پردیس علوم
دانشکده ریاضی، آمار و علوم کامپیوتر

معرفی و بررسی زبان‌های تابعی

نگارنده

محمدمهدی احمدی

استاد راهنما: دکتر مجتبی مجتهدی

پایان‌نامه برای دریافت درجه کارشناسی
در رشته علوم کامپیوتر

چکیده

این مقاله حاوی دو بخش اصلی است. بخش اول جنبه‌ی آموزشی داشته و بخش دیگر جنبه‌ی تحلیلی. هدف کلی افزایش اطلاعات در مورد زبان‌های تابعی بوده و در انتها نیز به بررسی دلایل اهمیت این رده زبانی پرداخته شده است.

پیشگفتار

قرار است به سه سوال مهم در مورد زبان‌های تابعی پاسخ دهیم. اول اینکه یک زبان تابعی چیست و چه مفاهیمی پشت آن است؟ دیگر اینکه یک زبان تابعی چگونه کار می‌کند؟ و در آخر ببینیم چرا این زبان‌ها اهمیت پیدا کرده‌اند!

سعی شده تا بخش اول دید کافی و مناسب را در مورد دو سوال اول به خواننده بدهد، گرچه در بخش دوم نیز مفاهیم بسیار مهمی مطرح می‌شود. این کار در قالب یادگیری بخش‌هایی از حساب لاندا (یا لامبدا) انجام گرفته است. همچنین سعی شده تا مطالب به ساده‌ترین شکل ممکن بیان شود و پیشنیاز خاصی به جز مبانی ریاضیات و مفاهیم پایه‌ی نظریه محاسبه لازم نخواهد بود. تمرکز روی یادگیری حساب لاندا نیست، بلکه روی فهم این موضوع که چگونه می‌شود مسائل را با نگاه تابعی مدل کرد تاکید شده است.

در بخش دوم نیز، پس از بررسی ویژگی‌های مهم زبان‌های تابعی، کمی به بررسی جایگاه این زبان‌ها در دنیای برنامه نویسی امروز می‌پردازیم.

فهرست مطالب

فصل ۱

مفاهیم مقدماتی

همان طور که از اسم آن‌ها برمی‌آید، هدف اصلی پشت زبان‌های تابعی این است که بتوان مسائل را به کمک توابع مدل کرد و آن‌ها را حل نمود. همه‌ی ما می‌دانیم که تابع چیست و چگونه تعریف میشود. همچنین می‌دانیم که به دو صورت می‌توان توابع را شناسایی و با آن‌ها کار کرد. یک راه این است که دامنه و برد را داشته باشیم و بدانیم که کدام عنصر دامنه به کدام عنصر برد متناظر شده است. راه دیگر نیز آن است که ضابطه‌ی تابع را داشته باشیم، که در این صورت صحبت کردن در مورد تابع و بررسی آن ساده‌تر به نظر می‌رسد. برای کار با زبان‌های تابعی، به ترکیبی از این دو روش نیازمندیم. درواقع، هم باید بدانیم که این تابع روی ورودی مشخص چگونه عمل می‌کند، هم این که تعیین کنیم دامنه‌ی این تابع چیست و به ازای چه ورودی‌هایی کار می‌کند.

سوالی که احتمالا در این بخش برای شما مطرح می‌شود، این است که این کار به چه دردی می‌خورد؟ آیا این مدل‌سازی قدرت حل کافی را دارد؟ در مقایسه با بقیه‌ی مدل‌ها چطور عمل میکند؟ پاسخ این است که هر مدلی که تا کنون ارائه شده قدرت حل یکسانی در مقایسه با دیگر مدل‌ها دارد. برای درک بهتر این موضوع و مشاهده‌ی اثبات یکسان بودن قدرت حل ماشین تورینگ و حساب لاندا می‌توانید به مقاله‌ی [۱] مراجعه کنید.

فصل ۲

حساب لاندا

۱.۲ تعریف

حساب لاندا [۱] را می‌توان به عنوان کوچکترین زبان برنامه نویسی تابعی [۲] شناخت، که در حدود صد سال پیش، توسط دانشمندی به نام چرچ [۳] معرفی شد. هدف از این کار این بود که بتوان تمام مسائل را در قالب توابع، به شکلی فرمال، طرح و بررسی کرد. در این فصل به وسیله‌ی این زبان سعی می‌شود مفاهیم مورد نیاز برای درک چگونگی کار زبان‌های تابعی و نیز روش فکر کردن صحیح و متناسب با این محیط، به خواننده منتقل شود.

برای این که بدانیم عبارات در این زبان به چه صورت ساخته می‌شود، با توجه به فرمال بودن آن، از چند قاعده پیروی می‌کنیم که در ادامه توضیح داده می‌شود. بهتر است این قواعد را در قالب شناخت ماهیت اجزای این زبان پیدا کنیم، تا آن‌که صرفا یک تعریف ارائه شود. اولین و ساده‌ترین جزء، متغیرهای اولیه‌ی [۴] این زبان هستند. برای نمایش ساده‌ی این اجزا، از حروف الفبا استفاده شده است. این متغیرها نشان دهنده‌ی امکان وجود هر عبارتی می‌باشند و به معنای یک تابع خاص نیستند. پس هنگامی که می‌نویسیم x یا y، این دو درواقع با هم متفاوتند؛ ولی هر دو می‌توانند هر مقداری را آزادانه اختیار کنند حتی مقادیر یکسان. از این رو، آن‌ها را متغیر آزاد نیز می‌نامند. هر متغیر، خود یک عبارت است. از کنار هم قرار دادن عبارات، عبارات جدید تولید می‌شود که به آن اپلیکیشن [۵] می‌گوییم. تا اینجای کار هنوز توانایی تعریف یک تابع مشخص را نداریم؛ چراکه هر عبارت صرفا یا یک متغیر است یا ترکیبی از متغیرهای کنار هم قرار داده شده. پس به قاعده ای برای ساخت توابع نیاز داریم. فرض کنید x یک متغیر

---

[۱] lambda calculus
[۲] functional language
[۳] Alonzo Church
[۴] variable
[۵] application

و M یک عبارت باشد آنگاه $\lambda x.M$ یک تابع [۶] را نشان می‌دهد. این به چه معناست؟ به بیان ساده، مانند آن است که ورودی این تابع x و ضابطه‌ی آن نیز عبارت M است. در اینجا متغیر x دیگر وابسته به ورودی تابع است و اگر داخل M ظاهر شده باشد نمی‌توان هر مقدار دلخواهی را به آن اختصاص داد بلکه مقدار آن چیزی خواهد بود که تابع به عنوان ورودی می‌گیرد. به این متغیر، متغیر وابسته گفته می‌شود. پس در یک عبارت، متغیرها تحت λ وابسته می‌شوند و بقیه‌ی متغیرها آزاد هستند. همانطور که مشاهده می‌کنید، برخلاف نظریه مجموعه‌ها، در این زبان توابع با کاری که انجام می‌دهند تعریف و مشخص می‌شوند. توابع تنها یک ورودی می‌گیرند اما ساختن تابع با بیش از یک ورودی در این زبان کار بسیار ساده‌ای است.

اکنون تمامی عبارات این زبان را می‌شناسیم. هر عبارت یا یک متغیر، یا اپلیکیشن و یا تابع است. این را می‌توان یک تعریف بازگشتی مناسب برای ساخت تمامی عبارات در این زبان در نظر گرفت. بازگشتی بودن این تعریف از این رو است که در تعریف اپلیکیشن و تابع نیز از عبارات استفاده کرده‌ایم و به این صورت هر عبارت بزرگ را به عبارات کوچکتر تقسیم میکنیم تا به اجزای اولیه و پایه‌ای، که همان متغیرها هستند، برسیم.

$$\begin{aligned} < term > &:= < variable > | < application > | < function > \\ < application > &:= < term >< term > \\ < function > &:= \lambda < variable > . < term > \end{aligned}$$

حال باید دید که چگونه می‌شود یک تابع را روی یک ورودی اعمال کرد. برای مثال تابع $N \equiv \lambda x.xy$ را فرض کنید. اگر تابع $M \equiv \lambda z.zy$ را به عنوان ورودی به N بدهیم، با توجه به توضیحات گفته شده، منطقی به نظر می‌رسد که M به جای x در N قرار گیرد. اما همیشه به همین سادگی نیست. مثلا اگر $N \equiv \lambda x.(\lambda y.x)$ باشد، این جایگذاری با مشکل روبه‌رو می‌شود؛ زیرا متغیر y داخل M وابستگی نداشت، در حالی که پس از جای‌گذاری دیگر این طور نخواهد بود و این خوشایند نیست. برای حل این مسائل نیز راهکارهایی وجود دارد که به زودی به آن می‌رسیم. در ادامه خواهید دید که چگونه همه‌ی این فرایندها به صورت فرمال نوشته می‌شود. اما پیش از آن لازم است تا با چند مفهوم اساسی در این بحث آشنا شوید.

۲.۲ توابع

تا اینجای کار، تعاریف را دیدیم و متوجه شدیم عبارات این زبان به چند دسته تقسیم می‌شود و هر دسته چه مفهومی دارد. در این بخش برآنیم تا توابع و نحوه‌ی کار آن‌ها را بررسی کنیم و ببینیم یک تابع اصولا چگونه شناخته و از بقیه‌ی توابع متمایز می‌شود.

[۶] lambda abstraction

۱.۲.۲ جای‌گذاری

همانطور که در بخش قبل اشاره شد، هر تابع برای گرفتن ورودی نیازمند نوعی جای‌گذاری است. این جای‌گذاری ، خود نیازمند تعریفی دقیق است تا بتوان تمام اجزای زبان را تحت این عمل پوشش داد. پس بهترین کار این است که این تعریف نیز بازگشتی باشد تا به ازای هر عبارتی، کاری که عمل جای‌گذاری روی آن عبارت انجام می‌دهد مشخص شود. دو نوع جای‌گذاری تعریف می‌کنیم. جای‌گذاری متغیر توسط متغیر [۷] و جای‌گذاری متغیر توسط تابع [۸] . M را فرض کنید. میگوییم $M\{y/x\}$ جایگزینی x است توسط y در M و به صورت بازگشتی تعریف می‌کنیم:

$$x\{y/x\} \equiv y \quad (۱.۲)$$
$$z\{y/x\} \equiv z \quad (۲.۲)$$
$$(MN)\{y/x\} \equiv (M\{y/x\})(N\{y/x\}) \quad (۳.۲)$$
$$(\lambda x.M)\{y/x\} \equiv \lambda y.(M\{y/x\}) \quad (۴.۲)$$
$$(\lambda z.M)\{y/x\} \equiv \lambda z.(M\{y/x\}) \quad (۵.۲)$$

که در تمام آن‌ها $z \neq x$ فرض شده. همچنین M و N را فرض کنید. میگوییم $M[N/x]$ جایگزینی x است توسط N در M و تعریف میکنیم:

$$x[N/x] \equiv N \quad (۶.۲)$$
$$z[N/x] \equiv z \quad (۷.۲)$$
$$(MP)[N/x] \equiv (M[N/x])(P[N/x]) \quad (۸.۲)$$
$$(\lambda x.M)[N/x] \equiv \lambda x.M \quad (۹.۲)$$
$$(\lambda y.M)[N/x] \equiv \lambda y'.((M\{y'/y\})[N/x]) \quad (۱۰.۲)$$

۲.۲.۲ برابری آلفا

در حساب لاندا، توابع با اشکال مختلف ممکن است یکسان عمل کنند. از آنجا که در این زبان هر تابع با کاری که انجام می‌هد شناخته می‌شود، توابعی که کار یکسانی انجام می‌دهند طبیعتا باید در یک دسته قرار گیرند. حال این دسته بندی باید چگونه انجام شود؟ یکی از راه های شناخت توابع یکسان، استفاده از برابری آلفا [۹] است. به این صورت که میگوییم:

$$\lambda x.M =_\alpha \lambda y.(M\{y/x\})$$

[۷] renaming variable
[۸] substitution
[۹] α equivalence

گرچه هنوز تعریف دقیق ورودی گرفتن تابع را بیان نکرده‌ایم، ولی با توضیحات داده شده در بخش‌های قبلی می‌توان به طور شهودی گفت که هر دو تابع ذکر شده در بالا، به ازای ورودی یکسان، جواب یکسانی می‌دهند. از این برابری در جای‌گذاری متغیرها توسط توابع استفاده می‌شود. هنگامی که متغیری در تابع اصلی باشد که داخل تابع جایگزین شونده وابسته است، برای جلوگیری از به وجود آمدن مشکل، از این برابری استفاده می‌شود.

۳.۲.۲ کاهش بتا

اکنون دیگر می‌توانیم بحث توابع را تکمیل کنیم. حال که مفهوم جای‌گذاری را میدانیم، ورودی گرفتن توابع را اینگونه تعریف می‌کنیم:

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

و به آن کاهش بتا [۱۰] می‌گوییم. با اندکی تامل در تعریف، در می‌یابیم که اعمال توابع روی یکدیگر، در قالب اپلیکیشن ها اتفاق می‌افتد. پس اپلیکیشن ها چیزی بیش از قرار دادن دو عبارت در کنار هم هستند. ترکیب توابع نیز نوعی اپلیکیشن است. مثلا برای توابع M ، N و P ترکیب آن‌ها به صورت اپلیکیشن MNP به این معنی است که ابتدا تابع M روی تابع N اعمال می‌شود، و سپس نتیجه روی P اعمال خواهد شد. (پرانتز گذاری از چپ به راست انجام می‌شود) وقتی عبارتی را کاهش می‌دهیم، عبارت جدید نتیجه‌ی اعمال تابع روی ورودی آن است. پس می‌توان گفت که عبارت جدید با عبارت قبلی برابر است و آن را با $M \ =_\beta \ N$ نمایش می‌دهیم.

به عنوان مثال فرض کنید $M \equiv \lambda x.(\lambda z.zx)$ و $N \equiv \lambda y.zy$ باشد. می‌خواهیم اپلیکیشن MN را حساب کنیم. با کاهش بتا داریم:

$$(\lambda x.(\lambda z.zx))(\lambda y.zy) \ \rightarrow_\beta \ (\lambda z.zx)[\lambda y.zy/x]$$

حال از قواعد جای‌گذاری استفاده می‌کنیم:

$$(\lambda z.zx)[\lambda y.zy/x] \ \equiv \ (\lambda k.kx)[\lambda y.zy/x]$$

در این جا از برابری آلفا استفاده شد تا متغیر z آزاد بماند. در نهایت می‌رسیم به:

$$(\lambda k.k(\lambda y.zy))$$

تا اینجای کار، فهمیدیم که اجزای اصلی این زبان چیست. در ادامه سعی میکنیم به هدف اصلی، که همان مدلسازی پدیده ها و مسائل است، بپردازیم.

---

[۱۰] β reduction

۳.۲ مدل سازی

برای این کار لازم است تا زبان مورد نظر هم توانایی محاسبات عددی [۱۱] داشته باشد، هم قابلیت تحلیل منطقی [۱۲] را دارا باشد و هم اینکه بتوان در این زبان یک فرایند را به تعداد لازم تکرار نمود. در حساب لاندا برای این منظور، از مفهوم بازگشتی [۱۳] استفاده می‌شود.

۱.۳.۲ محاسبات عددی

ابتدا باید نمایشی از اعداد داشته باشیم. این نمایش را به این صورت می‌گیریم:

$$\begin{aligned} 0 &= \lambda fx.x \\ 1 &= \lambda fx.(fx) \\ 2 &= \lambda fx.(f(fx)) \\ &\dots \end{aligned}$$

حال به تابع افزاینده [۱۴] نیاز داریم تا بتوانیم هر عددی که خواستیم را بسازیم. کافیست بتوان به نمایش عدد n یک f اضافه کرد. مثلا:

$$n = \lambda fx.(fx) \;\Rightarrow\; succ\, n = \lambda fx.(f(fx))$$

ابتدا باید عدد n را به عنوان ورودی بگیریم. پس ابتدای تابع بهتر است با λs شروع شود که در آن s برای گرفتن عدد ورودی است. در ابتدای هر عدد یک λfx وجود دارد.اگر چیزی پیش از n در ضابطه‌ی تابع افزاینده قرارگیرد، دیگر نمایش عددی را نخواهیم داشت. مثلا اگر افزاینده به شکل $\lambda s.(fs)$ باشد، جواب به فرم $f\ \lambda fx \dots$ خواهد بود که فرم مطلوب ما نیست. برای حل این مشکل، کافیست با این روش، لاندای ابتدای عدد را حذف کنیم:

$$succ \;=\; \lambda nfx.(f(nfx))$$

به این شکل ابتدا f و سپس x به عنوان ورودی به تابع نمایش عدد داده شده و سپس خروجی به فرم مطلوب خواهیم داشت. البته در ابتدای تابع هم fx تحت لاندا قرارگرفت تا در نهایت فرم نمایش عدد به دست بیاید. دقت کنید که پرانتز گذاری در این مورد از اهمیت زیادی برخوردار است؛ چراکه اگر پرانتزها را نمی‌گذاشتیم، با توجه به ترتیب اپلیکیشن ها، این خروجی تغییر می‌کرد. تابع جمع و ضرب را نیز به

---

[۱۱] arithmetical calculations
[۱۲] logical operations
[۱۳] recursion
[۱۴] successor

همین صورت می‌سازیم:

$$\begin{aligned} add &= \lambda abfx.af(bfx) \\ mult &= \lambda abf.a(bf) \end{aligned}$$

می‌توانید این توابع را نیز با ورودی دادن به آن‌ها چک کنید.
پس به این شکل می‌توان توابع مختلف مورد نیاز در این بخش را ساخت.

۲.۳.۲ محاسبات منطقی

اولین چیزی که در منطق به آن نیاز داریم، مفهوم صحیح و غلط بودن است. در اینجا برای نمایش این مفهوم از دو تابع استفاده می‌کنیم.

$$\begin{aligned} T &\equiv \lambda xy.x \\ F &\equiv \lambda xy.y \end{aligned}$$

در واقع تابع صحیح همواره بین دو ورودی، ورودی اول را انتخاب می‌کند. برای تابع غلط هم به همین شکل همواره ورودی دوم را برمی‌گرداند. این دو تابع را برای چه به این صورت تعریف کردیم؟ پاسخ به این سوال را وقتی می‌یابید که بخواهید تابع if را تعریف کنید. تعریف این تابع بسیار ساده است:

$$if\ T\ M\ N\ =\ M$$

و یا

$$if\ F\ M\ N\ =\ N$$

در نتیجه تابع if به صورت $\lambda x.x$ خواهد بود. این تابع خیلی ساده به دست آمد. سعی کنیم تابع منطقی دیگری را بسازیم:

$$\begin{aligned} or\ T\ T\ &=\ T \\ or\ F\ T\ &=\ T \\ or\ T\ F\ &=\ T \\ or\ F\ F\ &=\ F \end{aligned}$$

می‌توان نشان داد که $\lambda ab.aab$ به ازای هر دو ورودی مقدار تابع or را برمی‌گرداند برای بقیه ی توابع نیز داریم:

$$\begin{aligned} and &= \lambda ab.aba \\ not &= \lambda a.aFT \end{aligned}$$

سعی کنید این توابع را بسازید:

$$
\begin{aligned}
&checkifzero: \\
&checkifzero\ n\ =\ F \\
&checkifzero\ 0\ =\ T
\end{aligned}
$$

که در آن n مخالف صفر است.

$$
\begin{aligned}
&pred: \\
&pred\ n\ =\ n-1
\end{aligned}
$$

در بخش بعد با استفاده از این توابع، توابع بازگشتی را می‌سازیم.

۳.۳.۲ مفهوم بازگشتی

این بخش را با یک مثال پیش می‌بریم. فرض کنید می‌خواهیم توانی از عدد دو را به صورت بازگشتی حساب کنیم.

$$
\begin{aligned}
power\ 0\ &=\ 1 \\
power\ n\ &=\ mult\ (power\ (pred\ n))\ 2
\end{aligned}
$$

که می‌شود:

$$power\ n\ =\ if\ (checkifzero\ n)\ 1\ (mult\ 2\ (power\ (pred\ n)))$$

می‌توان نوشت:

$$power\ =\ (\lambda fn.\ if\ (checkifzero\ n)\ 1\ (mult\ 2\ (f\ (pred\ n))))\ power$$

برای راحتی، به جای عبارت بزرگ، از F استفاده می‌کنیم. داریم:

$$power\ =\ F\ power$$

که داخل F تابع $power$ ظاهر نشده است. قرار دهید:

$$Y\ =\ (\lambda y.(\lambda x.y(xx))(\lambda x.y(xx)))$$

در این صورت

$$
\begin{aligned}
YF\ &=\ (\lambda x.F(xx))(\lambda x.F(xx)) \\
&=_\beta\ F((\lambda x.F(xx))(\lambda x.F(xx))) \\
&=\ FYF
\end{aligned}
$$

در نتیجه می‌توان گفت که

$$YF = power$$

به همین صورت می‌توان هر تابع بازگشتی را با استفاده از همین روش و به‌کمک Y به دست آورد. بالاخره موفق شدیم تا یک زبان برنامه نویسی تابعی خیلی ساده را بسازیم! حال که درک مناسبی از چگونگی کار این زبان‌ها پیدا کرده‌ایم، میتوانیم به بررسی زبان های تابعی دنیای کنونی و ویژگی‌های آن‌ها بپردازیم.

فصل ۳

زبان‌های تابعی

۱.۳ معرفی

زبان‌های برنامه نویسی تابعی بر اساس توابع ریاضی کار می‌کنند. هدف از طراحی این زبان‌ها، بیشتر جاهایی است که کار محاسبات با داده‌های نمادین [۱] یا کار پردازش لیستی مدنظر باشد. چند تا از معروف‌ترین این زبان ها عبارتند از: LISP ، Haskell ، Erlang .

زبان‌های تابعی را می‌توان به دو دسته تقسیم کرد. یکی زبان‌های تابعی محض [۲] و دیگری زبان‌های تابعی غیر محض [۳] زبان‌های تابعی محض فقط برنامه نویسی تابعی را پشتیبانی می‌کنند، مانند Haskell و زبان‌های غیر محض هم رده زبانی تابعی را پشتیبانی می‌کنند، هم مدل دستوری را؛ مانند LISP در فصل قبل مشاهده کردیم که زبان‌های تابعی چگونه مسائل را مدل می‌کنند. در این فصل ویژگی‌های به خصوص این زبان‌ها را بررسی می‌کنیم که موجب محبوبیت آن‌ها شده است.

۲.۳ ویژگی‌ها

در زبان‌های تابعی از دو مفهوم مهم استفاده شده. یکی مفهوم ارزیابی کندرو [۴] و دیگری مفهوم توابع مرتبه بالا [۵] است. این دو مفهوم از مهم‌ترین دلایل محبوبیت این زبان‌ها می‌باشند. در ادامه دوباره به سراغ این مفاهیم خواهیم رفت و بیشتر در موردشان صحبت می‌کنیم.

در این زبان‌ها داده ها قابل تغییر نیستند، بلکه هر بار داده‌ی جدیدی ساخته می‌شود. این، هم امری خوب است و هم می‌تواند مشکل‌زا باشد. این که داده‌ها همیشه ثابت هستند، در کنار دیگر ویژگی‌های

---

[۱] symbolic object
[۲] pure functional languages
[۳] impure functional languages
[۴] lazy evaluation
[۵] higher order functions

این زبان باعث شده تا میزان خطا و باگ [۶] در برنامه‌های نوشته شده به این سبک، به حداقل برسد. از طرفی هنگامی که برای هر کاری مجبور به ساختن یک شی [۷] جدید باشیم، باید برای کار با حافظه‌ی دستگاه نیز تدابیری بیندیشیم. در غیر این صورت دچار مشکلات جدی خواهیم شد. البته که راه‌های زیادی برای بهینه سازی استفاده از حافظه وجود دارد و در زبان‌های برنامه نویسی بزرگ همه‌ی این موارد در نظر گرفته شده است.

در این زبان ها استیت [۸] نداریم. در نتیجه بسیاری از اورها دیگر وجود نخواهد داشت. چون امکان به وجود آمدن آن‌ها وابسته به همین مورد است. با توجه به فرم این زبان‌ها و نوع محاسبات (که در فصل قبل دیدیم همگی توسط توابع انجام میگیرد) این زبان‌ها با پردازش موازی بسیار سازگار بوده و نتیجه‌ی مطلوبی می‌دهند. برای حساب کردن یک تابع بزرگ دیدیم که گاهی لازم است چندین تابع عملیات مختلفی را روی ورودی‌های متفاوت به صورت مستقل از هم انجام دهند. در این حالت می‌توان هر کدام از این بخش‌ها را به طور موازی و جداگانه حساب کرد که این کار سرعت رسیدن به نتیجه‌ی نهایی را خیلی بالا می‌برد.

در ادامه به بررسی دو مفهوم اساسی گفته شده می‌پردازیم.

۱.۲.۳ توابع مرتبه بالا

به توابعی گفته می‌شود که حداقل یکی از ورودی‌ها یا خروجیشان تابع باشد. ایده‌ی این کار همان حساب لاندا است و در فصل قبل به اندازه‌ی کافی از این توابع شناخت پیدا کرده‌ایم. این توابع بیشتر در کار با لیست ها و مپ ها [۹] استفاده می‌شوند. یکی از نقاط قوت تابعی بودن زبان ها در توانایی آن‌ها برای ماژول بندی برنامه هاست. یکی از عوامل مهم در این امر نیز وجود توابع مرتبه بالا است.

۲.۲.۳ ارزیابی کندرو

مفهوم ارزیابی کندرو، در مقابل ارزیابی سریع [۱۰] به این معنی است که هر مقدار را تا زمانی که نیاز به آن احساس نشده حساب نکنیم؛ همچنین اگر محاسبه‌ای انجام دادیم آن را ذخیره کنیم تا در صورت نیاز دوباره، مجبور به انجام عملیات مجدد نباشیم. از این مفهوم استفاده‌های بسیاری در موارد مختلف جهت بهینه سازی سرعت محاسبات صورت گرفته است. البته این کار برای جاهایی قابل استفاده است که اولا مطمئن باشیم عبارت مورد نظر در طول برنامه مقدار ثابتی دارد [۱۱] و همچنین بدانیم که انجام این کار تاثیر جانبی [۱۲] بر برنامه نمی‌گذارد. برای مثال اگر محاسبات وابسته به تاریخ و زمان باشد، نمی‌توانیم از این

---

[۶] bug
[۷] object
[۸] state
[۹] maps
[۱۰] eager evaluation
[۱۱] referential transparency
[۱۲] side effect

روش بهره ببریم. در این مواقع باید از روش ارزیابی سریع استفاده کنیم. مفهوم ارزیابی سریع، همانطور که از نامش برمی‌آید، به معنی آن است که هر جا عبارتی را دیدیم همان لحظه مقدار آن را محاسبه کنیم. در ارزیابی کندرو مقدار هر عبارت یا کلا حساب نمی‌شود، که یعنی در برنامه مورد دسترسی قرار نگرفته، یا فقط یکبار این محاسبه صورت می‌گیرد. این مفهوم فقط برای محاسبه نیست، بلکه برای استفاده از حافظه نیز مطرح می‌شود. دقت کنید که در صورت اجرایی بودن این روش، تا حد زیادی در استفاده از زمان و فضا صرفه جویی می‌شود. هنگامی که با توابع محض سروکار داریم، عملا تغییراتی در کار نخواهد بود. همانطور که قبلا دیدیم، توابع تاثیری روی اطلاعات برنامه ندارند و از طرفی تنها چیزی که در مورد آن‌ها مهم است مقدار خروجی تابع می‌باشد. از این رو، زبان‌های تابعی برای پیاده سازی این مفهوم، محیط مناسبی را فراهم می‌کنند و با این مفهوم سازگاری بالایی دارند که باعث می‌شود بتوان بهینه سازی های مهمی روی آن‌ها انجام داد. تا حدی که در زبانی مانند Haskell تمامی فرایند ها با تاخیر صورت میگیرد و تمام آن با ارزیابی کندرو انجام می‌شود. این روش برای محاسبات جبری نیز بسیار مناسب است. در مقاله‌ی [۴] چند نمونه از بهینه سازی‌های ممکن با دو مفهوم ارزیابی کندرو و توابع مرتبه بالاتر در زبان‌های تابعی بیان شده است.

۳.۳ جایگاه کنونی

زبان‌های تابعی چند سالی است که بین برنامه نویسان به محبوبیت رسیده‌اند. البته همچنان این زبان‌ها به پیچیدگی معروف هستند. اما هر چه می‌گذرد موارد استفاده و استقبال از این زبان‌ها بیشتر می‌شود. دلایل علمی این محبوبیت و ویژگی‌های برجسته‌ی این زبان‌ها را تا حدودی بیان کردیم. در اینجا به بررسی کلی جایگاه این زبان‌ها در دنیای برنامه نویسی روز با توجه به ویژگی‌های بررسی شده در بخش‌های قبل خواهیم پرداخت.

با توجه به فرم برنامه‌های تابعی، کدها مختصر و مفید هستند. با حجم کد کمتر، کار بیشتری انجام می‌دهید و همچنین خوانایی و قابلیت فهم کد بالاتر می‌رود. اینگونه هم زمان بیشتری ذخیره می‌شود و هم ایرادات کمتری پیش می‌آید. علاوه بر این، توضیح کد به دیگران ساده‌تر خواهد بود. به این شکل، کمپانی‌ها و شرکت‌ها می‌توانند زودتر تیم‌های مورد نظر خود را سامان دهی کنند و همچنین افراد جدید راحت تر و در زمان کمتر با کار همراه می‌شوند. این امر باعث بالا رفتن بازدهی خواهد شد.

در حال حاضر، زبان‌های تابعی در مواردی که نیاز به پردازش موازی، محاسبات ریاضی و جبری، و یا عملیات متعدد و گوناگون روی دیتاست ثابت باشد؛ با توجه به دلایلی که گفته شد، بهتر عمل می‌کنند. با توجه به این که اکثر پروژه‌های این روزها به نوعی با این موارد درگیر است، طبیعتا تمایل استفاده از این نوع زبان‌ها و گرایش به سمت آن نیز زیاد می‌شود. برای مثال در بحث داده‌های بزرگ یا big-data حجم اطلاعات بسیار بیشتر از ظرفیت یک یا چند سرور است. در روش‌های قدیمی و غیر تابعی که توسط مدل ها و رده زبانی‌های دیگر اجرا می‌شود، امکان تغییر اطلاعات در حین اجرا همواره وجود دارد. مثلا ممکن است مقدار یک متغیر غیرمحلی یا بخشی از یک لیست ناخواسته تغییر کند. همچنین در برخی زبان‌ها امکان دسترسی به متغیرها در موارد مختلف (مثلا توابعی که به صورت call by reference

کار میکنند) وجود دارد. هنگامی که با این حجم از داده کار می‌کنیم، چنین اتفاقاتی می‌تواند خطاهای زیادی ایجاد کند. مخصوصا که در این موارد اعمال زیادی به صورت موازی روی دستگاه‌های متعدد در حال اجرا شدن است. از طرفی در ابتدای فصل بررسی کردیم و توضیح داده شد که زبان‌های تابعی برای اینگونه موارد بسیار مناسب هستند. به دلیل اینکه تاثیر جانبی ندارند چراکه همه چیز تابع است و توابع اطلاعات را تغییر نمی‌دهند. ساختار داده‌ها در زبان‌های تابعی، برخلاف زبان‌های شی گرایی، غیر قابل تغییر است. از طرف دیگر، به خاطر فرم تابعی (بخش های جدا از هم که هر یک خود به تنهایی تابع است) با قابلیت پردازش موازی سازگاری بالایی دارند. همچنین در مواردی که زمان یا حافظه‌ی زیادی مورد استفاده است از ارزیابی کندرو و توابع مرتبه بالا نیز می‌توان کمک گرفت که این موضوع را توضیح دادیم.

نکته‌ی دیگر این است که در این زبان‌ها، بهینه سازی‌ها اکثرا توسط خود برنامه مدیریت می‌شود. اینگونه کاربر فقط تمرکزش را روی کاری که می‌خواهد انجام دهد می‌گذارد؛ نه این که چطور این کار را انجام دهد. همچنین وقتی همه‌ی این مراحل توسط کامپایلر انجام می‌شود، از پیچیدگی کار برنامه نویس کاسته شده و احتمال خطا نیز پایین‌تر می‌آید.

بسیاری از زبان‌های شی گرایی نیز به سمتی رفته‌اند که با مفاهیم برنامه نویسی تابعی سازگار شوند. برای مثال java8 یا C++11 عبارات لاندا را میخوانند. البته مفاهیم دیگری نیز در آن‌ها وجود دارد که نشان می‌دهد زبان‌های تابعی راه درستی را طی کرده‌اند. مثلا توابع محض، توابع مرتبه بالا مانند transform یا مفهوم متغیر ثابت [۱۳] در زبان C++ وجود دارد. حتی چیزی نزدیک به مفهوم ارزیابی تنبل یا کندرو را در محاسبات منطقی این زبان می‌توان مشاهده کرد.

هر چیزی که بتواند نیاز دوره‌ی خود را تامین کند، محبوبیت خواهد یافت. حداقل تا وقتی این نیاز تامین شود. اکنون نیز زبان‌های برنامه نویسی تابعی به نیازهای روز پاسخ می‌دهند، پس عجیب نیست اگر ببینیم که پروژه‌های متعددی از قابلیت‌های این زبان‌ها سود می‌برند.

---

[۱۳] const

فصل ۴

نتیجه‌گیری

زبان‌های برنامه نویسی تابعی بدون توقف در حال رشد هستند. در این حین، هنوز بسیاری معتقدند که این نوع نگاه به مسائل، بیش از حد پیچیده است. در این گزارش سعی کردیم به بررسی کلی این زبان‌ها و نحوه‌ی فکر کردن در این محیط بپردازیم و نیز دلایل اهمیت یافتن این نوع نگاه را پیدا کنیم.

فصل ۵

کاربرد

در این مقاله، نحوه‌ی فکر کردن در رده‌ی زبانی تابعی را یاد می‌گیرید. در نتیجه، پس از فراگیری مطالب آن قادر خواهید بود به بررسی تخصصی کامپایلرهای این زبان‌ها و روش‌های بهینه سازی با استفاده از دیدگاه تابعی بپردازید. همچنین می‌توان اساس کار زبان‌هایی چون coq یا HOL Light که نوعی ارائه دهنده‌ی اثبات خودکار هستند را بررسی نمود. در کل، فعالیت‌های بسیاری در این زمینه‌ها در حال انجام است. این مقاله نیز، می‌تواند پایه‌ای برای ورود تخصصی به هر یک از این مباحث باشد.

کتاب‌نامه

[۱] Jacques Garrigue, Computability and lambda calculus, 2013

[۲] Peter Selinger, Lecture Notes on the Lambda Calculus Department of Mathematics and Statistics Dalhousie University, Halifax, Canada

[۳] Raul Rojas, A Tutorial Introduction to the Lambda Calculus, FU Berlin, WS-97/98

[۴] J. HUGHES, Why Functional Programming Matters, University of Glasgow

[۵] Zhenjiang Hu, John Hughes, Meng Wang, How functional programming mattered, National Science Review, Volume 2, Issue 3, September 2015

Abstract

This paper consists of two parts. First part is a brief introduction to the lambda calculus, in which you will learn the fundamentals of functional programming. The second part however is more focused on why today's functional programming languages are getting popular and being used more in industry and education.

College of Science
School of Mathematics, Statistics, and Computer Science

An Introduction to the Functional Programming Paradigm

MohammadMehdi Ahmadi

Supervisor: Dr. Mojtaba Mojtahedi

A thesis submitted to Graduate Studies Office
in partial fulfillment of the requirements for the degree of
B.Sc. in
Computer Science

2021