌توانیم دو تابع تعریف کنیم به طوری که تابع اول متغیر x را دریافت کرده و تابعی بازمی‌گرداند که آن تابع متغیر y را به عنوان ورودی دریافت می‌کند.
- با استفاده از دو انتزاع لامبدا که هر کدام، یک متغیر دریافت می‌کند، می‌توانیم بنویسیم : $\lambda x.(\lambda y.M)$

حساب لامبدا

- محاسبات در حساب لامبدا با استفاده از کاهش [1] انجام می‌شوند.
- کاهش در واقع نوعی استدلال معادله‌ای [2] است.
- وقتی می‌نویسیم $(\lambda x.M)N = [N/x]M$ ، در واقع می‌گوییم یک گام کاهش انجام داده‌ایم.

[1] reduction
[2] equational reasoning

حساب لامبدا

- برای مثال :

$$(\lambda f.\lambda z.f(fz))(\lambda y.y + x) = \lambda z.((\lambda y.y + x)((\lambda y.y + x)z)$$
$$= \lambda z.z + x + x$$

- محاسبات تا جایی ادامه پیدا می‌کند که گامی برای کاهش وجود نداشته باشد. اگر عبارتی تا جایی کاهش پیدا کند که دیگر نتوان آن را کاهش داد می‌گوییم به یک عبارت فرم نرمال [1] رسیده‌ایم.

[1] normal form

حساب لامبدا

- برای مثال فرایند کاهش زیر را در نظر بگیرید :

$$\begin{aligned}(\lambda f.\lambda x.f(fx))(\lambda y.y+1)2 &= (\lambda x.(\lambda y.y+1)((\lambda y.y+1)x))2\\ &= (\lambda x.(\lambda y.y+1)(x+1))2\\ &= (\lambda x.(x+1+1))2\\ &= (2+1+1)\end{aligned}$$

- عبارت نهایی به دست آمده فرمال نرمال در فرایند کاهش است. اما اگر تابع + را تعریف کنیم، آنگاه می‌توانیم فرایند کاهش را ادامه دهیم :

$(2+1+1) = 3+1 = 4$

- در واقع تعریف می‌کنیم $x+y$ برابر است با اعمال تابع plus بر روی دو متغیر x و y بنابراین $x+y = \text{plus } x\ y$. سپس باید تابع plus را تعریف کنیم.

حساب لامبدا

- یکی از خواص حساب لامبدا این است که اگر در یک فرایند کاهش چند انتخاب در یک گام برای کاهش وجود داشته باشد، همهٔ انتخاب‌ها در نهایت به یک فرم نرمال واحد منجر می‌شوند. این خاصیت را تلاقی [1] می‌نامیم.
- برای مثال در عبارت $(\lambda x.(\lambda y.y+1)((\lambda y.y+1)x))2$ می‌توانیم ابتدا عبارت $((\lambda y.y+1)x)$ را محاسبه کنیم که به طور جداگانه در پرانتز قرار گرفته است و یا عبارت $(\lambda y.y+1)((\lambda y.y+1)x)$ را ابتدا محاسبه کنیم.

[1] confluence

حساب لامبدا

- کدگذاری چرچ [1] وسیله‌ای است برای نمایش داده‌ها و عملگرها در حساب لامبدا. همان طور که گفته شده هر نوع محاسباتی را که توسط یک مدل محاسباتی قابل انجام است، می‌توان توسط حساب لامبدا انجام داد.
- در اینجا نشان می‌دهیم چگونه می‌توان اعداد صحیح و چندین عملگر ساده را توسط حساب لامبدا نمایش داد.
- از آنجایی که در حساب لامبدا تنها ابزاری که در اختیار داریم توابع هستند پس تنها توسط توابع می‌توانیم اعداد را نشان دهیم.

[1] church encoding

حساب لامبدا

- می‌توانیم عدد n را بدین صورت تعریف کنیم : n بار اعمال تابع f بر روی x . بنابراین اعداد را به صورت جدول زیر نمایش می‌دهیم.

عدد	تابع	عبارت لامبدا
0	x	$\lambda f.\lambda x.x$
1	$f(x)$	$\lambda f.\lambda x.fx$
2	$f(f(x))$	$\lambda f.\lambda x.f(fx)$
3	$f(f(f(x)))$	$\lambda f.\lambda x.f(f(fx))$
⋮	⋮	⋮
n	$f^n(x)$	$\lambda f.\lambda x.f^n x$

حساب لامبدا

- حال باید توابعی را به عنوان عملگر تعریف کنیم که بر روی توابعی که به عنوان عدد تعریف شدند، اعمال شوند و عملیات محاسبات را انجام دهند.
- یکی از عملیات مقدماتی عملگر افزایش یک واحد به یک عدد است.
- عملگر افزایش واحد را می‌توانیم به صورت زیر نشان دهیم که در واقع اعمال یک بار تابع f بر عدد n است.

$$inc \equiv \lambda n.\lambda f.\lambda x.f(nfx)$$

حساب لامبدا

- می‌خواهیم مقدار عبارت inc 3 را محاسبه کنیم.
- در واقع باید تابع inc را بر روی تابع 3 اعمال کنیم.
- ابتدا معادل عبارت های inc و 3 را می‌نویسیم.

$\text{inc} \equiv \lambda n.\lambda f.\lambda x.f(nfx)$
$3 \equiv \lambda f.\lambda x.f(f(fx))$

حساب لامبدا

- حال محاسبات را به صورت زیر انجام می‌دهیم.

$$\begin{aligned}
\text{inc } 3 &= (\lambda n.\lambda f.\lambda x.f(nfx))(\lambda f.\lambda x.f(f(fx))) \\
&= (\lambda n.\lambda f.\lambda x.f(nfx))(\lambda g.\lambda y.g(g(gy))) \\
&= \lambda f.\lambda x.f((\lambda g.\lambda y.g(g(gy)))fx) \\
&= \lambda f.\lambda x.f((\lambda y.f(f(fy)))x) \\
&= \lambda f.\lambda x.f(f(f(fx))) \\
&= 4
\end{aligned}$$

- مقدار $\lambda f.\lambda x.f(f(f(fx)))$ همان کدگذاری عدد چهار است، بنابراین $\text{inc } 3 = 4$.

حساب لامبدا

- برای جمع دو عدد m و n کافی است ابتدا m بار و سپس n بار تابع f را بر روی x اعمال کنیم :

$$\text{plus} \equiv \lambda m.\lambda n.\lambda f.\lambda x.mf(nfx)$$

حساب لامبدا

- برای مثال می‌خواهیم دو عدد ۲ و ۳ را با یکدیگر جمع کنیم.
- توابع متناظر با عملگر جمع، عدد ۲، و عدد ۳ را به صورت زیر می‌نویسیم.

$$\text{plus} \equiv \lambda m.\lambda n.\lambda f.\lambda x.mf(nfx)$$
$$2 \equiv \lambda f.\lambda x.f(fx))$$
$$3 \equiv \lambda f.\lambda x.f(f(fx))$$

حساب لامبدا

- حال محاسبات را به صورت زیر انجام می‌دهیم.

$$
\begin{aligned}
\text{plus } 2\ 3 &= (\lambda m.\lambda n.\lambda f.\lambda x.mf(nfx))(\lambda g.\lambda y.g(gy)))(\lambda h.\lambda z.h(h(hz))) \\
&= ((\lambda n.\lambda f.\lambda x.(\lambda g.\lambda y.g(gy))f(nfx)))(\lambda h.\lambda z.h(h(hz))) \\
&= ((\lambda n.\lambda f.\lambda x.(\lambda y.f(fy))(nfx)))(\lambda h.\lambda z.h(h(hz))) \\
&= ((\lambda n.\lambda f.\lambda x.(f(f(nfx)))))(\lambda h.\lambda z.h(h(hz))) \\
&= \lambda f.\lambda x.(f(f((\lambda h.\lambda z.h(h(hz)))fx))) \\
&= \lambda f.\lambda x.(f(f((\lambda z.f(f(fz)))x))) \\
&= \lambda f.\lambda x.f(f(f(f(fx)))) \\
&= 5
\end{aligned}
$$

- مقدار $\lambda f.\lambda x.f(f(f(f(fx))))$ همان کدگذاری عدد ۵ است، بنابراین $\text{plus } 2\ 3 = 5$.

حساب لامبدا

- همهٔ عملگرهای حسابی دیگر از جمله تفریق، ضرب، تقسیم، و توان می‌توانند با استفاده از توابع حساب لامبدا تعریف شوند.

حساب لامبدا

- می‌توانیم این توابع را در زبان پایتون آزمایش کنیم.

```
zero = lambda f : lambda x : x
inc = lambda n : lambda f : lambda x : f(n(f)(x))
one = inc(zero)
two = inc(one)
three = inc(two)

f = lambda x : x+1
zero(f)(0) # 0
one(f)(0) # 1
two(f)(0) # 2
three(f)(0) # 3

plus = lambda m : lambda n : lambda f : lambda x : m(f)(n(f)(x))
five = plus(two)(three)
five(f)(0) # 5
```

حساب لامبدا

- برای تعریف مقادیر منطقی درست و نادرست می‌توانیم دو تابع به صورت زیر تعریف کنیم.
- مقدار درست تابعی است که دو ورودی می‌گیرد و ورودی اول را انتخاب می‌کند و مقدار نادرست تابعی است که دو ورودی می‌گیرد و ورودی دوم را انتخاب می‌کند.
- بنابراین داریم :

$\text{true} \equiv \lambda a.\lambda b.a$
$\text{false} \equiv \lambda a.\lambda b.b$

حساب لامبدا

- حال ساختار شرطی در حساب لامبدا را می‌توان تعریف کرد. یک گزاره اگر مقدارش درست باشد ورودی اول (then) را انتخاب می‌کند و اگر مقدارش نادرست باشد ورودی دوم (else) را انتخاب می‌کند.

```
predcate    then-clause    else-clause
```

- عملگر شرطی را به صورت زیر تعریف می‌کنیم.

$$\text{if} \equiv \lambda p.\lambda a.\lambda b.pab$$

- برای مثال :

$$\text{if true } M_1\ M_2 = \text{true } M_1\ M_2 = M_1$$
$$\text{if false } M_1\ M_2 = \text{false } M_1\ M_2 = M_2$$

حساب لامبدا

- همچنین می‌توان عملگرهای عطف و فصل و نقیض منطقی را به صورت زیر تعریف کرد.

$$\text{and} \equiv \lambda p.\lambda q.pqp$$
$$\text{or} \equiv \lambda p.\lambda q.ppq$$
$$\text{not} \equiv \lambda p.\lambda a.\lambda b.pba$$
$$\text{not} \equiv \lambda p.(p\ \text{false}\ \text{true})$$

- برای مثال :

$$\text{and true false} = (\lambda p.\ \lambda q.\ pqp)\ \text{true false} = \text{true false true} = \text{false}$$
$$\text{or true false} = (\lambda p.\ \lambda q.\ ppq)\ \text{true false} = \text{true true false} = \text{true}$$
$$\text{not true} = (\lambda p.(p\ \text{false true}))\ \text{true} = \text{true false true} = \text{false}$$
$$\text{not false} = (\lambda p.\lambda a.\lambda b.pba)(\lambda a.\lambda b.b) = \lambda a.\lambda b.a = \text{true}$$

حساب لامبدا

- در ریاضیات نقطه ثابت [1] در یک تابع، نقطه‌ای است که توسط یک تابع به خودش نگاشت می‌شود. به عبارت دیگر c یک نقطه ثابت در تابع f است اگر $f(c) = c$.
- حال ببینیم چگونه از مفهوم نقطه ثابت برای تعریف توابع بازگشتی در حساب لامبدا استفاده می‌کنیم.
- فرض کنیم f یک نقطه ثابت برای تابع G است. بنابراین می‌توانیم بنویسیم :

$$f = G(f) = G(G(f)) = G(G(G(f))) = \cdots$$

- بدین ترتیب بازگشت را توسط نقطه ثابت تعریف می‌کنیم.

[1] fixed point

حساب لامبدا

- عملگر نقطه ثابت در حساب لامبدا به صورت زیر تعریف می‌شود :

$$Y \equiv \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

- اگر G یک تابع باشد، آنگاه YG یک نقطهٔ ثابت برای تابع G است. می‌توانیم عملگر Y را به صورت زیر بر روی تابع G اعمال کنیم :

$$YG = (\lambda x.G(xx))(\lambda x.G(xx)) = G((\lambda x.G(xx))(\lambda x.G(xx))) = G(YG)$$

- بنابراین داریم :

$$YG = G(YG) = G(G(YG)) = \cdots$$

حساب لامبدا

- حال می‌خواهیم تابع فاکتوریل را تعریف کنیم. می‌توانیم تابع غیربازگشتی f را به صورت زیر بنویسیم :

$f \equiv \lambda n.\ (\text{if } (n == 1) \text{ then } 1 \text{ else } n * f(n-1))$

- سپس تابع G را به صورت زیر تعریف می‌کنیم :

$G \equiv \lambda f.\lambda n.\ (\text{if } (n == 1) \text{ then } 1 \text{ else } n * f(n-1))$

- همانطور که مشاهده می‌کنیم $f = G(f)$ بنابراین f یک نقطهٔ ثابت برای تابع G است. حال برای به دست آوردن نقطهٔ ثابت G عملگر نقطهٔ ثابت Y را بر روی G اعمال می‌کنیم.

- بنابراین تابع فاکتوریل در واقع یک نقطه ثابت برای تابع G است. پس می‌توانیم بنویسیم:

$fact \equiv YG$

$fact\ n = (YG)n$

حساب لامبدا

- برای مثال :

$$
\begin{aligned}
\text{fact } 2 &= (\mathsf{Y}\mathsf{G})2 \\
&= \mathsf{G}(\mathsf{Y}\mathsf{G})2 \\
&= (\lambda f.\lambda n.\ \text{if } n == 1 \text{ then } 1 \text{ else } n * f(n-1))(\mathsf{Y}\mathsf{G})2 \\
&= (\lambda n.\ \text{if } n == 1 \text{ then } 1 \text{ else } n * (\mathsf{Y}\mathsf{G})(n-1))2 \\
&= \text{if } 2 == 1 \text{ then } 1 \text{ else } 2 * ((\mathsf{Y}\mathsf{G})(2-1)) \\
&= 2 * ((\mathsf{Y}\mathsf{G})1) \\
&= 2 * ((\mathsf{G}(\mathsf{Y}\mathsf{G}))1) \\
&= 2 * (\lambda f.\lambda n.\ \text{if } n == 1 \text{ then } 1 \text{ else } n * f(n-1))(\mathsf{Y}\mathsf{G})1 \\
&= 2 * (\lambda n.\ \text{if } n == 1 \text{ then } 1 \text{ else } n * (\mathsf{Y}\mathsf{G})(n-1))1 \\
&= 2 * \text{if } 1 == 1 \text{ then } 1 \text{ else } 1 * ((\mathsf{Y}\mathsf{G})(1-1)) \\
&= 2 * 1 = 2
\end{aligned}
$$

لیسپ

- قدیمی‌ترین زبان برنامه نویسی تابعی که هنوز هم استفاده می‌شود، زبان لیسپ [1] است که در سال ۱۹۵۹ توسط جان مک کارتی [2] در مؤسسه فناوری ماساچوست [3] توسعه یافت.
- زبان لیسپ به مرور زمان تغییرات زیادی کرده است و نسخه‌های متعددی از آن توسعه داده شده‌اند. به جز نسخه اولیه که یک زبان تابعی خالص است، در بقیه نسخه‌ها مفاهیم برنامه نویسی دستوری نیز در زبان اضافه شده‌اند.

[1] Lisp
[2] John McCarthy
[3] Massachausetts Institute of Technology (MIT)

لیسپ

- در زبان لیسپ تنها دو نوع داده وجود دارد : اتم‌ها [1] و لیست‌ها [2]
- هر یک از عناصر یک لیست از دو قسمت تشکیل شده است. قسمت اول محتوای داده‌ای عنصر را در بر می‌گیرد که در واقع یک اشاره‌گر به یک اتم یا یک اشاره‌گر به یک لیست دیگر است. قسمت دوم عنصر یک لیست می‌تواند اشاره‌گر به یکی از عناصر دیگر لیست یا مقدار تهی باشد. عناصر لیست توسط قسمت دوم هر عنصر به یکدیگر متصل شده‌اند.
- لیسپ به گونه‌ای طراحی شده بود که برای کاربردهای پردازش لیست بتواند مورد استفاده قرار بگیرد.
- لیست‌ها می‌توانند ساده [3] یا تودرتو [4] باشند.

---

[1] atoms
[2] lists
[3] simple list
[4] nested list

لیسپ

- لیست‌های ساده به صورت دنباله‌ای از اتم‌ها درون پرانتز می‌توانند توصیف شوند. برای مثال `(A B C D)` یک لیست ساده با چهارعنصر است.
- لیست‌های تودرتو نیز با افزودن لیست‌ها به عنوان عناصر لیست‌های دیگر توصیف می‌شوند. برای مثال `(A (B C) D (E (F G)))` یک لیست تودرتو با چهار عنصر است.

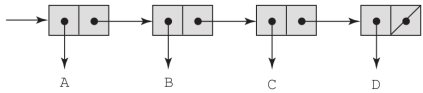
لیسپ

- در پیاده سازی لیسپ، لیست‌ها به صورت لیست‌های پیوندی [1] ساخته می‌شوند به طوری که اولین قسمت هر عنصر اشاره‌گر به دادهٔ آن عنصر و قسمت دوم عنصر برای تشکیل لیست پیوندی مورد استفاده قرار می‌گیرد.
- یک لیست توسط اشاره‌گری به اولین عنصر آن مشخص می‌شود و قسمت دوم آخرین عنصر لیست تهی [2] است.

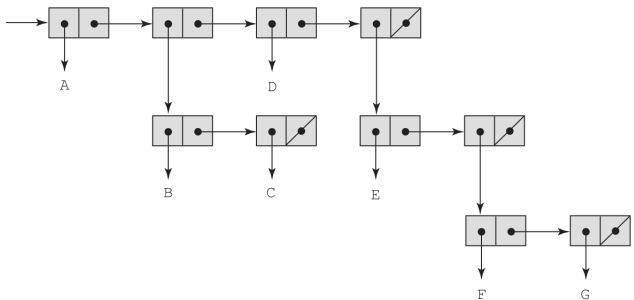
[1] linked list
[2] nil

لیسپ

- ساختار دو لیست در زبان لیسپ در شکل زیر نشان داده شده‌اند.



(A B C D)



(A (B C) D (E (F G)))

لیسپ

- در طراحی زبان لیسپ سعی شده است که قواعد نحوی همگن و ساده باشند تا بتوان توسط این زبان به راحتی محاسبه پذیری را مطالعه کرد، همان‌طور که محاسبه پذیری توسط ماشین تورینگ و حساب لامبدا استفاده می‌شود.
- بنابراین در زبان لیسپ فراخوانی توابع نیز مانند لیست‌ها درون پرانتز توصیف می‌شود. یک تابع به صورت `(function-name param-1 ⋯ param-n)` فراخوانی می‌شود.
- برای مثال اگر + تابعی باشد که مقادیر عددی را با هم جمع می‌کند، می‌توانیم آن را به صورت `( + 5 7 )` فراخوانی کنیم.
- همچنین برای تعریف توابع از همین نشانه‌گذاری [1] استفاده می‌شود.

```
(function-name (LAMBDA (param-1 ⋯ param-n) expression))
```

[1] notation

اسکیم

- زبان اسکیم [1] یکی از گویش‌های [2] زبان لیسپ است که در اواسط دهه ۱۹۷۰ در مؤسسه فناوری ماساچوست توسعه داده شد. در زبان اسکیم توابع می‌توانند عناصر یک لیست باشند یا به عنوان پارامتر به توابع دیگر ارسال شوند یا توسط توابع دیگر بازگردانده شوند.
- سادگی زبان اسکیم باعث شده است که این زبان در دانشگاه‌ها برای یادگیری برنامه نویسی تابعی مورد استفاده قرار بگیرد.

[1] scheme
[2] dialect

اسکیم

- در زبان اسکیم توابع ساده برای محاسبات عددی مانند جمع، تفریق، ضرب و تقسیم به صورت + ، - ، * ، / تعریف شده‌اند.
- توابع * و + می‌توانند تعداد صفر یا بیشتر پارامتر داشته باشند. اگر * صفر پارامتر داشته باشد مقدار یک را بازمی‌گرداند و اگر + صفر پارامتر داشته باشد مقدار صفر را بازمی‌گرداند.
- در عملیات تفریق همهٔ پارامترها به جز پارامتر اول از مقدار پارامتر اول کم می‌شوند. به همین ترتیب در عملیات تقسیم پارامتر اول بر پارامترهای دوم به بعد تقسیم می‌شود.
- توابع دیگری برای محاسبات ریاضی تعریف شده‌اند از جمله MODULO ، ROUND ، MAX ، MIN ، LOG ، SIN ، SQRT و غیره.
- یک برنامه اسکیم مانند هر برنامه دیگر در زبان تابعی مجموعه‌ای از فراخوانی توابع است.

اسکیم

- یک تابع بدون نام با کلمه کلیدی LAMBDA می‌تواند تعریف شود که این تعریف یک عبارت لامبدا [1] نام دارد. برای مثال :

```
( LAMBDA (x)  (* x x))
```

- این تابع یک ورودی دریافت می‌کند، بنابراین می‌توان آن را به صورت زیر با یک ورودی فراخوانی کرد :

```
(( LAMBDA (x)  (* x x)) 7 )
```

- در این عبارت متغیر x به مقدار ۷ مقید شده است. پس از انقیاد مقدار یک متغیر، مقدار آن دیگر تغییر نمی‌کند.

[1] lambda expression

اسکیم

- با استفاده از کلمهٔ کلیدی DEFINE می‌توان برای یک مقدار یا برای یک عبارت لامبدا، یک نام انتخاب کرد. در واقع کلمه DEFINE تنها مقادیر را نامگذاری می‌کند و نمی‌توان مقدار منتسب به اسامی را تغییر داد.
- با استفاده از عبارت تعریف می‌توان یک مقدار را به صورت (DEFINE symbol expression) نامگذاری کرد.
- برای مثال :

```
1 (DEFINE   pi  3.14159 )
```

اسکیم

- همچنین از عبارت تعریف، برای نامگذاری یک عبارت لامبدا نیز می‌توان استفاده کرد. در چنین مواقعی کلمه لامبدا حذف می‌شود. یک تابع لامبدا به صورت
(DEFINE (function-name parameters) (expression)) نامگذاری می‌شود.

- برای مثال تابع محاسبه مربع را به صورت زیر تعریف می‌کنیم.

```
1 (DEFINE   (square number)  (* number  number) )
```

- سپس برای محاسبه مربع یک عدد می‌نویسیم (square 5) که مقدار ۲۵ را باز می‌گرداند.

- مثال یک تابع دیگر در زیر آمده است که از تابع مربع برای محاسبه وتر مثلث قائم الزاویه استفاده می‌کند.

```
1 (DEFINE (hypotenuse side1 side2) (SQRT (+ (square side1)
2                                           (square side2))))
```

اسکیم

- تابع مسندی یا گزاره‌ای [1] تابعی است که یک مقدار منطقی (درست یا نادرست) باز می‌گرداند.
- اسکیم چندین تابع گزاره‌ای برای کار با مقادیر عددی دارد. از جمله = ، < ، > ، =< ، => برای برابری، بزرگتری، کوچکتری، بزرگتر یا برابری و کوچکتر یا برابری. همچنین توابع EVEN?، ODD?، ZERO? تعیین می‌کنند آیا یک عدد زوج یا فرد یا صفر است یا خیر.
- مقادیر درست و نادرست در اسکیم به صورت #T و #F تعیین می‌شوند. همچنین یک لیست خالی در اسکیم برابر با مقدار نادرست است.
- توابع AND، OR، NOT برای عطف، فصل و نقیض به کار می‌روند.

[1] predicate function

اسکیم

- زبان اسکیم دارای دو ساختار کنترلی است. اولی تابع IF است که اگر مقدار پارامتر دوم آن درست باشد پارامتر سوم را بازمی‌گرداند و در غیر اینصورت پارامتر چهارم را بازمی‌گرداند.
- بنابراین ساختار کنترل IF یک تابع به صورت `(IF predicate then-expr else-expr)` است.
- برای مثال تابع فاکتوریل را می‌توان به صورت زیر تعریف کرد :

```
1 (DEFINE  (factorial n)
2      ( IF (<= n 1)  1 (* n (factorial (- n 1))) )
3 )
```

اسکیم

- همچنین تابع COND یک ساختار کنترلی دیگر است که برای انتخاب یک گزینه از بین چندین گزینه استفاده می‌شود. اولین گزینه‌ای که مقدار آن برابر درست است محاسبه و بازگردانده می‌شود.
- تابع COND به صورت زیر تعریف می‌شود.

```
( COND    (pred-1 expr-1)
          (pred-2 expr-2)
          ...
          (pred-n expr-n)
          [(else expr)]
)
```

اسکیم

- برای مثال تابع زیر تعیین می‌کند آیا یک سال کبیسه است یا خیر.

```
(DEFINE    (leap? year)
     (COND
          ((ZERO? (MODULO year 400 )) #T)
          ((ZERO? (MODULO year 100 )) #F)
          (ELSE (ZERO? (MODULO year 4)))
     )
)
```

اسکیم

- یک برنامه اسکیم توسط تابع EVAL ارزیابی و اجرا می‌شود. تابع EVAL با هر تابعی که مواجه می‌شود، ابتدا پارامترهای آن را ارزیابی می‌کند و سپس خود تابع را ارزیابی و محاسبه می‌کند. توجه کنید که پارامترهای یک تابع می‌توانند خود فراخوانی توابع دیگر باشند که ابتدا باید محاسبه شوند. برای مثال فرض کنید تابع افزایش یک واحد را به صورت زیر تعریف کنیم.

```
۱ (DEFINE  (inc n) (+ n 1))
```

- حال فراخوانی زیر را در نظر بگیرید:

```
۱ (inc (inc 3))
```

- در اینجا ابتدا پارامتر تابع اول که (inc 3) است ارزیابی شده و مقدار ۴ بازگردانده می‌شود. سپس تابع اول به صورت (inc 4) ارزیابی شده و مقدار ۵ بازگردانده می‌شود.

- این برنامه را به صورت زیر می‌توان ارزیابی کرد.

```
۱ EVAL (inc (inc 3))
```

اسکیم

- حال فرض کنید در هنگام محاسبات، نمی‌خواهیم پارامترهای یک تابع ارزیابی و محاسبه شوند، بلکه می‌خواهیم پارامترها به عنوان لیست و اتم در نظر گرفته شوند.
- برای جلوگیری از ارزیابی شدن یک پارامتر در اسکیم از تابع QUOTE استفاده می‌شود. این تابع مقدار ورودی را بدون هیچ تغییری بازمی‌گرداند.
- برای مثال (QUOTE A) مقدار A را بازمی‌گرداند و ((QUOTE (A B C)) مقدار (A B C) را بازمی‌گرداند.
- فراخوانی (((QUOTE (inc (inc 3))) مقدار ((inc (inc 3)) را بازمی‌گرداند، در صورتی که فراخوانی ((inc (inc 3)) مقدار ۵ را بازمی‌گرداند.

اسکیم

- از آنجایی که در موارد زیادی نیاز به استفاده از تابع QUOTE می‌باشد، یک مخفف برای این تابع ساخته شده است که علامت آپوستروف (') است. بنابراین به جای (QUOTE (A B)) می‌توان نوشت '(A B).
- برای مثال فراخوانی '((inc (inc 3)) مقدار ((inc (inc 3)) را بازمی‌گرداند، در صورتی که فراخوانی ((inc (inc 3)) مقدار ۵ را بازمی‌گرداند.

اسکیم

- برای پردازش لیست‌ها سه تابع در اسکیم وجود دارد که عبارتند از CAR، CDR و CONS. تابع CAR اولین عنصر یک لیست را بازمی‌گرداند. تابع CDR همهٔ لیست به جز عنصر اول را بازمی‌گرداند.
- برای مثال (CAR '((A B) C)) مقدار (A B) را بازمی‌گرداند. (CAR 'A) خطا می‌دهد زیرا A یک لیست نیست و همچنین ((CAR '()) خطا می‌دهد چون یک لیست تهی عنصر اولیه ندارد. فراخوانی (CDR '((A B) C D)) مقدار (C D) را بازمی‌گرداند و ((A)' CDR) مقدار لیست تهی ( ) را بازمی‌گرداند.
- می‌توان تابعی به صورت زیر تعریف کرد که عنصر دوم یک لیست را بازمی‌گرداند.

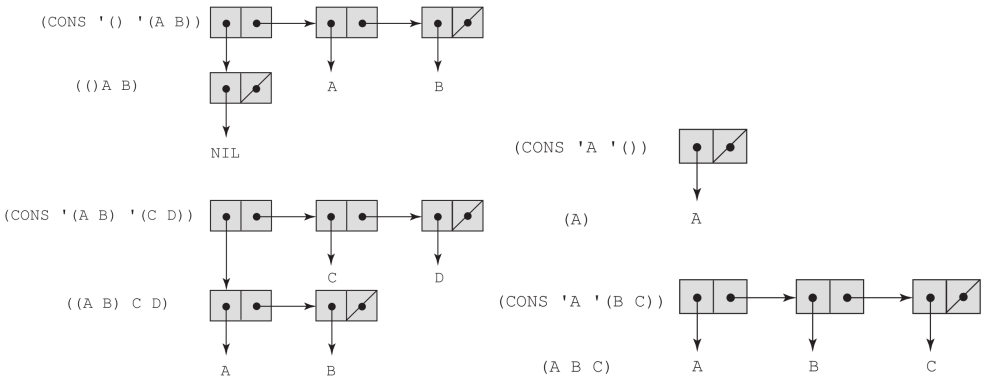
```
1 (DEFINE (second lst) (CAR (CDR lst)))
```

اسکیم

- توابعی نیز در اسکیم وجود دارند که ترکیب توابع CAR و CDR هستند.
- برای مثال (CADDDR x) برابر است با (CAR (CDR (CDR (CDR x)))) که چهارمین عنصر لیست را بازمی‌گرداند.
- همهٔ ترکیب‌های A و D تا چهار حرف به صورت تابع تعریف شده‌اند.
- تابع CONS برای ساختن یک لیست جدید با افزودن یک مقدار به ابتدای یک لیست استفاده می‌شود.
- برای مثال (CONS 'A '(B C)) مقدار (A B C) بازمی‌گرداند. همچنین (CONS '(A B) '(C D)) مقدار ((A B) C D) را بازمی‌گرداند.

اسکیم

- در شکل زیر نحوهٔ کار عملگر CONS نشان داده شده است.



اسکیم

- برای ساختن یک لیست توسط تعدادی اتم با استفاده از تابع CONS لازم است هر کدام از اتم‌ها را به طور مجزا به لیست اضافه کنیم.
- برای مثال (CONS 'apple (CONS 'orange (CONS 'grape '())))‌ لیستی از سه عنصر باز می‌گرداند.
- روش دیگر استفاده از دستور LIST است که تعداد دلخواهی عنصر را به صورت لیست در می‌آورد.
- برای مثال (LIST 'apple 'orange 'grape) لیست (apple orange grape) را بازمی‌گرداند.
- در اسکیم می‌توان برای تساوی دو مقدار از تابع ?EQ استفاده کرد. تابع ?LIST بررسی می‌کند ورودی یک لیست است یا خیر. همچنین تابع ?NULL بررسی می‌کند آیا یک لیست تهی است یا خیر.

اسکیم

- تابعی بنویسید که بررسی کند آیا یک اتم متعلق به یک لیست است یا خیر.
- برای مثال `(member 'B '(A B C))` مقدار `T#` و `(member 'B '(A C D))` مقدار `F#` را بازمی‌گرداند.

اسکیم

- در برنامه نویسی رویه‌ای معمولاً با استفاده از یک حلقه این کار را انجام می‌دهیم.
- در برنامه نویسی تابعی حلقه‌ها با استفاده از توابع بازگشتی توصیف می‌شوند.

```
(DEFINE (member  atm  lst)
      (COND
            ((NULL?  lst)  #F)
            ((EQ?  atm  (CAR  lst))  #T)
            (ELSE  (member  atm (CDR  lst))
       )
)
```

اسکیم

- تابعی بنویسید که دو لیست ساده را دریافت کرده و بررسی کند آیا دو لیست برابر هستند یا خیر. لیست ساده لیستی است که اعضای آن فقط اتم هستند.

اسکیم

```
(DEFINE (equalsimp list1 list2)
  (COND
    ((NULL? list1) (NULL? list2))
    ((NULL? list2) #F)
    ((EQ? (CAR list1) (CAR list2))
          (equalsimp (CDR list1) (CDR list2)))
    (ELSE #F)
  )
)
```

اسکیم

- تابعی بنویسید دو لیست معمولی را دریافت کند و بررسی کند آیا با یکدیگر برابرند یا خیر. دو لیست معمولی می‌توانند شامل اتم‌ها یا لیست‌های دیگر باشند.

اسکیم

```
۱  DEFINE (equal list1 list2)
۲    (COND
۳      ((NOT (LIST? list1)) (EQ? list1 list2))
۴      ((NOT (LIST? list2)) #F)
۵      ((NULL? list1) (NULL? list2))
۶      ((NULL? list2) #F)
۷      ((equal (CAR list1) (CAR list2))
۸              (equal (CDR list1) (CDR list2)))
۹      (ELSE #F)
۱۰   )
۱۱ )
```

اسکیم

- برنامه‌ای بنویسید که یک لیست را به یک لیست دیگر اضافه کند.
- برای مثال
  `(append '(A B) '(C D R))`
  لیست `(A B C D R)` را بازمی‌گرداند و
  `(append '((A B) C) '(D (E F)))`
  لیست `((A B) C D (E F))` را بازمی‌گرداند.

اسکیم

```
(DEFINE (append list1 list2)
  (COND
    ((NULL? list1) list2)
    (ELSE (CONS (CAR list1) (append (CDR list1) list2)))
  )
)
```

اسکیم

- تابع LET یک حوزهٔ تعریف محلی می‌سازد که در آن یک نام به یک مقدار انتساب داده می‌شود.
- معمولاً از تابع LET وقتی استفاده می‌کنیم که یک عبارت طولانی و پیچیده می‌شود و در نتیجه نیاز داریم قسمتی از عبارت را به صورت جداگانه با استفاده از یک نام تعریف کنیم.
- مقدار این اسامی را نمی‌توان تغییر داد، زیرا در برنامه نویسی تابعی تعریف متغیر وجود ندارد. تعریف متغیر باعث ایجاد حالت [1] می‌شود، در حالی که برنامه نویسی تابعی بدون حالت [2] است.

[1] state
[2] stateless

اسکیم

- برای مثال فرض کنید می‌خواهیم ریشهٔ یک معادله درجه دو را با استفاده از تابعی محاسبه کنیم. معادله درجه دو به صورت $ax^2 + bx + c$ است که ریشه‌های آن $-b/2a + sqrt(b^2 - 4ac))/2a$ و $-b/2a - sqrt(b^2 - 4ac))/2a$ هستند.

اسکیم

```
(DEFINE (quadratic_roots a b c)
  (LET (
    (root_part_over_2a
              (/ (SQRT (- (* b b) (* 4 a c))) (* 2 a)))
    (minus_b_over_2a (/ (- 0 b) (* 2 a)))
       )
  (LIST (+ minus_b_over_2a root_part_over_2a)
            (- minus_b_over_2a root_part_over_2a))
  )
)
```

اسکیم

- با استفاده از LET می‌توانیم عباراتی همانند عبارت لامبدا بنویسیم وقتی لامبدا بر روی مقداری اعمال می‌شود.

```
(LET ((alpha 7)) (* 5 alpha))
((LAMBDA (alpha) (* 5 alpha)) 7)
```

اسکیم

- به یک تابع بازگشتی از آخر [1] گفته می‌شود، اگر فراخوانی تابع بازگشتی آن آخرین فراخوانی در تابع باشد.
- تابع member را که قبلا پیاده سازی کردیم در نظر بگیرید.

```
(DEFINE (member atm a_list)
  (COND
    ((NULL? a_list) #F)
    ((EQ? atm (CAR a_list)) #T)
    (ELSE (member atm (CDR a_list)))
  )
)
```

- آخرین فراخوانی در این تابع، فراخوانی بازگشتی است و کامپایلر نیازی به نگهداری مقادیر فراخوانی‌های متعدد در این فراخوانی بازگشتی ندارد و آخرین فراخوانی بازگشتی مقدار نهایی تابع را به دست می‌دهد.

[1] tail recursive

اسکیم

- حال تابع فاکتوریل را در نظر بگیرید

```
(DEFINE (factorial n)
  (IF (<= n 1)
    1
    ( * n (factorial (- n 1)))
  )
)
```

- آخرین فراخوانی در این تابع، فراخوانی تابع ضرب است. پس برای محاسبه فاکتوریل کامپایلر نیاز دارد مقادیر همه فراخوانی‌های بازگشتی را نگه دارد تا پس از اتمام فراخوانی‌های بازگشتی به عقب بازگردد و مقدار نهایی را محاسبه کند.

اسکیم

- توابع بازگشتی از آخر سرعت بیشتری دارند و برنامه نویسان بهتر است سعی کنند توابع بازگشتی را به صورت بازگشتی از آخر بنویسند.
- برای مثال تابع فاکتوریل را می‌توان به صورت زیر بازنویسی کرد.

```
(DEFINE (fact n factval)
  (IF (<= n 1)
    factval
    (fact (- n 1) (* n factval))
  )
)

(DEFINE (factorial n) (fact n 1))
```

اسکیم

- تابع فاکتوریل بازگشتی از آخر را در زبان پایتون می‌توان به صورت زیر نوشت.

```
۱ def fact(n, factval) :
۲     if n<=1 :
۳         return factval
۴     else :
۵         return fact(n-1,n*factval)
۶
۷ def factorial(n) :
۸     return fact(n,1)
```

اسکیم

- فرض کنید می‌خواهیم با استفاده از دو تابع f و g تابع h(x) = f(g(x)) را محاسبه کنیم. می‌توانیم این مقدار را به طور دستی محاسبه کنیم. برای مثال

```
۱ (DEFINE (g x) (* 3 x))
۲ (DEFINE (f x) (+ 2 x))
۳ (DEFINE (h x) (+ 2 (* 3 x)))
```

- برای ترکیب [1] دو تابع می‌توانیم تابعی به نام compose به صورت زیر بنویسیم :

```
۱ (DEFINE (compose f g) (LAMBDA (x) (f (g x))))
```

[1] compose

اسکیم

- حال می‌توانیم ترکیب دو تابع را بر روی یک مقدار ورودی اعمال کنیم.

```
(DEFINE (g x) (* 3 x))
(DEFINE (f x) (+ 2 x))
(DEFINE (compose f g) (LAMBDA (x) (f (g x))))
((compose f g) 6)
```

- همچنین می‌توانیم از ترکیب دو تابع یک تابع تعریف کنیم.

```
(DEFINE (h x) ((compose f g) x))
(h 6)
```

اسکیم

- به عنوان مثال دیگر با استفاده از ترکیب توابع به صورت زیر می‌توانیم سومین عنصر یک لیست را محاسبه کنیم.

```
(DEFINE (third a_list)
  ((compose CAR (compose CDR CDR)) a_list))
```

- این تابع معادل تابع CADDR است.

اسکیم

- یکی از توابع مهم در برنامه نویسی تابعی، تابع نگاشت [1] است. این تابع یک تابع و یک لیست را به عنوان ورودی می‌گیرد و آن تابع را بر روی همه عناصر لیست اعمال می‌کند.
- به عبارت دیگر تابع map عملیات زیر را انجام می‌دهد.

```
(DEFINE (map fun a_list)
  (COND
    ((NULL? a_list) '())
    (ELSE (CONS (fun (CAR a_list)) (map fun (CDR a_list))))
  )
)
```

[1] map

اسکیم

- برای مثال فرض کنید می‌خواهیم همهٔ عناصر یک را به توان ۳ برسانیم. می‌توانیم بنویسیم :

```
(map (LAMBDA (num)  (* num num num)) '(3 4 2 6))
```

که لیست (27 64 8 216) را بازمی‌گرداند.

اسکیم

- مفسر اسکیم در واقع یک تابع است به نام EVAL که یک برنامه اسکیم را دریافت می‌کند و مقدار آن را محاسبه می‌کند. در واقع EVAL بر روی کل برنامه اعمال می‌شود و سپس هرکدام از اجزای آن به طور بازگشتی ارزیابی می‌شوند.
- برنامه نویسان اسکیم نیز می‌توانند از تابع EVAL استفاده کنند.

اسکیم

- فرض کنید می‌خواهیم برنامه‌ای بنویسیم که عناصر یک لیست را با هم جمع کند. می‌توانیم تابعی به صورت زیر تعریف کنیم.

```
(DEFINE (adder a_list)
  (COND
    ((NULL? a_list) 0)
    (ELSE (+ (CAR a_list) (adder (CDR a_list))))
  )
)
```

اسکیم

- این تابع به طور بازگشتی به صورت زیر محاسبات را انجام می‌دهد.

```
(adder '(3 4 5))
(+ 3 (adder (4 5)))
(+ 3 (+ 4 (adder (5))))
(+ 3 (+ 4 (+ 5 (adder ()))))
(+ 3 (+ 4 (+ 5 0)))
(+ 3 (+ 4 5))
(+ 3 9)
12
```

اسکیم

- با استفاده از تابع EVAL می‌توانیم این تابع را با استفاده از تابع عملگر + تعریف کنیم.

```
(DEFINE (adder a_list)
  (COND
    ((NULL? a_list) 0)
    (ELSE (EVAL (CONS '+ a_list)))
  )
)
```

- بنابراین تابع به صورت زیر محاسبه می‌شود.

```
(adder '(3 4 5))
(EVAL (+ 3 4 5))
12
```

ام‌ال

- ام‌ال [1] یکی دیگر از زبان‌های برنامه نویسی تابعی است. تفاوت آن با لیسپ و اسکیم در این است که در آن نوع همه داده‌ها در زمان کامپایل مشخص می‌شود.
- در زبان ام‌ال یک تابع به صورت `fun function-name (parameters) = expression` تعریف می‌شود.
- نوع‌ها اگر به صورت صریح تعیین نشده باشند، به صورت ضمنی توسط کامپایلر تشخیص داده می‌شوند برای مثال تابع زیر مقدار اعشاری real بازمی‌گرداند.

```
۱ fun circumf (r) =  3.14159 * r * r ;
```

[1] ML

ام‌ال

- می‌توانیم نوع مقدار خروجی یک تابع و یا نوع پارامترهای آن را نیز به صورت زیر مشخص کنیم.

```
۱ fun square (x) : real = x * x ;
۲ fun square (x : real) = x * x ;
```

- در زبان ام‌ال ساختار کنترلی if-then-else نیز وجود دارد.
- به طور مثال تابع فاکتوریل به صورت زیر محاسبه می‌شود.

```
۱ fun fact ( n : int ) : int   = if   n <= 1 then 1
۲                                else   n * fact (n-1) ;
```

ام‌ال

- یک تابع می‌تواند با استفاده از چند مقدار پارامتر متفاوت به صورت‌های متفاوت تعریف شود. تعاریف متفاوت یک تابع با استفاده از علامت (|) از یکدیگر جدا می‌شوند.
- برای مثال فاکتوریل را می‌توان به صورت زیر نیز تعریف کرد :

```
۱ fun fact (0) = 1
۲ | fact (1) = 1
۳ | fact (n : int) : int  = n * fact (n - 1);
```

ام‌ال

- در زبان لیسپ و اسکیم، اولین عنصر لیست را با استفاده از تابع CAR جدا می‌کنیم. در زبان ام‌ال این کار توسط عملگر (::) انجام می‌شود.
- برای مثال طول یک لیست را به صورت زیر می‌توانیم محاسبه کنیم.

```
۱ fun length ([ ]) = 0
۲ | length ( h :: t ) = 1 + length(t);
```

ام‌ال

- برنامه‌ای بنویسید که دو لیست را به یکدیگر الحاق کند.

```
fun append ([ ] , list2) = list2
| append ( h :: t , list2) = h :: append (t , list2);
```

- با استفاده از کلمهٔ کلیدی `val` می‌توان یک نام را به یک مقدار مقید کرد. البته مقدار را نمی‌توان بعد از انقیاد تغییر داد. این انقیاد مقدار به نام معمولاً برای ساده کردن عبارات به کار می‌رود.
- برای مثال :

```
let val radius = 2.7
    val pi = 3.14159
in pi * radius * radius
end;
```

ام‌ال

- توابع لامبدا در ام‌ال توسط کلمه fn تعریف می‌شوند. برای مثال `fn(x) => x < 100` یک تابع لامبدا است که اگر ورودی آن کوچکتر از ۱۰۰ باشد مقدار درست را بازمی‌گرداند.

ام‌ال

- تابع فیلتر دو ورودی می‌گیرد. ورودی اول آن یک تابع است که مقدار درست یا نادرست بازمی‌گرداند و ورودی دوم آن یک لیست است. تابع فیلتر هر یک از اعضای لیست را به عنوان ورودی به تابع ورودی آن می‌دهد. اگر تابع مقدار درست به ازای آن عنصر لیست بازگرداند، آن عضو به لیست خروجی تابع فیلتر افزوده می‌شود.
- برای مثال :

```
1 val lst = List.filter ( fn(x) => x<100 )
2                         [25, 1, 50, 710, 100, 7, 160, 3] ;
3 -- lst = [25, 1, 50, 7, 3]
```

ام‌ال

- تابع نگاشت تابع مهم دیگری است که دو ورودی می‌گیرد و یک لیست بازمی‌گرداند. ورودی اول آن یک تابع و ورودی دوم آن یک لیست است. تابع نگاشت تابع ورودی خود را بر روی همهٔ اعضای لیست ورودی اعمال می‌کند و به عنوان لیست خروجی بازمی‌گرداند.
- برای مثال :

```
۱ val lst = List.map (fn x =>  x * x * x)  [1, 3, 5] ;
۲ -- lst = [1, 27, 125]
```

- می‌توانیم از تابع نگاشت به صورت زیر نیز استفاده کنیم :

```
۱ fun cube  x = x * x * x ;
۲ val cubList = List.map cube ;
۳ val list = cubeList [1, 3, 5] ;
```

ام‌ال

- توابع ام‌ال همانند عملگر لامبدا در حساب لامبدا عمل می‌کنند. هر تابع فقط یک ورودی می‌گیرد.
- وقتی ورودی‌های یک تابع با علامت ویرگول جدا می‌شوند، در واقع ام‌ال ورودی‌ها را به عنوان یک ورودی چندتایی [1] در نظر می‌گیرد.
- اگر ورودی‌ها با ویرگول جدا نشود، ام‌ال به ازای هر ورودی یک تابع می‌سازد و ورودی بعدی را بر روی تابع ساخته شده اعمال می‌کند.

[1] tuple

ام‌ال

- برای مثال تابع زیر را در نظر بگیرید :

```
1 fun  add  a b = a + b ;
```

- این تابع را می‌توانیم با یک ورودی یا دو ورودی فراخوانی کنیم. اگر تابع با دو ورودی فراخوانی شود، حاصل جمع محاسبه می‌شود، اما اگر تابع با یک ورودی فراخوانی شود، یک تابع بازگردانده می‌شود. برای مثال اگر عبارت add 4 فراخوانی شود، تابع add4 به صورت زیر ساخته می‌شود.

```
1 fun  add4  b = 4 + b ;
```

- سپس می‌توانیم مقدار ۶ را به تابع add4 به عنوان ورودی ارسال کنیم که مقدار ۱۰ حاصل می‌شود.
- همچنین می‌توانیم تابعی به صورت زیر تعریف کنیم:

```
1 val  addfour = add 4 ;
2 val res = addfour 6;
3 -- res = 10
```

هسکل

- در زبان هسکل همانند ام‌ال نوع‌ها قبل از اجرای برنامه مشخص می‌شوند.
- هسکل یک زبان تابعی خالص است بدین معنی که عبارات حالت برنامه را تغییر نمی‌دهند (در ام‌ال امکان تعریف متغیر وجود دارد که باعث ایجاد حالت می‌شوند.) به عبارت دیگر می‌گوییم در هسکل هیچ دستوری اثر جانبی [1] ایجاد نمی‌کند.
- تابع فاکتوریل را در هسکل می‌توان به صورت زیر تعریف کرد.

```
۱ :{
۲ fact 0 = 1
۳ fact 1 = 1
۴ fact n = n * fact (n - 1)
۵ :}
```

[1] side effect

هسکل

- در هسکل توابع می‌توانند ورودی‌ها با نوع‌های متفاوت بگیرند.
- برای مثال در تابع `Square x = x * x` نوع x می‌تواند بسته به استفاده از تابع صحیح یا اعشاری باشد.
- عملگرهایی برای کار با لیست‌ها وجود دارند که در برنامه زیر به برخی از آنها اشاره شده است :

```
1 5 : [2, 7, 9]          -- results in  [5, 2, 7, 9]
2 [1, 3 .. 11]          -- results in  [1, 3, 5, 7, 9, 11]
3 [1, 3, 5]  ++ [2, 4, 6]          -- results in  [1, 3, 5, 2, 4, 6]
```

- عملگر : برای افزودن یک عنصر به لیست، عملگر .. برای تعریف لیست‌های طولانی با یک الگوی معین، و عملگر ++ برای افزودن دو لیست به یکدیگر استفاده می‌شوند.

هسکل

- در هسکل ورودی یک تابع بر روی تعریف انطباق داده می‌شود. به عبارت دیگر می‌گوییم اعمال تابع بر اساس تطبیق الگو [1] صورت می‌گیرد.
- برای مثال برنامه زیر را در نظر بگیرید :

```
:{
prod [] = 1
prod (a : x)  =  a * prod x
:}
```

- لیست ورودی تابع یا خالی است که بر روی الگوی اول تطبیق داده می‌شود، و یا دارای حداقل یک عنصر است که بر روی الگوی دوم تطبیق داده شده و توسط عملگر (:) در فرایند تطبیق الگو اولین عنصر لیست جدا می‌شود.

[1] pattern matching

برنامه زیر را در نظر بگیرید :

```
:{
tell [] = "The list is empty"
tell (x:[]) = "The list has one element: " ++ show x
tell (x:y:[]) = "The list has two elements: "
                ++ show x ++ " and " ++ show y
tell (x:y:_) = "This list is long. The first two elements are: "
                ++ show x ++ " and " ++ show y
:}
```

- وقتی تابع `tell` با یک ورودی فراخوانی می‌شود، ورودی بر روی یکی از حالات تعریف شده تطبیق داده می‌شود.
- عملگر ++ برای الحاق دو رشته یا دو لیست استفاده می‌شود. در تطبیق الگو کاراکتر زیرخط _ برای تطبیق هرگونه الگویی به کار می‌رود.

- در هسکل می‌توان لیست‌ها را به روشی به نام شمول کامل لیست [1] ایجاد کرد.
- برای مثال در زیر لیستی از همه اعداد بین ۱ تا ۵۰ به توان ۳ ایجاد شده است.

```
[n * n * n | n <- [1  ..  50]]
```

- تابع زیر به ازای عدد داده شدهٔ n لیست مقسوم علیه‌های آن را تولید می‌کند.

```
factor n = [ i | i <- [1 .. n `div` 2], n `mod` i == 0]
```

[1] List comprehension

هسکل

- یک زبان برنامه نویسی دارای مکانیزم ارزیابی کندرو [1] یا فراخوانی به هنگام نیاز است، اگر همه محاسبات را در هنگام فراخوانی انجام ندهد، بلکه محاسبات را به زمانی موکول کند که به آنها نیاز پیدا می‌شود.
- بر خلاف ارزیابی کندرو، در ارزیابی تندرو [2] ، محاسبات به محض فراخوانی انجام می‌شوند.
- برای مثال فرض کنید تابع h دو عدد به عنوان ورودی دریافت می‌کند. فرض کنید این دو ورودی را به صورت خروجی دو تابع f(x) و g(y) به تابع h ارسال کنیم. در ارزیابی تندرو، ابتدا هر دو ورودی f و g محاسبه می‌شوند و سپس مقادیر خروجی دو تابع به تابع h ارسال می‌شوند. در ارزیابی کندرو، اگر به ورودی دوم نیاز نباشد (برای مثال ورودی دوم در یک بلوک شرطی قرار داشته باشد و اجرا نشود)، در اینصورت تابع ورودی دوم ارزیابی نمی‌شود که باعث صرفه جویی در زمان می‌شود.

[1] lazy evaluation
[2] eager evaluation

هسکل

- مکانیزم ارزیابی کندرو باعث می‌شود بتوان عبارت‌هایی را بیان کرد که در زبان‌هایی با ارزیابی تندرو قابل بیان نیستند. برای مثال یک لیست با تعداد نامحدود عنصر را می‌توان در یک زبان با ارزیابی کندرو تعریف کرد. ولی در عمل تنها قسمتی از لیست محاسبه می‌شود که به آن نیاز است.
- بنابراین در زبان هسکل که دارای مکانیزم ارزیابی کندرو است، می‌توان لیست‌هایی به صورت زیر تعریف کرد.

```
۱ poisitive  =  [0  ..]
۲ evens  =  [2, 4  ..]
۳ squares = [n * n | n < - [0 ..] ]
```

- همهٔ این لیست‌ها دارای تعداد نامحدودی از مقادیر هستند ولی با تعریف آنها مقدار آن‌ها محاسبه نمی‌شود، چرا که در غیر اینصورت برنامه پایان ناپذیر می‌شد. بلکه تنها قسمتی از این لیست‌ها محاسبه می‌شود که به آنها نیاز است.

هسکل

- تابع بررسی عضویت یک عنصر در یک لیست را در زبان هسکل در نظر بگیرید :

```
:{
member   b [ ]  =  False
member   b ( a : x )  =  ( a == b ) || member b x
:}
```

- علامت || نمایانگر یای منطقی است. پس تابع درست را بر می‌گرداند، اگر اولین عنصر لیست برابر با مقدار b باشد و یا اینکه مقدار b در باقیمانده لیست باشد.
- حال فراخوانی member 16 squares را در نظر بگیرید. از آنجایی که لیست squares دارای تعداد نامحدودی عنصر است، این لیست تنها به مقداری محاسبه می‌شود که نتیجه فراخوانی به دست بیاید.

هسکل

- فرض کنید در یک برنامه، تابع f وجود دارد که تابع g را به عنوان ورودی دریافت می‌کند و می‌خواهیم مقدار `f(g(x))` را محاسبه کنیم. حال فرض کنید که g مقدار زیادی داده تولید می‌کند و f باید این داده‌ها را به ترتیب پردازش کند.

- در یک زبان برنامه نویسی با ارزیابی تندرو، f باید صبر کند تا g همهٔ داده‌ها را پردازش کند و تنها پس از اتمام پردازش g ، تابع f می‌تواند محاسبات را آغاز کند.

- اما در یک زبان با ارزیابی کندرو، به محض آماده شدن تعدادی از مقادیر توسط g ، تابع f آغاز به کار می‌کند چرا که داده‌های مورد نیاز را به دست آورده است. این مکانیزم باعث افزایش راندمان برنامه می‌شود.
همچنین g ممکن است تابعی باشد که پایان ناپذیر است ولی به محض اینکه f مقدار مورد نیاز را به دست آورد محاسبات پایان می‌پذیرد.

- قطعا چنین مکانیزمی بدون سربار و هزینه نخواهد بود. چنین انعطاف پذیری در یک زبان نیاز به توصیف معنایی پیچیده‌تر و همچنین پیاده سازی‌های پیچیده‌تر کامپایلر دارد که باعث می‌شود سرعت اجرای برنامه‌ها نیز کاهش پیدا کند.

پایتون

- پایتون زبانی است که پارادایم (الگوواره) های متعددی از جمله برنامه نویسی رویه‌ای، شیءگرا و تابعی را پشتیبانی می‌کند. در یک برنامه ممکن است قسمت‌های مختلف به روش‌های مختلف نوشته شوند. مثلاً برای پردازش لیست‌ها برنامه نویسی تابعی و برای طراحی ساختار داده‌ها و طراحی گرافیکی، برنامه نویسی شیءگرا می‌تواند مورد استفاده قرار بگیرد.
- همانطور که گفته شد، در برنامه نویسی تابعی ورودی برنامه به مجموعه‌ای از توابع ارسال می‌شود و هر تابع بر روی ورودی‌های خود عمل می‌کند و خروجی تولید می‌کند. در برنامه نویسی تابعی آثار جانبی [1] وجود ندارد بدین معنی که تابع حالت داخلی [2] ندارد و اینگونه نیست که خروجی تابع وابسته به حالت داخلی تابع باشد. پس خروجی تابع تنها وابسته به ورودی آن است. بنابراین هیچ ساختاری در یک زبان برنامه نویسی خالص نمی‌تواند وجود داشته باشد که مقدارش تغییر کند و به روز رسانی شود یا به عبارت دیگر دارای حالت باشد.

[1] side effect
[2] internal state

- برخی از زبان‌های برنامه نویسی تابعی، انتساب را ممنوع کرده‌اند تا از آثار جانبی جلوگیری کنند. اما در زبان پایتون انتساب وجود دارد و اگر بخواهیم به روش برنامه نویسی تابعی خالص برنامه نویسی کنیم باید در نظر داشته باشیم که خروجی تابع به حالت متغیرهای سیستم بستگی نداشته باشد. برای مثال از متغیرهای عمومی یا ایستا در روش برنامه نویسی تابعی نمی‌توان استفاده کرد.
- برنامه نویسی تابعی می‌تواند چندین مزیت داشته باشد که از جملهٔ آنها می‌توان به اثبات پذیری رسمی [1] برنامه، ماژولار [2] بودن برنامه‌ها، و سهولت تست [3] نام برد.

[1] formal provability
[2] modularity
[3] ease of test

- هدف از اثبات رسمی برنامه‌ها، این است که به صورت ریاضی بدون آزمایش و خطا اثبات شود که برنامه درست است. معمولاً برنامه نویسان با تست کردن برنامه‌ها توسط تعداد زیادی داده به این نتیجه می‌رسند که برنامه عملکرد درستی دارد اما ممکن است هنوز داده‌هایی باشند که برنامه برای آنها نتیجه نادرست تولید کند. در اثبات رسمی برنامه برای هر تابع اثبات می‌شود به ازای یک محدود از داده‌ها با ویژگی‌های معین، نتایج مورد نظر تولید می‌شود. البته از روش‌های اثبات درستی برای برنامه‌های بزرگ نمی‌توان استفاده کرد چرا که بسیار طولانی هستند، اما به فهم بهتر برنامه کمک خواهند کرد.
- مزیت دیگر برنامه نویسی تابعی در این است که برنامه‌ها به واحدهای کوچکتر یعنی توابع شکسته می‌شوند و راحت‌تر می‌توان برنامه را بررسی کرد و متغیر داد.
- همچنین تست کردن برنامه‌های تابعی بسیار ساده است چرا که کافی است نشان دهیم هر تابع نتیجه مورد نظر را تولید می‌کند و هیچ تابعی به حالت سیستم بستگی ندارد، پس درستی یک تابع درستی برنامه را نتیجه می‌دهد.

- از ویژگی مهم برنامه نویسی تابعی می‌توان به دریافت تابع به عنوان آرگومان توابع و بازگرداندن تابع از توابع دیگر اشاره کرد. همچنین از لیست‌ها در برنامه نویسی تابعی به کثرت استفاده می‌شود. همهٔ این ویژگی‌ها در زبان پایتون وجود دارد. لیست‌ها به عنوان یکی از انواع داده‌ای اصلی در پایتون استفاده می‌شوند و همچنین توابع را می‌توان به صورت لامبدا یا بدون نام و همچنین به صورت تابع نامگذاری شده به توابع دیگر به عنوان ورودی ارسال کرد. توابع می‌توانند تابع نیز بازگردانند.

- یکی از ویژگی‌های برنامه‌نویسی تابعی این است که می‌توان به توابع تابع ارسال کرد و از توابع تابع بازگرداند.
- در پایتون نیز می‌توان به یک تابع، یک تابع به عنوان ورودی ارسال کرد. برای مثال فرض کنید می‌خواهیم لیستی را مرتب کنیم. اما مرتب کردن لیست می‌تواند بر اساس معیارهای متفاوت صورت بگیرد. معیار مرتب سازی را می‌توانیم به صورت یک تابع به تابع مرتب‌سازی ارسال کنیم، تا مرتب‌سازی با استفاده از آن صورت بگیرد.

```
۱ def get_length(word):
۲     return len(word)
۳ words = ['apple', 'banana', 'cherry', 'date', 'strawberry']
۴ sorted_words = sorted(words, key=get_length)
۵ print(sorted_words)
۶ # ['date', 'apple', 'banana', 'cherry', 'strawberry']
```

- همچنین می‌توان از یک تابع تابع بازگرداند.
- برای مثال فرض کنید می‌خواهیم تابعی تعریف کنیم که بر اساس رشتهٔ ورودی (که ماژولی است که در آن خطا رخ داده است) تابعی تولید کند که آن تابع یک پیام خطا به همراه زمان ایجاد خطا تولید کند.

```
۱ def error(module):
۲     return lambda message : "In Module " + module + " ["
۳                 + datetime.now().strftime("%m/%d/%Y, %H:%M:%S")
۴                 + " ] : " +  message
۵ e = error("Test")
۶ e("wrong number")
۷ # 'In Module Test [11/28/2023, 08:09:06 ] : wrong number'
```

- در زبان پایتون می‌توان همانند هسکل از روش شمول کامل لیست [1] برای تولید لیست‌ها استفاده کرد.
- برای مثال :

```
۱ seq1 = 'a12'
۲ seq2 = (1, 2, 3)
۳ lst = [ (x,y) for x in seq1 for y in seq2  if x != str(y)]
۴ # lst = [('a',1),('a',2),('a',3),('1',2),('1',3),('2',1),('2',3)]
```

[1] List comprehension

پایتون

- عبارت مولد [1] در پایتون به صورت زیر توصیف می‌شوند.

```
(expression for expr1 in seq1 if cond1
            for expre2 in seq2 if cond2
            ...
            for expreN in seqN if condN)
```

[1] generator expression

پایتون

- در برنامه نویسی رویه‌ای، عبارات مولد به صورت زیر نوشته می‌شود که خوانایی پایین‌تری دارد و همچنین زمان بیشتری برای نوشتن برنامه صرف می‌شود:

```
for expr1 in seq1 :
    if not (cond1) :
        continue
    for expr2 in seq2 :
        if not (cond2) :
            continue
        ...
        for exprN in seqN :
            if not (condN) :
                continue
            ...
            # Output the value of the expression.
```

پایتون

- پیمایشگرها [1] اشیائی هستند که جریانی از داده‌ها را نشان می‌دهند. بر روی لیست‌ها می‌توان پیمایشگرهایی را داشت که عناصر یک لیست را پیمایش می‌کنند. توسط تابع iter() می‌توان یک پیمایشگر از یک لیست تولید کرد.
- برای مثال :

```
۱  L = [1, 2, 3]
۲  it = iter (L)
۳  n = next (it)     # n = 1
۴  n = next (it)     # n = 2
۵  T = (4, 5, 6, 7)
۶  itt = iter (T)
۷  n = next (itt)     # n = 4
۸  D = { 8 : 'a', 9 : 'b', 10 : 'c', 11 : 'd', 12 : 'e' }
۹  itd = iter (D)
۱۰ n = next (itd)     # n = 8
```

[1] iterator

- چندین تابع مهم وجود دارند که در برنامه نویسی تابعی بسیار مورد استفاده قرار می‌گیرند که در اینجا به آنها اشاره می‌کنیم.

پایتون

- تابع نگاشت یک تابع و یک پیمایشگر را به عنوان ورودی دریافت می‌کند و تابع ورودی را بر روی عناصری که پیمایشگر تولید می‌کند اعمال می‌کند.
- برای مثال :

```
1 def upper (s) :
2     return s.upper()
3 list (map (upper, ['a', 'b']))
4 # ['A', 'B']
5 list (map (sum, [[1, 2],[3,4]]))
6 # [3,7]
```

- تابع نگاشت را می‌توان با استفاده از شمول کامل لیست نیز به صورت زیر نوشت :

```
1 [s.upper() for s in ['a', 'b']]
```

- تابع فیلتر یک تابع را به عنوان ورودی اول و یک پیمایشگر را به عنوان ورودی دوم می‌گیرد. تابع ورودی باید مقدار منطقی درست یا نادرست بازگرداند. به ازای هر یک از عناصر تولید شده توسط پیمایشگر اگر مقدار خروجی درست بود، تابع فیلتر آن عنصر را در خروجی درج می‌کند.
- برای مثال :

```
1 def is_even(x):
2     return (x % 2) == 0
3 list (filter (is_even, range(10)))
4 # [0, 2, 4, 6, 8]
```

- تابع فیلتر را می‌توان با استفاده از شمول کامل لیست نیز به صورت زیر نوشت :

```
1 [x for x in range (10) if x % 2 == 0]
```

- تابع any در صورتی که یکی از عناصر لیست ورودی آن درست باشد مقدار درست بازمی‌گرداند و تابع all در صورتی که همه عناصر لیست ورودی آن درست باشد، مقدار درست بازمی‌گرداند.

```
۱ any ([0,1,0])      #True
۲ any ([0,0,0])      #False
۳ all ([0,1,1])      #False
۴ all ([1,1,1])      #True
```

- تابع zip یک عنصر از هر یک از پیمایشگرهای ورودی خود می‌گیرد و آنها را به صورت یک چندتایی درمی‌آورد.

```
list (zip ( ['a','b'] , [1,2,3] ) )
# [ ('a',1) , ('b',2) ]
```

- تابع کاهش، یک تابع به عنوان ورودی اول خود و یک پیمایشگر به عنوان ورودی دوم می‌گیرد. سپس تابع را بر روی هر یک از عناصر پیمایشگر اعمال می‌کند و خروجی تابع در هر گام اعمال را به عنوان ورودی تابع در گام بعد استفاده می‌کند.
- تابعی که به عنوان ورودی به تابع کاهش ارسال می‌شود، الزاما باید دو پارامتر ورودی داشته باشد.

```
1 from functools import reduce
2 def add(a,b) : return a+b
3 reduce (add , ['A','BB','C'])
4 # 'ABBC'
5 reduce (add , [1,2,3,4])
6 # 10
```

- همچنین تابع کاهش می‌تواند یک مقدار اولیه به عنوان سومین پارامتر دریافت کند.

```
1 def mul(a,b) : return a*b
2 reduce (mul , [1,2,3,4] , 1)
3 # 24
```

- توابع بدون نام لامبدا با استفاده از کلمهٔ lambda نوشته می‌شوند.

```
1 adder = lambda a,b : a+b
2 reduce(lambda a,b : a+b , [1,2,3,4] )
3 # 10
4 list (map (lambda x : x ** 3 , [2,4,6,8]))
5 # [8,64,216,512]
```

راست

- در طراحی زبان راست از برنامه نویسی تابعی بسیار تأثیر گرفته شده است.
- در برنامه نویسی تابعی توابع می‌توانند به عنوان پارامتر ورودی به توابع دیگر ارسال شوند و همچنین توابع می‌توانند توابعی را بازگردانند. همچنین در برنامه نویسی تابعی متغیر وجود ندارد، زیرا متغیرها باعث ایجاد حالت می‌شوند. در زبان راست نیز در حالت عادی با کلمه let می‌توان نماد تعریف کرد و اگر نیاز به متغیر بود باید به طور صریح با کلمه `mut` اعلام شود.
- در زبان راست به توابعی که می‌توان در متغیر ذخیره کرد بستار [1] گفته می‌شود. برای پیمایش و پردازش دنباله‌ها نیز از پیمایشگرها [2] استفاده می‌شود.
- بستارها در زبان راست در واقع توابع بی‌نام هستند که می‌توانند در یک متغیر ذخیره شوند یا به عنوان آرگومان به پارامترهای یک تابع ارسال شوند. برخلاف توابع که فقط به پارامترهای خود دسترسی دارند، بستارها می‌توانند به متغیرهای حوزهٔ تعریف خود نیز دسترسی داشته باشند.

[1] closure
[2] iterator

راست

- بستارها در راست در واقع همان توابع لامبدا در زبان‌های دیگر هستند.
- بستار را می‌توان به شکل‌های زیر با توجه به صریح و ضمنی بودن ورودی و خروجی آن تعریف کرد.

```
۱ fn  add_one_v1   (x: u32) -> u32 { x + 1 }
۲ let add_one_v2 = |x: u32| -> u32 { x + 1 };
۳ let add_one_v3 = |x|             { x + 1 };
۴ let add_one_v4 = |x|               x + 1  ;
```

راست

- اگر نوع داده‌های ورودی و خروجی بستار به طور صریح مشخص نشده باشند، در اولین فراخوانی کامپایلر برای آنها نوع تعیین می‌کند.

```
let example_closure = |x| x;
let s = example_closure(String::from("hello"));
let n = example_closure(5); //error: x is String
```

- یک بستار می‌تواند به متغیر هایی در حوزهٔ تعریف خود دسترسی داشته باشد که این دسترسی به سه نوع می‌تواند وجود داشته باشد : قرض گرفتن تغییرناپذیر [1] ، قرض گرفتن تغییرپذیر [2] و گرفتن مالکیت [3] .

[1] borrowing immutably
[2] borrowing mutably
[3] taking ownership

- در مثال زیر بستار مقدار متغیری را که از حوزهٔ تعریف گرفته [1] تغییر نمی‌دهد، پس دسترسی به صورت قرض گرفتن تغییرناپذیر است.

```
۱ fn main() {
۲     let list = vec![1, 2, 3];
۳     println!("Before defining closure: {:?}", list);
۴     let only_borrows = || println!("From closure: {:?}", list);
۵     println!("Before calling closure: {:?}", list);
۶     only_borrows();
۷     println!("After calling closure: {:?}", list);
۸ }
```

[1] cature

راست

- در برنامهٔ زیر بستار، متغیر تسخیر شده [1] را تغییر می‌دهد، پس دسترسی به صورت قرض گرفتن تغییرپذیر است.

```
fn main() {
    let mut list = vec![1, 2, 3];
    println!("Before defining closure: {:?}", list);
    let mut borrows_mutably = || list.push(7);
    borrows_mutably();
    println!("After calling closure: {:?}", list);
}
```

- توجه کنید که بعد از تعریف بستار و قبل فراخوانی آن یک متغیر ارجاعی تعریف شده که به لیست اشاره می‌کند پس نمی‌توانیم از `println!` استفاده کنیم چرا که این تابع متغیر ارجاعی تغییر ناپذیر تعریف می‌کند که مغایر با قوانین قرض گرفتن است.

[1] captured variable

راست

- وقتی می‌خواهیم مالکیت یک متغیر را به بستار انتقال بدهیم، از کلمه move استفاده می‌کنیم.
- برای مثال وقتی می‌خواهیم یک ریسه [1] کنترلی بسازیم، باید مالکیت را انتقال دهیم.

```
۱ use std::thread;
۲ fn main() {
۳     let list = vec![1, 2, 3];
۴     println!("Before defining closure: {:?}", list);
۵     thread::spawn(move || println!("From thread: {:?}", list))
۶         .join()
۷         .unwrap();
۸ }
```

[1] thread

راست

- فرض کنید در مثال قبل مالکیت لیست به ریسه داده نشود. در این صورت، ممکن است تابع `main` زودتر از ریسه به اتمام برسد در اینصورت در زمان اتمام، حافظه `list` را آزاد می‌کند و دسترسی ریسه به متغیر `list` غیر مجاز خواهد بود.
- اگر مالکیت متغیر تسخیر شده به ریسه انتقال داده نشود، کامپایلر پیام خطا صادر می‌کند که برای جلوگیری از خطر احتمالی توصیف شده است.

راست

- در راست برای بسیاری از ساختارهای داده پیمایشگر پیاده سازی شده است. از پیمایشگرها به صورت زیر استفاده می‌کنیم.

```
let v1 = vec![1, 2, 3];
let v1_iter = v1.iter();
for val in v1_iter {
    println!("Got: {}", val);
}
```

- همچنین تابع next یک پیمایشگر را مصرف می‌کند، بدین معنی که مقدار بعدی پیمایشگر را می‌خواند و از صف پیمایش آن را دور می‌ریزد.

```
۱ fn iterator_demonstration() {
۲     let v1 = vec![1, 2, 3];
۳     let mut it = v1.iter();
۴     println!("{:?}", it.next()); // Some(1)
۵     println!("{:?}", it.next()); // Some(2)
۶     println!("{:?}", it.next()); // Some(3)
۷     println!("{:?}", it.next()); // None
۸ }
```

- تابع sum را می‌توان بر روی یک پیمایشگر فراخوانی کرد : این تابع مجموع همهٔ مقادیر در پیمایشگر را با هم جمع می‌کند و پیمایشگر را مصرف می‌کند.

```
۱ fn iterator_sum() {
۲     let v1 = vec![1, 2, 3];
۳     let it = v1.iter();
۴     let total : i32 = it.sum();
۵     println!("{}", total); // 6
۶ }
```

- تابع نگاشت یا map بر روی یک پیمایشگر فراخوانی می‌شود و یک تابع دریافت می‌کند و تابع دریافتی را بر روی همه مقادیر پیمایشگر اعمال می‌کند و در نهایت یک پیمایشگر بازمی‌گرداند.

```
۱ fn map_demo() {
۲     let v1 = vec![1, 2, 3];
۳     let it = v1.iter();
۴     let v2 : Vec<_> = it.map(|&x| x + 1).collect();
۵     println!("{:?}", v2); // [2, 3, 4]
۶ }
```

- تابع فیلتر یا `filter` بر روی پیمایشگر تعریف شده است به طوری که یک تابع دریافت می‌کند که مقدار منطقی باز می‌گرداند. تابع دریافتی بر روی عناصر پیمایشگر اعمال می‌شود و عناصری که به ازای آنها مقدار درست بازگردانده شده است جمع‌آوری و بازگردانده می‌شوند.

```
۱ fn filter_demo() {
۲     let v1 = vec![1, 2, 3, 4];
۳     let it = v1.iter();
۴     let v2 : Vec<_> = it.filter(|&x| x % 2  == 0).collect();
۵     println!("{:?}", v2); // [2, 4]
۶ }
```

- تابع کاهش یا `fold` بر روی پیمایشگر تعریف شده است به طوری که یک مقدار اولیه و یک تابع با دو پارامتر را دریافت می‌کند و تابع را ابتدا بر روی مقدار اولیه و عنصر اول پیمایشگر و سپس به ترتیب بر روی خروجی تابع و مقادیر بعدی پیمایشگر اعمال می‌کند.

```
۱ fn fold_demo() {
۲     let v1 = vec![1, 2, 3, 4, 5];
۳     let it = v1.iter();
۴     let sum = it.fold(0, |a, b| a + b);
۵     println!("{sum}"); // 15
۶ }
```

- تابع `fold` در زبان‌های دیگر با نام `reduce` تعریف شده است.

راست

- نشان داده شده است که استفاده از پیمایشگرها و روش برنامه نویسی تابع از استفاده از حلقه در زبان راست سریع‌تر است.
- به عبارت دیگر این ساختارهای انتزاعی مانند نگاشت و فیلتر به صورت بهینه پیاده سازی شده‌اند و نمی‌توان آنها را به طور دستی بهینه‌تر پیاده سازی کرد.

برنامه نویسی تابعی

- به طور کلی گفته می‌شود برنامه نویسی تابعی چندین برتری نسبت به برنامه‌نویسی رویه‌ای دارد. یکی اینکه توصیف معنایی آن به دلیل ساختار ساده‌تری که دارد ساده‌تر است و دیگر آنکه اثبات درستی برنامه‌های آن آسان‌تر است.
- برخی بر این باورند که برنامه نویسی رویه‌ای برای برنامه نویسان راحت تر است ولی کسانی که برنامه نویسی را به صورت تابعی از ابتدا یاد گرفته‌اند بر این باورند که سختی برنامه نویسی تابعی به دلیل عادت نداشتن به آن است.
- برنامه نویسان تابعی معتقدند به دلیل اینکه برنامه‌ها توسط برنامه نویسی تابعی کوتاه‌تر و مختصرتر است راندمان برنامه نویسی در آن بالاتر است.
- همچنین در حال حاضر کامپایلرهای سریعی برای زبان‌های تابعی وجود دارد که با کامپایلرهای زبان‌های رویه‌ای و شیءگرا قابل مقایسه‌اند.

برنامه نویسی تابعی

- برنامه‌های تابعی به دلیل اختصار آنها برای خواندن نیز ساده‌ترند. برای مثال برنامه زیر به زبان سی را با معادل آن در هسکل مقایسه کنید

```c
int sum_cubes (int n) {
    int sum = 0 ;
    for (int index = 1 ; index <= n ; index ++)
       sum += index * index * index ;
    returne sum ;
}
```

```haskell
sumCubes n =  sum (map (^3) [1 .. n])
```

- همچنین از آنجایی که توابع در زبان‌های تابعی از یکدیگر مستقل هستند و حالت داخلی وجود ندارد اجرای آنها به صورت موازی توسط کامپایلر ساده‌تر است.