به نام خدا

زبان‌های برنامه‌نویسی

آرش شفیعی

فهرست مطالب

مقدمه

نحو و معناشناسی

متغیرها و نوع‌های داده‌ای

برنامه نویسی تابعی

برنامه نویسی رویه‌ای

برنامه نویسی شیءگرا

برنامه نویسی همروند

کتاب‌های مرجع

- مفاهیم زبان‌های برنامه‌نویسی، از روبرت سبستا [1]
- مفاهیم در زبان‌های برنامه‌نویسی، از جان میچل [2]
- زبان‌های برنامه‌نویسی : اصول و الگوواره‌ها، از ماریتسیو گابریلی [3]
- مستندات زبان‌های برنامه‌نویسی

[1] Concepts of Programming Languages، by Robert W. Sebesta
[2] Concepts in Programming Languages، by John C. Mitchell
[3] Programming Languages : Priciples and Paradigms، by Maurizio Gabbrielli

مقدمه

زبان‌های برنامه‌نویسی

- یک زبان برنامه‌نویسی، یک سیستم نشانه‌گذاری [1] است برای بیان یک برنامهٔ کامپیوتری [2]. یک برنامهٔ کامپیوتری مجموعه‌ای از دستورات است برای انجام محاسبات بر روی داده‌ها.
- یک زبان برنامه‌نویسی ایده‌آل، به برنامه‌نویسان اجازه می‌دهد یک برنامه را به طور مختصر و واضح بیان کنند.
- از آنجایی که یک برنامهٔ کامپیوتری باید در طول دوران حیات خود توسط برنامه‌نویسان مختلف خوانده شود، فهمیده شود، و تغییر داده شود، بنابراین یک زبان برنامه‌نویسی خوب زبانی است که برنامه‌نویسان را قادر می‌سازد، برنامه‌های یکدیگر را بهتر بفهمند.
- معمولا یک برنامهٔ بزرگ از تعداد زیادی اجزا تشکیل شده است که با یکدیگر در ارتباط هستند. یک زبان برنامه‌نویسی خوب، زبانی است که به برنامه‌نویسان کمک می‌کند برنامه‌های پیچیده و ارتباط بین اجزا را به سادگی و به طور بهینه توصیف کنند.

[1] Notation system
[2] Computer program

زبان‌های برنامه‌نویسی

- در طراحی یک زبان برنامه‌نویسی معمولا باید انتخاب‌های مختلف را سبک‌سنگین [1] کرد.
- به طور مثال برخی از ویژگی‌های زبان‌های برنامه‌نویسی به برنامه‌نویسان کمک می‌کنند برنامه‌ها را سریع بنویسند، اما از طرفی همین ویژگی‌ها باعث کاهش بهره‌وری [2] و پیچیده‌تر شدن و کندتر شدن کامپایلر می‌شوند.
- برخی از ویژگی‌های زبانی باعث می‌شوند کامپایلر به طور بهینه پیاده‌سازی شود، اما از طرف دیگر باعث می‌شوند برنامه برای برنامه‌نویسان پیچیده‌تر شود.
- از آنجایی که برنامه‌های متفاوت نیازهای متفاوتی دارند، بنابراین زبان‌های برنامه‌نویسی متفاوتی به وجود آمده‌اند تا این نیازها را برآورده سازند. هر زبان برنامه‌نویسی موفقی در ابتدا برای یک مجموعه از برنامه‌ها با نیازهای مشابه به وجود آمده است. البته این بدین معنی نیست که یک زبان فقط برای انجام یک کار به وجود آمده است، بلکه بدین معنی است که تمرکز یک زبان بر روی حل کردن مجموعهٔ مشخصی از مسائل است.

[1] Trade-off
[2] Performance

زبان‌های برنامه‌نویسی

- مطالعهٔ زبان‌های برنامه‌نویسی به ما کمک خواهد کرد مفاهیم اصلی در زبان‌های برنامه‌نویسی را بهتر بفهمیم و با نقاط قوت و ضعف آنها بهتر آشنا شویم و بتوانیم از زبان‌ها حداکثر بهره را ببریم.
- هر زبان برنامه‌نویسی روشی متفاوت برای حل یک مسئله ارائه می‌کند. در واقع یک زبان برنامه‌نویسی یک چهارچوب فکری برای حل مسئله و طراحی نرم‌افزار ارائه می‌کند. بنابراین مطالعهٔ یک زبان و روش‌های موجود در آن زبان جهت حل یک مسئله، می‌تواند به حل آن مسئله در زبان‌های دیگر نیز کمک کند.

زبان‌های برنامه‌نویسی

- در اینجا ابتدا توضیح می‌دهیم چرا به مطالعه زبان‌های برنامه نویسی احتیاج داریم.
- سپس به طور خلاصه حوزه هایی را توضیح می‌دهیم که در آنها به زبان‌های برنامه نویسی نیاز داریم.
- توضیح می‌دهیم بر اساس چه معیارهایی می‌توان زبان‌های برنامه نویسی را مورد ارزیابی قرار داد.
- در پایان تاریخچهٔ زبان‌های برنامه نویسی و روش‌های پیاده‌سازی آنها را به اختصار شرح می‌دهیم.

دلایل مطالعه زبان‌های برنامه نویسی

- **افزایش توانایی بیان ایده‌ها :** معمولا عمق تفکر افراد وابسته به توان بیان آنها در زبانی است که با استفاده از آن ارتباط برقرار می‌کنند. از آنجایی که زبان ابزاری است برای تفکر و اندیشیدن، هر چقدر یک زبان را بهتر بشناسیم، بهتر می‌توانیم با استفاده از آن ایده‌ها را منتقل و مسائل را حل کنیم. معمولا یک زبان ساختار معینی دارد و با استفاده از آن ساختار می‌توان ارتباط برقرار کرد ولی هر ساختاری محدودیت‌هایی نیز دارد. بنابراین یادگیری یک زبان متفاوت که ساختار متفاوتی دارد کمک می‌کند بتوانیم بهتر بیاندیشیم.

دلایل مطالعه زبان‌های برنامه نویسی

- برنامه نویسان نیز در طراحی نرم‌افزار محدود به زبانی هستند که با استفاده از آن برنامه می‌نویسند. معمولا در برنامه نویسی ساختارهای کنترلی و ساختارهای داده موجود بر روش تفکر ما در حل یک مسئله محدودیت اعمال می‌کنند. یادگیری زبان‌های متفاوت کمک می‌کند که ساختارهای متفاوتی در ذهن ما به وجود بیایند و بتوانیم برای حل مسئله از آن ساختار ها استفاده کنیم.
- برای مثال یک برنامه نویس سی که با پایتون آشنا باشد و از سهولت استفاده از لیست‌ها آگاه باشد، می‌تواند لیست‌هایی مشابه با پایتون در برنامه سی ایجاد کند و از آنها برای حل مسئله خود استفاده کند.

دلایل مطالعه زبان‌های برنامه نویسی

- **انتخاب مناسب یک زبان برای یک کاربرد خاص**: معمولا یک زبان برای کاربردی خاص به وجود می‌آید و ممکن است کسی که در یک حوزه فعالیت می‌کند و زبانی را برای کاربردی خاص استفاده می‌کند، از زبان‌هایی که برای کاربردهای دیگر وجود دارند بی اطلاع باشد. مطالعه زبان‌های برنامه‌نویسی کمک می‌کند نقاط قوت و ضعف و کاربرد زبان‌ها را به خوبی بشناسیم و هر زبان را مناسب با حوزه مرتبط با آن به کار ببریم.

- **افزایش توانایی در یادگیری زبان‌های جدید :** علوم کامپیوتر و تکنولوژی‌های مرتبط با آن همواره در حال پیشرفت هستند و زبان‌های جدید در حال ابداع شدن هستند. یادگیری مفاهیم اصلی در طراحی زبان‌های برنامه نویسی به ما کمک می‌کند تا بتوانیم زبان‌های جدید را بهتر یاد بگیریم. برای مثال وقتی با مفهوم برنامه نویسی شیء گرا آشنا شدیم می‌توانیم از شیء گرایی در همه‌ی زبان‌های شیء گرا استفاده کنیم. این اصل در یادگیری زبان‌های طبیعی نیز صادق است. هرچه با گرامرهای بیشتری در زبان‌های متفاوت آشنا باشیم یادگیری یک زبان طبیعی جدید برای ما آسان‌تر می‌شود.

دلایل مطالعه زبان‌های برنامه نویسی

- **استفاده بهتر از یک زبان:** وقتی با نحوه پیاده سازی یک زبان آشنا شویم، می‌توانیم از آن به طور بهینه‌تر استفاده کنیم. به طور مثال وقتی بفهمیم در زبان‌های شیء گرا، وراثت چه سر باری تحمیل می‌کند متوجه می‌شویم چه جاهایی بهتر است از وراثت استفاده کنیم و چه جاهایی از آن استفاده نکنیم. به عنوان مثالی دیگر، برنامه نویسی که از سربار و پیچیدگی فراخوانی توابع آگاه نباشد، ممکن است از توابع به کثرت استفاده کند که این امر باعث کاهش بهره‌وری برنامه می‌شود. بنابراین با یادگیری مفاهیم پایه در طراحی زبان‌ها می‌توانیم برنامه نویس بهتری شویم. همچنین یادگیری مفاهیم برنامه نویسی به یک برنامه نویس کمک می کند تا از قابلیت‌هایی که تاکنون استفاده نکرده است، استفاده کند.
- **ابداع زبان‌های جدید:** در نهایت با مطالعه مفاهیم زبان‌های برنامه نویسی می‌توانیم دیدگاه روشن‌تری نسبت به گستره‌ی زبان‌های برنامه نویسی به دست آوریم و بنابراین می‌توانیم زبان‌های جدیدی را به فراخور نیازهای خود ابداع و پیاده سازی کنیم.

حوزه‌های برنامه نویسی

- کامپیوترها و برنامه‌های کامپیوتر در حوزه‌های مختلفی استفاده می‌شوند. از سیستم‌های تعبیه شده در هواپیما گرفته تا سیستم های محاسباتی پیچیده در آزمایشگاه‌های تحقیقاتی تا بازی‌های کامپیوتری در کامپیوترهای خانگی یا تلفن‌های همراه.
- **کاربردهای علمی :** اولین کامپیوترهای دیجیتال در دهۀ ۱۹۴۰ و ۱۹۵۰ برای انجام محاسبات علمی و استفاده در آزمایشگاه‌های تحقیقاتی به وجود آمدند. در آن زمان برنامه‌های پیچیده و ساختارداده‌های پیچیده وجود نداشتند، اما به محاسبات پیچیده نیاز بود. معمولا برای انجام چنین محاسباتی به آرایه و ماتریس و یک حلقه برای شمارش نیاز بود. بنابراین زبان‌هایی که در آن زمان ابداع شدند، در جهت رفع نیازها طراحی شده بودند. قبل از آن دوره از زبان اسمبلی استفاده می شد. اولین زبانی که برای محاسبه فرمول‌های ریاضی به وجود آمد زبان فورترن [1] بود. در آن دوره زبان الگول [2] نیز به وجود آمد. اما زبان فورترن به نحوی برای آن کاربرد خاص طراحی شده بود که هنوز هم مورد استفاده است. زبان‌های آن زمان به دلیل محدودیت سخت‌افزار به راندمان قابل توجهی نیاز داشتند.

[1] Fortran
[2] Algol

حوزه‌های برنامه نویسی

- **کاربردهای تجاری**: استفاده از کامپیوترها برای کاربردهای تجاری در ابتدای دهه‌ی ۱۹۵۰ آغاز شد. اولین زبانی که در این حوزه به وجود آمد زبان کوبول [1] بود. در کاربردهای تجاری نیاز به سهولت در گزارش‌گیری و همچنین انجام دقیق محاسبات تجاری است. توسعه برنامه‌های کاربردی برای انجام محاسبات در بانک‌ها یکی از موارد کاربردهای تجاری است.

- **هوش مصنوعی**: یکی از کاربردهای مهم در محاسبات کامپیوتر در حوزه هوش مصنوعی است. در این حوزه از ماتریس‌ها و آرایه‌ها و انجام محاسبات بر روی این ساختارها به کثرت استفاده می‌شود. اولین زبانی که در این حوزه به وجود آمد زبان لیسپ [2] بود که یک زبان تابعی است و در سال ۱۹۵۹ ابداع شد. در دهۀ ۱۹۷۰ زبان دیگری برای کاربردهای هوش مصنوعی به نام زبان پرولوگ [3] به وجود آمد. در دهۀ اخیر بسیاری از برنامه‌های هوش مصنوعی با استفاده از پایتون [4] نوشته می‌شوند.

---

[1] Cobol
[2] Lisp
[3] Prolog
[4] Python

حوزه‌های برنامه نویسی

- **نرم‌افزارهای وب:** صفحه‌های سادهٔ وب در بدو ابداع آن با استفاده از زبان اچ‌تی‌ام‌ال [1] طراحی می‌شدند. با پیشرفت تکنولوژی وب و نیاز به صفحه های پویا (صفحه‌هایی که محتوای آن قابل تغییر است)، نیاز به زبان‌های دیگری شد که می‌توان در این دسته به جاوااسکریپت [2] یا پی‌اچ‌پی [3] اشاره کرد.

[1] HTML
[2] JavaScript
[3] PHP

ارزیابی زبان‌ها

- برای ارزیابی زبان‌های برنامه نویسی تعدادی معیار را در نظر می‌گیریم.
- **خوانایی :** یکی از مهم‌ترین معیارها برای ارزیابی زبان‌های برنامه نویسی، خوانایی است، یعنی سهولت خواندن و فهمیدن برنامه‌ای که توسط برنامه نویسان دیگر نوشته شده است. قبل از ۱۹۷۰ به علت محدودیت های سخت‌افزار، آنچه در یک برنامه بیشتر اهمیت داشت راندمان یک برنامه بود. به تدریج برنامه های پیچیده‌تری به وجود آمدند و همچنین سخت‌افزارهای قوی‌تر ابداع شد، بنابراین به مرور زمان آنچه اهمیت بیشتری می‌یافت، خوانایی برنامه‌ها بود. همچنین نگهداری برنامه‌ها با به وجود آمدن برنامه‌های پیچیده و بزرگ اهمیت بیشتری پیدا می‌کرد، بنابراین لازم بود برنامه ها به میزان کافی خوانا باشند. پس زبان‌های برنامه نویسی که تا آن زمان به زبان ماشین شباهت بیشتری داشتند، به مرور زمان به زبان انسان شباهت بیشتری پیدا کردند. خوانایی را می‌توان از جنبه‌های مختلفی بررسی کرد : (۱) سادگی، (۲) تعامد [1] ، (۳) نوع‌های داده‌ای، (۴) طراحی نحوی [2] .

[1] Orthogonality
[2] Syntax design

ارزیابی زبان‌ها

- **(۱) سادگی**: سادگی یک زبان برنامه نویسی، بر خوانایی آن تأثیر مستقیم دارد. زبانی که تعداد زیادی ساختار کنترلی وکلمات کلیدی دارد طبیعتاً برای یادگیری سخت‌تر است. وقتی یک زبان، بسیار پیچیده می‌شود، معمولا برنامه نویسان تنها از قسمتی از زبان استفاده می‌کنند.
- این سادگی باید در حد متوسط باشد تا برنامه بهترین خوانایی را داشته باشد. اگر یک برنامه بیش از حد ساده باشد خوانایی آن کاهش می‌یابد. برای مثال، زبان اسمبلی زبان بسیار ساده‌ای است اما خواندن یک برنامه در زبان اسمبلی نسبت به زبان سی سخت تر است.

ارزیابی زبان‌ها

- **(۲) تعامد:** تعامد در یک زبان برنامه نویسی بدین معناست که مجموعه نسبتاً کوچکی از ساختارهای ابتدایی در یک زبان بتوانند به چندین روش محدود با یکدیگر ترکیب شوند و همه ی ساختارهای داده و کنترلی مورد نیاز را بسازند. اگر همهٔ ساختارهای ابتدایی بتوانند با یکدیگر ترکیب شوند و ترکیب‌های معناداری بسازند، می‌گوییم یک زبان تعامد بالایی دارد. تعامد کم در یک زبان بدین معناست که برخی ترکیب‌ها معنادار نیستند. بنابراین یک زبان با تعامد کم استثناهای بیشتری دارد. تعداد استثناهای زیاد در یک زبان یادگیری زبان را پیچیده تر می‌کند.

- برای مثال در کامپیوترهای IBM دو دستور برای جمع وجود داشت. دستور `A Reg, Memory` محتوای یک رجیستر و یک خانه حافظه را جمع می‌کند و دستور `AR Reg1, Reg2` محتوای دو رجیستر را جمع می‌کند. بنابر این برخی از ترکیب‌ها در این زبان بی‌معنا هستند. مثلا توسط دستور `A` نمی‌توان دو رجیستر را با هم جمع کرد که این یک استثنا در زبان است. اما در کامپیوترهای سری VAX تنها یک دستور `ADDL op1, op2` برای جمع طراحی شده بود. بنابراین رجیسترها و حافظه‌ها به شکل‌های مختلف می‌توانند با استفاده از این دستور با هم جمع شوند و در نتیجه زبان این نوع کامپیوترها تعامد بیشتری دارد.

ارزیابی زبان‌ها

- اگر تعامد کم باشد، تعداد استثناها افزایش می‌یابد و در نتیجه نوشتن و خواندن برنامه سخت می‌شود. از طرفی دیگر اگر تعامد زیاد باشد، ترکیب‌های پیچیده‌ای توسط زبان به وجود می‌آیند و باز هم خوانایی برنامه کاهش می‌یابد.
- در زبان سی برای مثال یک ساختمان (استراکت) را می‌توانیم توسط یک تابع بازگردانیم ولی یک آرایه را نمی‌توانیم بازگردانیم. اعضای یک ساختمان میتوانند از هر نوعی باشند غیر از `void` . مقادیر به توابع با مقدار داده می‌شوند اما اگر آرایه به تابع ارسال کنیم، با ارجاع به تابع داده می‌شود. این استثناها از تأثیرات تعامد کم است که زبان را برای یادگیری و استفاده پیچیده تر می‌کند.
- از طرفی اگر تعامد خیلی زیاد باشد نیز زبان پیچیده می‌شود. مثلا در زبان الگول نوع‌ها می‌توانند با هم ترکیب شوند و نوع‌های بسیار پیچیده‌ای بوجود آورند. در این حالت هیچ استثنایی وجود ندارد و بنابراین تعامد بسیار بالا است ولی همین امر خوانایی برنامه را کاهش می‌دهد.

ارزیابی زبان‌ها

- **(۳) نوع داده:** وجود امکانات کافی برای نوع داده‌ها و ساختمان داده‌ها در یک زبان به خوانایی آن زبان کمک می‌کند. برای مثال در زبانی که نوع بولی وجود ندارد، مجبوریم برای تعریف مقادیر درست و نادرست از اعداد صفر و یک استفاده کنیم که این امر موجب کاهش خوانایی کد می‌شود.

- **(۴) طراحی قوائد نحوی:** قوائد نحوی در یک زبان تأثیر مهمی بر روی خوانایی برنامه در آن زبان دارد. برای مثال در زبان فورترن برای اتمام یک بلوک از کد از یک کلمهٔ کلیدی متناسب با آن بلوک استفاده می‌شود و بدین ترتیب خوانایی کد در آن نسبت به زبان سی که همهٔ بلوک‌ها در آن با آکولاد پایان می‌یابند بالاتر است. مثلا برای خاتمه بلوک `IF` در فورترن از `END IF` استفاده می‌شود. در اینجا می‌بینیم که برای بالا بردن خوانایی در یک زبان می‌توان کلمات کلیدی را افزایش داد ولی از طرفی افزایش کلمات کلیدی، زبان را پیچیده تر و برای یادگیری سخت تر می‌کند، بنابراین رعایت حد اعتدال در اینجا نیز بسیار دارای اهمیت است.

ارزیابی زبان‌ها

- همچنین برای خوانایی بهتر، قوائد نحوی باید به گونه‌ای طراحی شوند که صورت و معنی یک عبارت همخوانی داشته باشند. برای مثال در زبان سی اگر متغیری درون یک تابع با استفاده از کلمهٔ کلیدی `static` تعریف شود، آن متغیر در اولین اجرای تابع بر روی حافظه در بخش داده تعریف می‌شود و مقدار خود را در فراخوانی‌های پی در پی حفظ می‌کند. اما اگر متغیری به صورت عمومی با کلمهٔ `static` تعریف شود، آن متغیر فقط در آن فایل قابل استفاده است. پس این کلمه کلیدی معانی متعددی دارد که این امر از خوانایی برنامه می‌کاهد ولی از طرف دیگر تعداد کلمات کلیدی کم باعث سادگی زبان می‌شود.

ارزیابی زبان‌ها

- **قابلیت اطمینان** [1]: یک برنامه قابل اطمینان است اگر همیشه همان کاری را انجام دهد که برنامه‌نویس انتظار دارد. به عبارت دیگر برای افزایش قابلیت اطمینان خطاهای برنامه نویس باید توسط کامپایلر تشخیص داده شود.
- از جمله عواملی که بر قابلیت اطمینان تأثیر می‌گذارند، عبارتند از (۱) بررسی نوع [2]، (۲) مدیریت استثنا [3]، (۳) نام‌های مستعار [4]، و (۴) خوانایی

[1] Reliability
[2] Type checking
[3] Exception handling
[4] Aliasing

ارزیابی زبان‌ها

- **(۱) بررسی نوع**: بررسی نوع بدین معناست که خطاهای نوع داده‌ای در یک برنامه توسط کامپایلر یا در هنگام اجرا تشخیص داده شوند. بررسی نوع در زمان اجرا هزینه بالایی دارد بنابراین در زبانی که به سرعت اجرای زیاد نیاز دارد، بررسی نوع توسط کامپایلر انجام می‌شود. همچنین اگر خطاها در زمان کامپایل شناخته شوند، می‌توان آنها را زودتر رفع کرد و نیازی به اجرای برنامه برای رفع خطا نیست. برای مثال در نسخه‌های اولیه زبان سی، هنگام ارسال یک متغیر به تابع، لزومی نداشت نوع آن متغیر با نوع داده‌ای که تابع دریافت می‌کند همخوانی داشته باشد. این امر موجب خطاهای احتمالی توسط برنامه نویس می‌شد و قابلیت اطمینان زبان را پایین می‌آورد.

ارزیابی زبان‌ها

- **(۲) مدیریت استثنا:** توانایی یک برنامه برای تشخیص خطاها در زمان اجرا و ادامه برنامه در صورت بروز خطا قابلیت اطمینان برنامه را بالا می‌برد. به این قابلیت مدیریت استثنا گفته می‌شود. برای مثال وجود مدیریت استثنا در زبان سی++ این زبان را نسبت به سی قابل اطمینان‌تر می‌کند.
- **(۳) نام‌های مستعار:** با استفاده از قابلیت ایجاد نام‌های مستعار، یک خانه حافظه می‌تواند دو یا چند نام داشته باشد. برای مثال اشاره‌گرها و متغیرهای مرجع در سی++ می‌توانند به یک متغیر چندین نام را منتسب کنند. وجود چنین قابلیتی امکان بروز خطا را بیشتر و در نتیجه قابلیت اطمینان را کمتر می‌کند.
- **(۴) خوانایی:** هرچه خوانایی یک برنامه بیشتر باشد، برنامه نویس راحت‌تر می‌تواند برنامه‌هایی بنویسد که درست‌تر و در نتیجه قابل اطمینان‌تر باشند.

ارزیابی زبان‌ها

- **هزینه:** انگیزه ابتدایی طراحی زبان‌های برنامه نویسی پایین آوردن هزینهٔ نوشتن آنها بود. برنامه نویسان می‌توانستند با استفاده از یک زبان برنامه نویسی سطح بالا یک برنامه را در زمان کمتری بنویسند. حال هر چه یک زبان ساده‌تر باشد یادگیری آن نیز نیازمند زمان کمتری است و نوشتن برنامه توسط آن ساده‌تر است.
- هزینهٔ زمانی اجرای یک برنامه نیز یکی از معیارهای ارزیابی یک زبان است. هرچه یک زبان در زمان کمتری یک برنامهٔ معین را اجرا کند، آن برنامه بهتر است. اما گاهی این معیار با معیارهای دیگر در تضاد است. برای مثال اشاره‌گرها گرچه قابلیت اطمینان را پایین می‌آورند، اما از هزینه اجرا نیز می‌کاهند.

ارزیابی زبان‌ها

- گاهی (برای مثال در یک برنامهٔ حسابداری) خوانایی یک برنامه برای ما مهم‌تر است پس زبانی را انتخاب می‌کنیم که خواناتر باشد و هزینه اجرا برای ما در اولویت نیست. اما گاهی (برای مثال در سیستم‌های تعبیه شده در هواپیما) زمان اجرا بسیار پر اهمیت است و خوانایی اهمیت زیادی ندارد.
- هزینه می‌تواند با قابلیت اطمینان به گونه‌ای دیگر نیز در ارتباط باشد. قابلیت اطمینان پایین ممکن است باعث بروز خطا شود و در یک سیستم حساس مانند هواپیما، ممکن است باعث تحمیل هزینه‌های زیادی شود.
- هزینه نگهداری یک نرم‌افزار نیز معیار مهمی است. هرچه خوانایی یک برنامه بالاتر باشد، هزینه نگهداری آن پایین‌تر است چرا که برنامه نویسان آتی می‌توانند برنامه را به آسانی فرا بگیرند و تغییر دهند.

ارزیابی زبان‌ها

- قدرت بیان در مقابل راندمان [1]: در بسیاری مواقع یک زبان برنامه‌نویسی قسمتی از وظیفهٔ برنامه‌نویس را به طور خودکار انجام می‌دهد. به طور مثال یک برنامه‌نویس، وظیفه دارد حافظه را مدیریت کند، اما برخی از زبان‌های برنامه‌نویسی مدیریت حافظه را به طور اتوماتیک و خودکار انجام می‌دهند و حافظه‌های تخصیص داده شده را وقتی به آنها دیگر نیازی نیست به طور خودکار آزادسازی می‌کنند. گرچه چنین خودکارسازی‌هایی باعث می‌شود برنامه‌نویس نیازی به فکر کردن نداشته باشد، اما از طرف دیگر باعث می‌شود از سرعت اجرای برنامه‌ها در یک زبان برنامه‌نویسی کاسته شود.
- هر چه قدرت بیان یک زبان بیشتر باشد (کارهای بیشتری توسط زبان به طور خودکار انجام شوند)، برنامه‌نویس نیاز به زمان کمتری برای نوشتن برنامه دارد، اما برنامه با سرعت کمتری اجرا می‌شود و راندمان کمتری دارد. برخی مواقع نیاز است برنامه قدرت بیان بیشتری داشته باشد تا پیچیدگی برنامه کمتر شود و خطاهای برنامه کاهش یابد و برخی مواقع نیاز است که یک برنامه راندمان بیشتری داشته باشد.

[1] Expressiveness versus efficiency

ارزیابی زبان‌ها

- **معیارهای دیگر**: یکی از معیارهای مقایسه زبان‌ها می‌تواند قابلیت اجرای برنامه‌های آن بر روی سیستم‌ها و معماری های سخت‌افزاری مختلف با قابلیت جابجایی [1] باشد. هرچه یک زبان دارای استانداردهای بهتری باشد، کامپایلرهای متنوع در سیستم عامل‌های مختلف همگون‌تر پیاده سازی می‌شوند و بنابراین یک کد واحد را می‌توان بر روی سیستم‌های مختلف کامپایل و اجرا کرد. به عنوان یک معیار دیگر می‌توانیم عمومی بودن یا اختصاصی بودن یک زبان را در نظر بگیریم. برخی از زبان‌ها برای یک استفاده خاص به کار می‌روند و برخی از زبان‌ها در کاربردهای متنوع‌تری می‌توانند استفاده شوند.

- در طراحی زبان‌های برنامه نویسی معمولا معیارهای زیادی وجود دارد که این معیارها با هم در تضادند و بدست آوردن حد وسط مناسب برای یک کاربرد خاص بسیار حائز اهمیت است. برای مثال در زبان جاوا بررسی می‌شود که دسترسی به اندیس‌های آرایه در محدوده تعریف شدهٔ آرایه باشد. بدین ترتیب قابلیت اطمینان جاوا از سی بیشتر است ولی این بررسی اجرای برنامه جاوا را کندتر می‌کند، پس هزینه اجرا افزایش پیدا می‌کند.

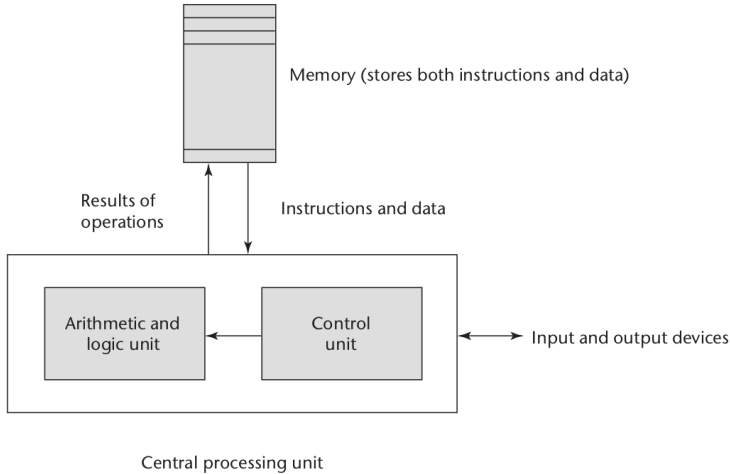
[1] Portability

عوامل تأثیرگذار بر طراحی زبان

- دو عامل مهم تأثیرگذار بر طراحی زبان‌های برنامه‌نویسی عبارتند از معماری کامپیوتر و متودهای طراحی.
- **معماری کامپیوتر**: معماری سخت‌افزار عامل تأثیرگذار مهمی در طراحی زبان‌های برنامه‌نویسی است. همهٔ زبان‌های ۶۰ سال گذشته تحت تأثیر معماری وان نویمان [1] بوده‌اند. در معماری وان نویمان کد برنامه و داده‌ها هر دو بر روی حافظهٔ اصلی قرار می‌گیرند. سپس واحد پردازنده مرکزی، دستورات برنامه را یک به یک از حافظه می‌خواند و اجرا می‌کند. نتیجه محاسبات دوباره بر روی حافظه قرار می‌گیرد. همه کامپیوترها از دهه ۱۹۴۰ تا کنون بر اساس این معماری ساخته شده‌اند که این معماری نیز بر اساس ماشین تورینگ طراحی شده و ماشین تورینگ نیز از موتور تحلیلی چارلز بابج الهام گرفته است. بنابراین همه زبان‌ها نیز طبیعتاً تحت تأثیر این معماری بوده‌اند.

[1] Van Neumann

عوامل تأثیرگذار بر طراحی زبان



شکل: معماری وان نویمان

عوامل تأثیرگذار بر طراحی زبان

- به دلیل استفاده از این معماری اکثر زبان‌های برنامه نویسی از متغیرها استفاده می‌کنند که معرف خانه‌های حافظه است، از عملیات انتساب استفاده می‌کنند که همان عملیات تغییر مقادیر توسط پردازنده است و از حلقه‌ها استفاده می‌کنند که ساده‌ترین روش برای تکرار دسته‌ای از دستورات است.

عوامل تأثیرگذار بر طراحی زبان

- اجرای یک برنامه بر روی این معماری بدین شکل است که پردازنده دستورات را از حافظه می‌خواند و یک به یک اجرا می‌کند. آدرس آخرین دستور برای اجرا در یک رجیستر به نام رجیستر شمارنده برنامه [1] ذخیره می‌شود.
- می‌توان الگوریتم اجرای برنامه را به صورت زیر نوشت :

```
initialize the program counter
repeat forever
    fetch the instruction pointed to by the program counter
    increment the program counter to point at the next instruction
    decode the instruction
    execute the instruction
end repeat
```

[1] program counter

عوامل تأثیرگذار بر طراحی زبان

- برخی از زبان‌های برنامه نویسی عملیات انتساب متغیر ندارند. این زبان‌ها که زبان‌های تابعی نامیده می‌شوند خروجی را براساس اعمال تعدادی توابع بر ورودی محاسبه می‌کنند. این زبان‌ها به طور مستقیم از معماری وان نویمان پیروی نمی‌کنند، اما کامپایلری که برای آنها نوشته می‌شود باید نهایتاً برنامه را به زبان ماشین که براساس معماری وان نویمان است ترجمه کند.
- **متودهای طراحی**: در اوایل دهۀ ۱۹۷۰ نرم‌افزارها برای برنامه‌های بزرگتری به کار می‌رفتند و بنابراین نیاز به سازماندهی بهتر نرم‌افزارها بود. این امر سبب شد زبان‌هایی به وجود بیایند که تمرکزشان بر روی ساخت نوع داده‌ها است. اولین زبانی که برای حل آن مشکلات به وجود آمد زبان سیمولا [1] بود که شروعی برای طراحی زبان‌های شیء گرا بود. پس از آن زبان‌های اسمالتاک [2]، سی++ و جاوا به وجود آمدند.

[1] Simula
[2] Smalltalk

تاریخچهٔ زبان‌های برنامه‌نویسی

- در طول نیم قرن اخیر صدها زبان برنامه‌نویسی طراحی و پیاده‌سازی شده‌اند. بسیاری از زبان‌های برنامه‌نویسی از مفاهیم مشابهی استفاده می‌کنند، بنابراین در اینجا بر روی تعدادی از آنها تمرکز می‌کنیم که مفاهیم متفاوتی را ارائه می‌کنند.
- هر زبان برنامه‌نویسی بر اساس تعدادی الگوواره [1] ساخته شده است.
- یک الگوواره یا پارادایم به طور کلی یک چهارچوب فکری است که مجموعه ای از نظریه ها را تشکیل داده است. به عبارت دیگر مجموعه ای از مفروضات و مفاهیم و ارزش‌هاست که الگوهای مشابه را می سازند.

[1] paradigm

تاریخچهٔ زبان‌های برنامه‌نویسی

- دو دسته مهم از روش‌های برنامه‌نویسی که بر اساس دو الگوواره به وجود آمده‌اند، عبارتند از: برنامه‌نویسی دستوری [2]، و برنامه‌نویسی اعلانی [3].
- در برنامه‌نویسی دستوری، مراحل اجرای یک برنامه گام به گام توسط برنامه‌نویس بیان می‌شود، در حالی که در برنامه‌نویسی اعلانی تنها هدف انجام محاسبات بدون شرح چگونگی انجام آن توصیف می‌شود.

[2] imperative programming
[3] declarative programming

تاریخچهٔ زبان‌های برنامه‌نویسی

- برای مثال در زبان پایتون [1]، به روش برنامه‌نویسی دستوری برای محاسبهٔ عدد فیبوناچی n ام برنامه‌ای به صورت زیر می‌نویسیم:

```
1  def fib(n) :
2      i,j = 0,1
3      for k in range(1,n + 1):
4          i,j = j, i + j
5      return j
```

- همین برنامه در زبان هسکل [2]، به روش برنامه‌نویسی اعلانی به صورت زیر نوشته می‌شود:

```
1  fib 0 = 0
2  fib 1 = 1
3  fib n = fib (n-1) + fib (n-2)
```

[1] Python
[2] Haskell

تاریخچهٔ زبان‌های برنامه‌نویسی

- دو پارادایم (الگووارهٔ) مهم را می‌توان زیرمجموعهٔ پارادایم برنامه‌نویسی دستوری، به حساب آورد: برنامه‌نویسی رویه‌ای [1] و برنامه‌نویسی شیءگرا [2].

[1] Procedural programming
[2] Object-oriented programming

تاریخچهٔ زبان‌های برنامه‌نویسی

- در برنامه‌نویسی رویه‌ای، یک برنامه از تعدادی رویه تشکیل شده است. هر رویه مقادیری را به عنوان ورودی می‌گیرد و پس از انجام محاسبات مقادیری را باز می‌گرداند. یک رویه می‌تواند در محاسبات خود از تعدادی متغیر عمومی استفاده کند. زبان‌های مهم در این دسته عبارتند از: فورترن [1]، الگول [2]، کوبول [3] و سی.
- در برنامه‌نویسی شیءگرا، یک برنامه از تعدادی از اشیاء که با یکدیگر در ارتباط هستند تشکیل شده است. هر شیء نمونه‌ای از یک کلاس است و یک کلاس ویژگی‌ها و رفتارهای معینی دارد. زبان‌های مهم در این دسته عبارتند از: سیمولا [4]. اسمالتاک [5]، سی++، و جاوا.

[1] Fortran
[2] Algol
[3] Cobol
[4] Simula
[5] Smalltalk

تاریخچهٔ زبان‌های برنامه‌نویسی

- دو پارادایم مهم را می‌توان زیرمجموعهٔ پارادایم برنامه‌نویسی اعلانی، به حساب آورد: برنامه‌نویسی تابعی [1] و برنامه‌نویسی منطقی [2].

[1] Functional programming
[2] Logic programming

تاریخچهٔ زبان‌های برنامه‌نویسی

- در برنامه‌نویسی تابعی، یک برنامه از تعدادی تابع تشکیل شده است. هر تابع مقادیری را به عنوان ورودی می‌گیرد و مقادیری را باز می‌گرداند. یک تابع به ازای یک ورودی معین همیشه خروجی ثابتی را بازمی‌گرداند. زبان‌های مهم در این دسته عبارتند از: لیسپ [1]، ام‌ال [2]، اوکمل [3]، هسکل [4].
- در برنامه‌نویسی منطقی، یک برنامه از عبارات منطقی تشکیل شده است. زبان پرولوگ [5] در این دسته قرار دارد.

[1] Lisp
[2] ML
[3] Ocaml
[4] Haskell
[5] Prolog

تاریخچهٔ زبان‌های برنامه‌نویسی

- یک پارادایم مهم دیگر در زبان‌های برنامه‌نویسی برنامه‌نویسی همروند [1] و توزیع‌شده [2] است. در برنامه‌نویسی همروند و توزیع‌شده محاسبات به صورت موازی توسط تعدادی پردازنده انجام می‌شود. زبان‌های مهم در این دسته عبارتند از: گو [1]، و ارلنگ [2].

[1] Concurrent programming
[2] Distributed programming
[1] Go
[2] Erlang

تاریخچهٔ زبان‌های برنامه‌نویسی

- برخی از زبان‌ها تعدادی از پارادایم‌ها را پشتبانی می‌کنند. برای مثال زبان پایتون هم یک زبان رویه‌ای است، هم شیءگرا، و هم تابعی. همچنین کتابخانه‌ای برای برنامه‌نویسی همروند ارائه می‌کند، بنابراین می‌توان برای برنامه‌نویسی همروند هم از آن استفاده کرد.
- به طور مشابه جاوا و سی++ گرچه در گروه زبان‌های شیءگرا قرار می‌گیرند، اما کتابخانه‌هایی را ارائه می‌کنند که با استفاده از آنها می‌توان به صورت تابعی و همروند نیز استفاده کرد.
- برخی از زبان‌ها تنها یک پارادایم را پشتیبانی می‌کنند. برای مثال هسکل فقط برای برنامه‌نویسی تابعی به کار می‌رود.

تاریخچهٔ زبان‌های برنامه‌نویسی

- در دههٔ ۱۹۵۰ تعدادی از زبان‌های برنامه‌نویسی برای تسهیل نوشتن دستورات کامپیوتری که تا قبل از آن به زبان اسمبلی نوشته می‌شدند به وجود آمدند. قبل از به وجود آمدن زبان‌های برنامه‌نویسی، به ازای هر نوع معماری سخت‌افزاری یک زبان اسمبلی وجود داشت. با این که زبان اسمبلی زبانی است که به زبان ماشین شباهت بیشتری دارد تا زبان انسان، و بنابراین نوشتن برنامه با استفاده از این زبان نسبتا دشوار است اما به علت کنترلی که برنامه‌نویس بر روی سخت‌افزار دارد، با استفاده از آن برنامه‌های کارآمدی می‌توان نوشت.
- اولین زبان برنامه‌نویسی در یک رسالهٔ دکتری در سال ۱۹۵۱ توسط کورادو بوهم [1] در دانشگاه ای‌تی‌اچ زوریخ توصیف و به همراه یک کامپایلر عرضه شد.
- دو زبان مهم تجاری که در این دهه به وجود آمدند، عبارتند از فورترن و کوبول.
- فورترن در بین سال‌های ۱۹۵۴ و ۱۹۵۶ توسط تیمی به رهبری جان باکوس [2] در آی‌بی‌ام به وجود آمد.

[1] Corrado Bohm
[2] John Backus

تاریخچهٔ زبان‌های برنامه‌نویسی

- نوآوری جدید فورترن این بود که به برنامه نویس کمک می‌کرد تا بتواند فرمول‌های ریاضی را به همان صورتی که بر روی کاغذ نوشته می‌شود بنویسد. در واقع کلمهٔ فورترن مخفف کلمهٔ ترجمه فرمول [1] بود. برای مثال برنامه‌نویسان فورترن می‌توانستند فرمولی مانند $i + 2 * j$ را بنویسند. تا قبل از آن نیاز بود که برنامه نویس متغیر i را در یک رجیستر ذخیره کند و j را در یک رجیستر دیگر. سپس j را دو برابر کند و سپس مقدار این دو رجیستر را با هم جمع کند. بنابراین فورترن به برنامه‌نویسان کمک می‌کرد که فرمول‌های ریاضی را به زبان خود بنویسند و نه به زبان کامپیوتر و کامپایلر عملیات مورد نیاز برای تبدیل فرمول به زبان اسمبلی را انجام می‌داد.

[1] Formula Translation

تاریخچهٔ زبان‌های برنامه‌نویسی

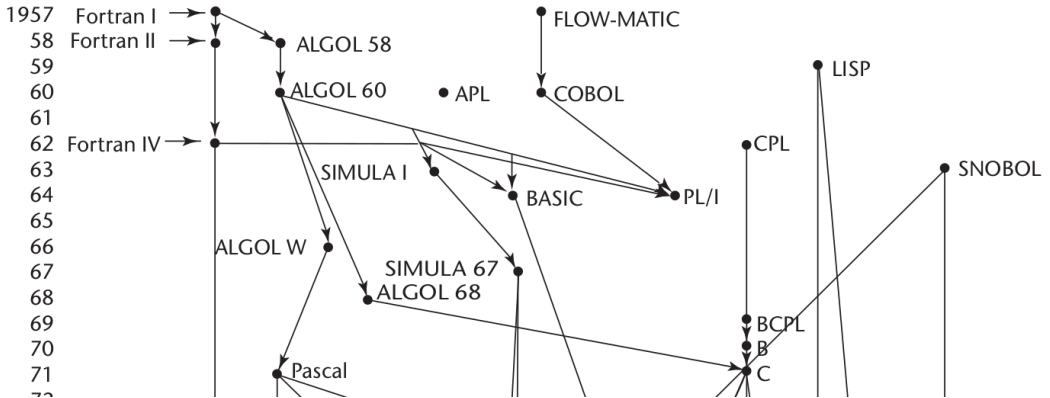
- فورترن همچنین دارای زیربرنامه و همچنین آرایه بود و بدین ترتیب برنامه‌نویسان می‌توانستند برنامه‌های قابل فهم‌تری بنویسند. البته فورترن دارای محدودیت‌هایی نیز بود. به طور مثال با استفاده از فورترن یک تابع نمی‌توانست خود را فراخوانی کند، زیرا برای این کار به تکنیک‌هایی نیاز بود که تا آن زمان به وجود نیامده بودند.
- در همان دوره زبان کوبول نیز برای استفاده در برنامه‌های تجاری توسط گریس هاپر [1] به وجود آمد. دستورات کوبول به زبان انسان شباهت زیادی داشت.

[1] Grace M. Hopper

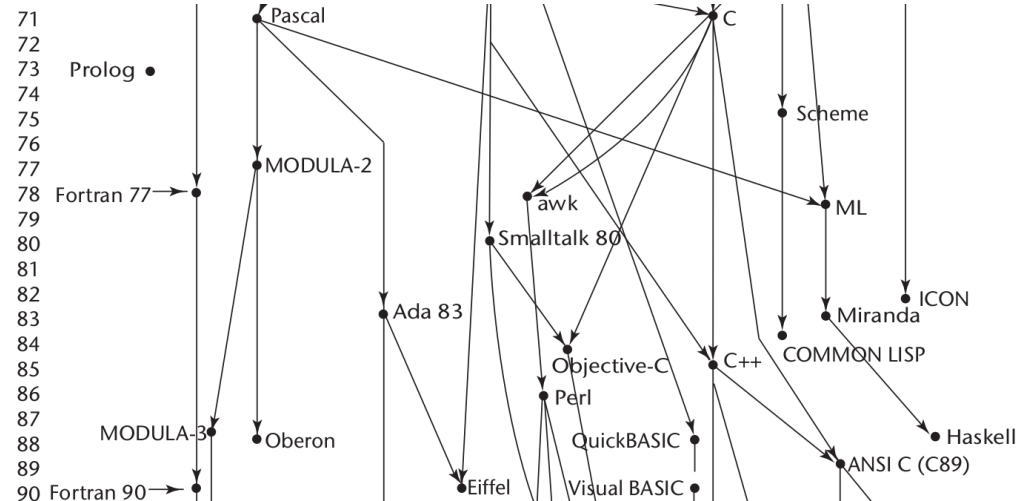
تاریخچهٔ زبان‌های برنامه‌نویسی

- در اواخر دههٔ ۱۹۵۰ و اویل دههٔ ۱۹۶۰ زبان‌های الگول و لیسپ به وجود آمدند. در این زبان‌ها امکان فراخوانی تابع توسط خودش و بنابراین نوشتن توابع بازگشتی وجود داشت.
- در دههٔ ۱۹۷۰ روش‌هایی برای ساختاربندی داده‌ها و ایجاد نوع داده‌های انتزاعی به وجود آمدند. با استفاده از این روش‌ها برنامه‌های پیچیده نظم بیشتری پیدا می‌کردند.
- با افزایش سرعت سخت‌افزار به تکنیک‌هایی برای استفاده بهینه از آنها نیاز بود و بنابراین برنامه‌نویسی همروند برای اجرای قسمت‌های مختلف یک برنامه به طور همزمان و یا اجرای چند برنامه به طور همزمان به وجود آمد. همچنین با به وجود آمدن شبکه‌های کامپیوتری به روش‌هایی برای بهره‌گیری از چندین کامپیوتر به طور همزمان نیاز بود و بنابراین برنامه‌نویسی توزیع شده به وجود آمد.

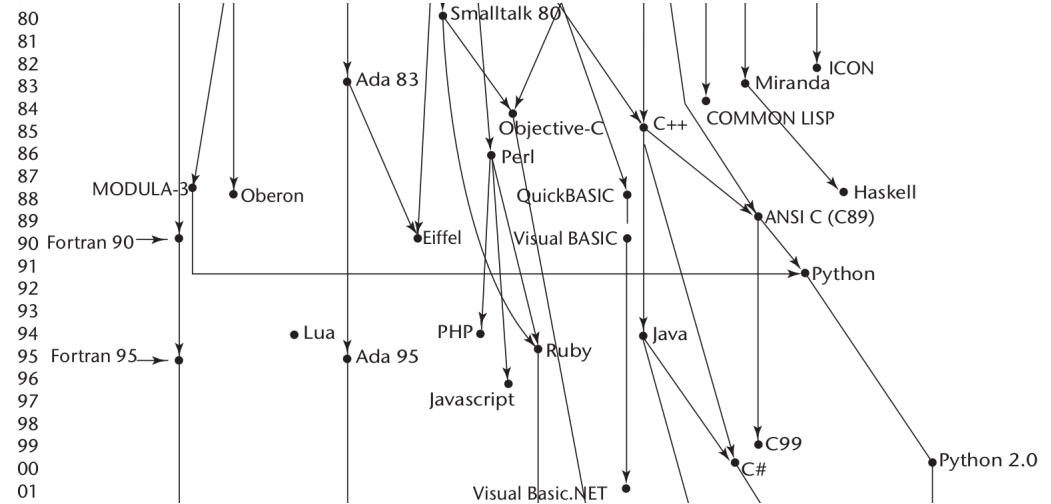
تاریخچهٔ زبان‌های برنامه‌نویسی



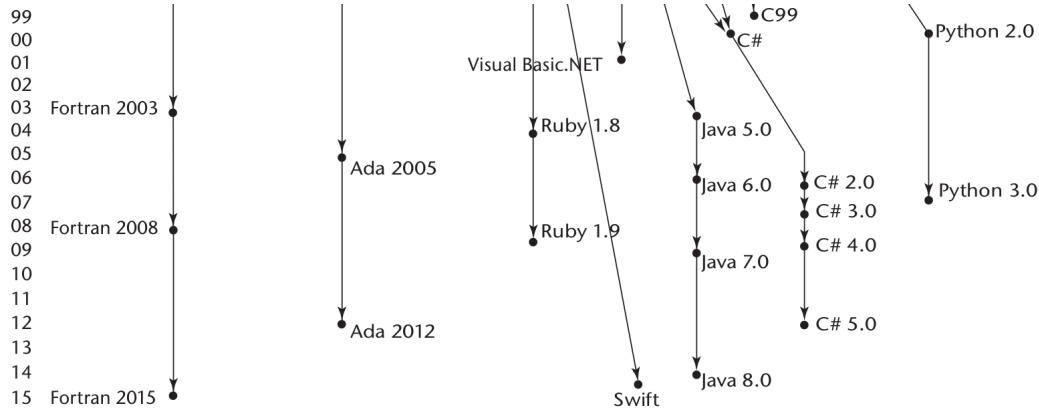
تاریخچهٔ زبان‌های برنامه‌نویسی



تاریخچهٔ زبان‌های برنامه‌نویسی



تاریخچهٔ زبان‌های برنامه‌نویسی



محاسبه پذیری

- برخی از توابع ریاضی محاسبه پذیرند و برخی محاسبه پذیر نیستند. در همه زبان های برنامه نویسی تنها برای توابعی می‌توان برنامه ساخت که محاسبه پذیرند.
- از دیدگاه ریاضی، یک برنامه کامپیوتری در واقع یک تابع است. خروجی یک برنامه بر اساس ورودی برنامه و حالت ماشین قبل از شروع برنامه محاسبه می‌شود.
- در علوم ریاضی، عبارت $2 + 3$ مقدار تعریف شده ای دارد، اما عبارت $0 \div 3$ مقدار تعریف شده ای ندارد. دلیل آن این است که در عملیات تقسیم معکوس ضرب تعریف شده است و هیچ مقداری وجود ندارد که با ضرب آن دو عدد صفر، عدد ۳ به دست بیاید. وقتی به چنین عبارتی در ریاضی بر می‌خوریم محاسبات را نمی‌توانیم ادامه دهیم و می‌گوییم عملیات تعریف نشده است.

محاسبه پذیری

- در نظریه محاسبات، یک عبارت به دو دلیل غیر محاسبه پذیر است :
  ۱. خاتمه یافتن با خطا : محاسبه یک عبارت نمی‌تواند پایان بپذیرد به این علت که تضادی بین عملگردها و عملوندها وجود دارد.
  ۲. پایان ناپذیری یا تصمیم ناپذیری : محاسبه یک عبارت به طور نامحدود ادامه پیدا می‌کند.

محاسبه پذیری

- خاتمه یافتن با خطا : برای مثال هنگام برخورد با عبارت تقسیم بر صفر خطا رخ می‌دهد و محاسبات متوقف می‌شود.
- پایان ناپذیری : محاسباتی باید بر روی ورودی انجام شود اما به ازای برخی از ورودی‌ها، محاسبات ممکن است پایان نپذیرد. برای مثال تابع زیر را در نظر بگیرید.

```
1 def f(x) :
2     if x == 0 : return 0
3     else : return x + f(x-2)
```

- این تابع یک تابع جزئی است، بدین معنی که به ازای برخی از مقادیر ورودی، مقداری را باز نمی‌گرداند. در واقع تابع به ازای برخی ورودی‌ها تا بی‌نهایت ادامه پیدا می‌کند و مقداری را محاسبه نمی‌کند. با فراخوانی `f(4)` یک مقدار معین بازگردانده می‌شود، اما `f(5)` هیچ مقداری ندارد.

روش‌های پیاده‌سازی زبان‌ها

- زبان‌های برنامه‌نویسی میتوانند به یکی از سه روش زیر پیاده سازی شوند : ترجمه [1]، تفسیر [2]، پیاده‌سازی ترکیبی [3]

[1] complitation
[2] interpretation
[3] hybrid implementation

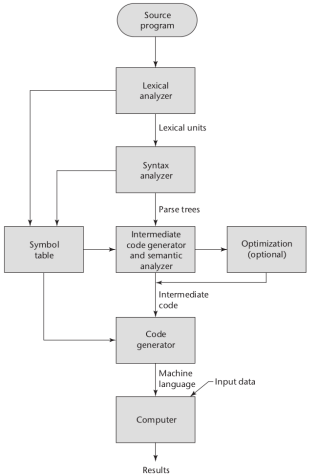
ترجمه

- یک برنامه کامپیوتری می‌تواند مستقیماً به زبان ماشین ترجمه شده، و بر روی کامپیوتر مقصد اجرا شود. این روش را پیاده سازی توسط کامپایلر [1] یا مترجم می‌نامیم.
- مزیت این روش این است که برنامه پس از کامپایل شدن با سرعت بالایی اجرا می‌شود.
- پیاده‌سازی زبان‌هایی مانند کوبول، سی و سی++ با استفاده از کامپایلر صورت گرفته است.
- زبانی را که یک کامپایلر ترجمه می‌کند زبان مبدأ [2] یا زبان منبع نامیده می‌شود.

[1] compiler
[2] source languege

ترجمه

- در شکل زیر مراحل انجام ترجمه نشان داده شده است.



ترجمه

- تحلیل‌گر واژگانی [1] متن نوشته شده در برنامه منبع [2] را به واحدهای واژگانی تبدیل می‌کند. این واحدهای واژگانی می‌توانند شناسه‌ها [3]، کلمات کلیدی [4]، عملگرها [5] و علائم نشانه گذاری [6] باشند. تحلیل‌گر واژگانی، از توضیحات [7] چشم پوشی می‌کند.

[1] lexical analyzer
[2] source program
[3] identifier
[4] key word
[5] operator
[6] punctuation symbol
[7] comment

ترجمه

- تحلیل‌گر نحوی [1] واحدهای واژگانی را از تحلیل‌گر واژگانی دریافت می‌کند و با استفاده از آنها یک ساختار به نام درخت تجزیه [2] می‌سازد. این درخت‌های تجزیه ساختار نحوی یک برنامه را نشان می‌دهند.

- تولید کنندهٔ کد واسط [3] یک برنامه به یک زبان دیگر تولید می‌کند که حد واسط برنامه منبع و برنامه زبان ماشین [4] است. این زبان میانی معمولا نوعی از زبان اسمبلی است.

- تحلیل‌گر معنایی [5] که بخشی از تولید کننده کد واسط است، خطاهای برنامه را بررسی می‌کند.

- کد تولید شده توسط تولید کنندهٔ کد واسط، در برخی موارد به یک واحد بهینه سازی [6] داده می‌شود تا کد تولید شده را در صورت امکان کوچک‌تر و سریع‌تر کند. بسیاری از بهینه سازی‌ها را به سختی می‌توان بر روی کد زبان ماشین انجام داد بنابراین این بهینه سازی‌ها بر روی کد واسط انجام می‌شوند.

[1] syntax analyzer
[2] parse tree
[3] intermediate code generator
[4] machine languege
[5] semantic analyzer
[6] optimization

ترجمه

- در نهایت تولید کنندهٔ کد [1] ، کد بهینه سازی شده در زبان واسط را به برنامه‌ای در زبان ماشین ترجمه می‌کند.
- جدول علائم [2] اطلاعات مورد نیاز برای فرایند کامپایل را نگهداری می‌کند. محتوای این جدول همهٔ نمادها و نوع آنهاست که برای فرایند تولید کد مورد نیاز است. این گونه اطلاعات توسط تحلیل‌گر واژگانی و نحوی در جدولی نگهداری می‌شوند و توسط تحلیل‌گر معنایی و تولیدکننده کد استفاده می‌شوند.
- دقت کنید که برنامه ی نوشته شده تقریبا هیچگاه به تنهایی قابل استفاده نیست بلکه معمولاً نیاز به برنامه‌های جانبی است که توسط برنامه‌های دیگر یا سیستم عامل نوشته شده‌اند. برای مثال برای استفاده از ورودی و خروجی، برنامه نیاز به برنامه‌های جانبی دارد که توسط سیستم عامل برای استفاده از ورودی و خروجی مهیا شده‌اند.

[1] code generator
[2] symbel table

ترجمه

- بنابراین قبل از این که کد تولید شده به زبان ماشین بتواند اجرا شود، برنامه‌های جانبی باید به برنامه مورد نظر پیوند [1] داده شوند. در این فرآیند پیوند دادن یا لینک کردن، برنامه‌های مورد نیاز دیگر (که توسط برنامه نویسان دیگر پیاده سازی شده‌اند) و یا برنامه‌های سیستمی (که توسط سیستم عامل مهیا شده‌اند) باید به کد ماشین تولید شده لینک شوند. برای این کار آدرس کد ماشین برنامه‌های جانبی به برنامه مورد نظر برای اجرا داده می‌شود.
- فرایند پیوند کدهای جانبی به کد تولید شده، توسط یک پیوند دهنده [2] یا لینکر انجام می‌شود.
- بنابراین لینکر وظیفه دارد کد ماشین تولید شده را به کد ماشین برنامه‌های جانبی که به صورت کتابخانه‌ها با ارائهٔ توابع پر استفاده توسعه داده شده‌اند و کد ماشین برنامه‌های سیستمی که توسط سیستم عامل مهیا شده‌اند، پیوند دهد.
- سرعت انتقال کد از حافظه به پردازنده معمولا کمتر از سرعت اجرای کد در پردازنده است و بنابراین سرعت اجرای برنامه را سرعت انتقال کد تعیین می‌کند. به این پدیده تنگنای معماری وان نویمان [3] می‌گوییم.

[1] Link
[2] Linker
[3] Van Neumann bottleneck

ترجمه

- یک پیش پردازنده [1] برنامه‌ای است که یک برنامه را قبل از کامپایل شدن پردازش می‌کند. در مرحله پیش پردازش، معمولاً دستوراتی که میانبر برای دستورات دیگر هستند حذف می‌شوند و با دستورات اصلی جایگزین می‌شوند.
- برای مثال در زبان سی با استفاده از دستور `#include "lib.h"` محتوای فایل `lib.h` در ابتدای برنامه قرار می‌گیرد. در مرحله پیش پردازش محتوای فایل مورد نظر به محتوای فایل کد منبع الحاق می‌شود.

[1] preprocessor

ترجمه

- به عنوان مثال دیگر، در زبان سی می‌توانیم با استفاده از دستور `#define` نام‌های نمادین ایجاد کنیم. به قواعد و دستوراتی که مشخص می‌کنند چگونه یک الگوی خروجی بر اساس یک الگوی ورودی تولید شود، ماکرو [1] گفته می‌شود.
- برای محاسبه ماکزیمم دو عدد می‌توانیم ماکرویی به صورت زیر تعریف کنیم. `#define max(A,B) ((A)>(B) ? (A):(B))`. حال اگر در برنامه منبع داشته باشیم `max(x+2,y)` ، در مرحله پیش پردازش این دستور به دستور `((x+2)>y ? (x+2):(y))` تبدیل می‌شود.

[1] macro

تفسیر

- تفسیر روش دیگری برای پیاده سازی زبان برنامه‌نویسی است. با استفاده از این روش، برنامه‌های زبان منبع توسط یک مفسر [1] خط به خط خوانده می‌شوند، به زبان ماشین تبدیل شده و اجرا می‌شوند.
- مزیت این روش این است که کد برنامه نیاز به کامپایل ندارد و همه جا قابل استفاده است. همچنین با سرعت بیشتری می‌توان برنامه‌های کامپیوتری تولید کرد چرا که کد برنامه را می‌توان به راحتی با استفاده از مفسر خط به خط اجرا و تست نمود.
- از طرفی دیگر عیب این روش این است که معمولاً زمان اجرای برنامه‌ها در آن نسبت به روش ترجمه پایین‌تر است.

[1] interpreter

تفسیر

- در روش تفسیر تنگنای زمان اجرا، کدگشایی دستورات است که بسیار زمان‌بر است و نه انتقال دستورات از حافظه به پردازنده.
- زبان‌هایی مانند ای‌پی‌ال [1] و لیسپ از زبان‌های دهۀ ۱۹۶۰ توسط مفسر پیاده سازی شدند. در سال‌های اخیر زبان‌های وب مانند پی‌اچ‌پی و جاوااسکریپت نیز توسط مفسر پیاده سازی می‌شوند.

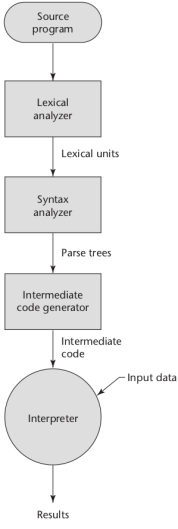
[1] APL

پیاده سازی ترکیبی

- برخی از زبان‌های برنامه سازی توسط یک روش ترکیبی بین تفسیر و ترجمه پیاده سازی شده‌اند. در این نوع پیاده سازی، زبان سطح بالا ابتدا به یک برنامه در یک زبان میانی ترجمه می‌شود. سپس تفسیر از زبان میانی به زبان ماشین آسان‌تر می‌تواند اجرا شود.

پیاده سازی ترکیبی

- روند پیاده سازی زبان در این روش در شکل زیر نشان داده شده است.



پیاده سازی ترکیبی

- زبان پرل [1] و پایتون [2] به این روش پیاده سازی شده‌اند.
- همچنین زبان جاوا به این روش پیاده سازی شده است. زبان میانی بایت کد [3] (کد بایتی) نامیده می‌شود. با استفاده از این روش می‌توان بایت کد را بر روی هر سیستمی که دارای ماشین مجازی جاوا باشد اجرا نمود.
- زبان‌های برنامه‌نویسی به روش‌های مختلفی پیاده سازی شده‌اند. یک مفسر برای استفاده آسان‌تر یک برنامه نویس ارائه می‌شود، یک پیاده سازی ترکیبی برای فراهم کردن امکان جابجایی کد ارائه می‌شود و یک مترجم برای بهبود سرعت برنامه استفاده می‌شود.

[1] Perl
[2] Python
[3] byte code

نحو و معناشناسی

نحو و معناشناسی

- در این فصل با تعریف نحو و معناشناسی آغاز می‌کنیم. سپس روش‌های مهم برای توصیف نحو را ارائه می‌کنیم. در مورد گرامرهای مستقل از متن، فرایند اشتقاق، درخت تجزیه، ابهام و تقدم عملگرها توضیح می‌دهیم.
- گرامرهای صفت [1] را که برای توصیف نحو و معناشناسی در زبان‌های برنامه‌نویسی استفاده می‌شوند توضیح می‌دهیم. در انتها سه روش رسمی برای توصیف معناشناسی، یعنی معناشناسی عملیاتی [2] ، معناشناسی دلالتی [3] ، و معناشناسی اصلی موضوعی [4] را توضیح می‌دهیم.

[1] attribute grammar
[2] operational semantics
[3] denotational semantics
[4] axiomatic semantics

نحو و معناشناسی

- برای پیاده سازی درست یک زبان برنامه‌نویسی لازم است آن زبان را به طور دقیق و قابل فهم توصیف کنیم. برای زبان الگول در ابتدا توصیف دقیقی ارائه شد ولی آن توصیف قابل فهم نبود. اهمیت توصیف درست زبان این است که کسانی که زبان را پیاده سازی می‌کنند باید آن را درست بفهمند و همچنین استفاده کنندگان زبان نیاز دارند ویژگی‌های زبان را به درستی بشناسند.
- مطالعه زبان‌های برنامه نویسی مانند مطالعه زبان‌های طبیعی می‌تواند به دو قسمت تقسیم شود : نحو شناسی [1] به مطالعه صورت عبارت در زبان مورد نظر و معناشناسی [2] به مطالعه معانی عبارات می‌پردازد.
- به عبارت دیگر در نحو شناسی مطالعه می‌کنیم چگونه کلمات و تکواژها [3] ترکیب می‌شوند تا عبارات و جمله‌ها را بسازند، اما در معناشناسی به مطالعهٔ معنی آن عبارات و جملات و درستی و نادرستی آنها می‌پردازیم.

[1] syntax
[2] semantics
[3] morphem

نحو و معناشناسی

- برای مثال در زبان جاوا ترکیب نحوی برای یک حلقه `while` به صورت زیر است.

```
while (boolean-expr) statement
```

- معنی این عبارت این است که تا وقتی که مقدار جمله `boolean-expr` درست است، دستورات عبارت `statement` را تکرار کن. در صورتی که مقدار جمله `boolean-expr` نادرست شد، کنترل برنامه به بعد از حلقه منتقل می‌شود.

توصیف نحوی

- یک زبان چه طبیعی باشد مانند زبان انگلیسی و چه ساختگی مانند زبان جاوا، شامل رشته‌هایی است که از کاراکترهایی از یک الفبای معین تولید شده‌اند. به رشته‌های یک زبان، جمله نیز گفته می‌شود.
- قوائد نحوی یک زبان تعیین می‌کنند کدام رشته‌های تولید شده از یک الفبا متعلق به زبان هستند. زبان انگلیسی به طور مثال شامل یک مجموعه بزرگ و پیچیده از قوائد نحوی است که جملات زبان را تعیین می‌کنند. در مقایسه با زبان انگلیسی، حتی پیچیده‌ترین و بزرگ‌ترین زبان‌های برنامه‌نویسی قوائد نحوی ساده‌تر و کمتری دارند.

توصیف نحوی

- توصیف رسمی قوائد نحوی زبان‌های برنامه‌نویسی، معمولاً نحوهٔ تولید کوچک‌ترین اجزای زبان را مشخص نمی‌کند. کوچکترین اجزای یک جمله را تکواژ [1] می‌نامیم. توصیف تکواژها را می‌توان به طور جداگانه با استفاده از یک توصیف کنندهٔ تکواژها مشخص کرد. تکواژها در یک زبان برنامه‌نویسی شامل اعداد و ارقام، عملگرها، کلمات کلیدی و شناسه‌ها و اسامی می‌شود. در واقع می‌توانیم یک برنامه را مجموعه‌ای از تکواژها در نظر بگیریم.
- تکواژها را می‌توانیم به چند گروه تقسیم کنیم : برای مثال شناسه‌ها [1] شامل اسامی متغیرها، توابع، کلاس‌ها و غیره می‌شوند. برای هر گروه از تکواژها که در یک دسته قرار می‌گیرند، یک نام در نظر می‌گیریم که به آن نشانه یا توکن [2] می‌گوییم.

[1] lexeme
[1] identifier
[2] token

توصیف نحوی

- عبارت `index = 2 * count + 10 ;` در زبان جاوا را در نظر بگیرید.
- تکواژها و توکن‌های مربوط به این عبارت را می‌توانیم به صورت زیر نشان دهیم.

lexemes	**tokens**
`index`	identifier
`=`	equal-sign
`2`	int-literal
`*`	mult-op
`count`	identifier
`+`	plus-op
`10`	int-literal
`;`	semicolon

توصیف نحوی

- زبان‌ها را می‌توان به دو روش توصیف کرد. به وسیله تشخیص [1] یا به وسیله تولید [2]

[1] recognition
[2] generation

تشخیص دهندهٔ زبان

- فرض کنید زبان L را داریم که از الفبای Σ استفاده می‌کند. برای تعریف زبان L توسط تشخیص دهنده باید از یک مکانیزم R به عنوان دستگاه تشخیص دهندهٔ زبان استفاده کنیم که قادر باشد رشته‌هایی که از الفبای Σ تشکیل شده است را دریافت و تشخیص دهد آیا آن رشته عضو زبان است یا خیر. R یا رشته را می‌پذیرد و یا رد می‌کند. این دستگاه تشخیص دهنده مانند فیلتری است که رشته‌های مجاز در آن زبان را از رشته‌های غیر مجاز جدا می‌کند. اگر R برای همهٔ رشته‌ها بر روی الفبای Σ، تنها جملات زبان L را پذیرفت آنگاه R یک توصیف برای زبان L است.
- تحلیل‌گر نحوی در یک کامپایلر در واقع یک تشخیص دهنده برای زبانی است که کامپایل می‌کند. یک تحلیل‌گر نحوی یک ورودی را که یک برنامه دریافت کرده است دریافت کرده و تعیین می‌کند آیا آن برنامه متعلق به زبان مورد ترجمه است یا خیر.

تولید کننده زبان

- یک تولید کننده زبان دستگاهی است که جملات یک زبان را تولید می‌کند.
- مکانیزمی که توسط آن یک زبان تولید می‌شود گرامر [1] نامیده می‌شود.
- توسط یک دستگاه تولید کنندهٔ زبان یا یک گرامر می‌توانیم بررسی کنیم آیا یک برنامه توسط آن گرامر تولید می‌شود یا خیر.
- یکی از روش‌های توصیف قوائد نحوی، استفاده از گرامر است. معمولاً برای توصیف زبان از گرامر آن استفاده می‌کنیم.

[1] grammar

گرامرهای مستقل از متن

- در اواسط دههٔ ۱۹۵۰ نوام چامسکی [1] یکی از زبان شناسان مطرح، چهار دسته از گرامرها را برای تولید چهار دسته از زبان‌ها توصیف کرد. دو دسته از این گرامرها به نام گرامرهای منظم [2] و گرامرهای مستقل از متن [3] برای توصیف نحوی زبان‌های برنامه‌نویسی بسیار مورد استفاده قرار گرفتند.
- صورت توکن‌ها در زبان‌های برنامه‌نویسی می‌توانند توسط گرامرهای منظم توصیف شوند و کل زبان برنامه‌نویسی معمولاً می‌تواند توسط گرامرهای مستقل از متن وصف شود.

[1] Noam Chamsky
[2] Regular
[3] Context-free

گرامرهای مستقل از متن

- کمی بعد از انتشار تحقیقات چامسکی، جان باکوس [1] که عضو گروهی بود که بر روی زبان الگول کار می‌کردند، مقاله‌ای در مورد روش توصیف زبان‌های برنامه‌نویسی منتشر کرد. این روش جدیدِ توصیف زبان، بعدها توسط پیتر نائور [2] کمی اصلاح شد. این فرمِ توصیف، بعدها به نام فرم باکوس–نائورَ [3] یا بی‌ان‌اف (BNF) مشهور شد.

[1] John Backus
[2] Peter Naur
[3] Backus–Naur form (BNF)

گرامرهای مستقل از متن

- یک فرا زبان [1] زبانی است که برای توصیف یک زبان دیگر استفاده می‌شود. یک گرامر در واقع یک فرا زبان برای زبان‌های برنامه‌نویسی است. گرامر در واقع ساختار نحوی یک زبان را مشخص می‌کند. برای مثال عبارت تخصیص مقدار [2] در زبان جاوا را می‌توانیم با مفهوم <assign> نمایش دهیم. و می‌توانیم بنویسیم :
<assign> → <var> = <expression>
- مقدار سمت چپ علامت فلش را lhs [1] می‌نامیم که مفهومی است که توسط این گرامر می‌خواهیم تعریف کنیم. مقدار سمت راست علامت فلش را rhs [2] می‌نامیم که تشکیل شده‌است از ترکیبی از تکواژها و مفاهیم دیگر. به طور کلی عبارت lhs → rhs را یک قانون تولید [3] می‌نامیم.

---

[1] metalanguage
[2] assignment expression
[1] left-hand side
[2] right-hand side
[3] production rule

گرامرهای مستقل از متن

- در مثالی که بیان شد مفاهیم <var> و <expression> باید تعریف شوند تا <assign> بتواند کاملا تعریف شده و قابل استفاده باشد. با استفاده از قانونی که بیان شد می‌توانیم عبارت تخصیص مقدار زیر را بسازیم.

```
total = t1 + t2
```

- مفاهیم انتزاعی در گرامر را نمادهای غیر پایانی [1] یا متغیر و تکواژها را نمادهای پایانی [2] یا ترمینال می‌نامیم. یک گرامر از تعدادی قوانین تولید تشکیل شده‌است.

- یک نماد غیرپایانی معمولاً می‌تواند دو یا چند تعریف داشته باشد. چندین تعریف از یک نماد را می‌توانیم در چند قانون تعریف کنیم و یا اینکه در یک قانون تعریف و از علامت خط عمودی | در سمت راست قانون برای جداسازی قوانین متعدد استفاده کنیم.

[1] nonterminal symbols
[2] terminal symbols

گرامرهای مستقل از متن

- برای مثال عبارت `if` در جاوا را می‌توانیم به صورت زیر تعریف کنیم.

<if-stmt> → if (<logic-expr>) <stmt>
<if-stmt> → if (<logic-expr>) <stmt> else <stmt>

- برای ترکیب دو قانون بالا می‌توانیم بنویسیم :

<if-stmt> → if (<logic-expr>) <stmt> | if (<logic-expr>) <stmt> else <stmt>

گرامرهای مستقل از متن

- گرچه گرامرهای مستقل از متن ساده به نظر می‌سند ولی با استفاده از آنها می‌توانیم همه قوانین نحوی زبان‌های برنامه‌نویسی را توصیف کنیم.
- یک قانون می تواند بازگشتی باشد بدین معنی که مفهوم سمت چپ می‌تواند در سمت راست قانون نیز به کار رود. برای مثال برای تعریف یک لیست از شناسه‌ها می‌توانیم بنویسیم :

<id-list> → <id> | <id> ، <id-list>

- در اینجا از یک قانون بازگشتی استفاده شده است تا بتوانیم یک لیست با هر طول دلخواهی را بسازیم.

اشتقاق

- همانطور که گفتیم یک گرامر در واقع یک دستگاه تولیدکننده برای توصیف زبان است. یک جمله از یک زبان توسط دنباله‌ای از اعمال قوانین با شروع از یک نماد غیر پایانی که نماد آغازی [1] نامیده می‌شود. این دنباله از اعمال قوانین را فرایند اشتقاق [2] می‌نامیم.
- معمولاً در یک زبان برنامه نویسی نماد آغازی <program> نامیده می‌شود که یک برنامه را توصیف می‌کند.

[1] start symbol
[2] derivation

اشتقاق

- فرض کنید گرامر زیر یک زبان ساده را توصیف می‌کند.

<program> → begin <stmt-list> end
<stmt-list> → <stmt> ; | <stmt> ; <stmt-list>
<stmt> → <var> = <expression>
<var> → A | B | C
<expression> → <var> + <var> | <var> - <var> | <var>

اشتقاق

- یک فرایند اشتقاق به صورت زیر است.

<program> $\Rightarrow$ begin <stmt-list> end
$\Rightarrow$ begin <stmt> ; <stmt-list> end
$\Rightarrow$ begin <var> = <expresssion> ; <stmt-list> end
...
$\Rightarrow$ begin A = B + C ; B = C ; end

- علامت $\Rightarrow$ را می‌خوانیم «به دست می‌دهد» یا «می‌دهد» یا «مشتق می‌کند».

اشتقاق

- در هر مرحله از فرایند اشتقاق یکی از نمادهای غیر پایانی جایگزین می‌شود. هر یک از جملات به دست آمده در فرایند اشتقاق را یک صورت جمله‌ای [1] می‌نامیم.
- اگر در فرایند اشتقاق همیشه ابتدا اولین متغیر از سمت چپ جایگزین شود، آن فرایند را اشتقاق چپ [2] می‌نامیم. فرایند اشتقاق تا جایی ادامه پیدا می‌کند که هیچ متغیری در صورت جمله‌ای باقی نماند. یک صورت جمله ای در آن هیچ متغیری نباشد را یک جمله می‌نامیم.

[1] sentential form
[2] leftmost derivation

اشتقاق

- فرایند اشتقاق می‌تواند از سمت راست نیز انجام شود، یعنی همیشه اولین متغیر سمت راست را جایگزین کنیم و یا فرایند اشتقاق ممکن است بدون هیچ ترتیبی انجام شود. ترتیب اشتقاق هیچ تأثیری بر روی زبان تولید شده توسط یک گرامر ندارد.
- با یک جستجوی کامل بر روی گرامر می‌توان جملات یک زبان را یک به یک تولید کرد. البته اگر یک زبان نامحدود باشد تعداد جمله‌های آن نامحدود است و امکان تولید همه جملات وجود ندارد.
- گرامر زیر را در نظر بگیرید. این گرامر عبارات تخصیص مقدار در یک زبان ساده را تعریف می‌کند.

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr> | <id> * <expr> | (<expr>) | <id>

- برای مثال توسط این گرامر اگر می‌توانیم عبارت `A = B * (A+C)` را بسازیم.

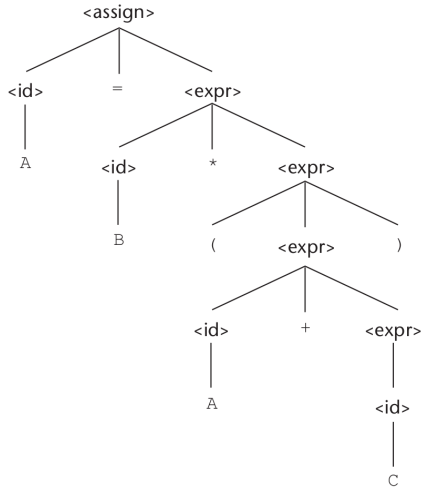
درخت تجزیه

- گرامرها در واقع توسط یک ساختار سلسله مراتبی در فرایند اشتقاق جملات یک زبان را می‌سازند، بدین معنی که در هر سطح از سلسله مراتب یک از متغیرها جایگزین می‌شود. این ساختار سلسله مراتبی درخت تجزیه [1] نامیده می‌شود.

[1] parse tree

درخت تجزیه

- برای مثال درخت تجزیه در شکل زیر نشان می‌دهد چگونه یک عبارت تخصیص مقدار با استفاده از گرامر قبلی به دست می‌آید.

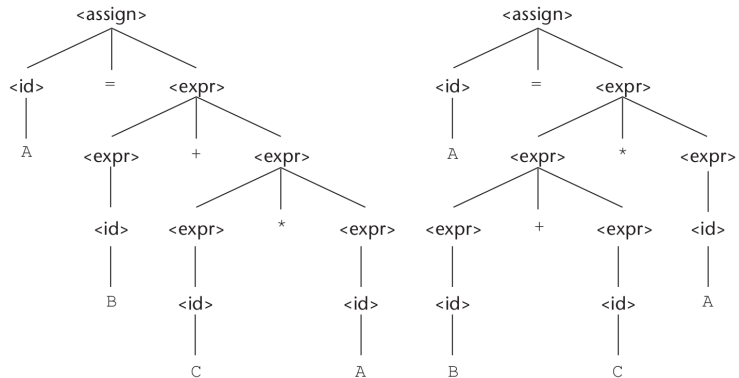


درخت تجزیه

- هر یک از رئوس میانی در این درخت توسط متغیرها برچسب زده شده‌اند و هر برگ توسط یک ترمینال یا نماد پایانی برچسب زده شده‌است.

ابهام

- اگر یک جمله متعلق به یک گرامر، بتواند توسط بیش از یک درخت تجزیه تولید شود، به آن گرامر یک گرامر مبهم گفته می‌شود.
- گرامر قبلی برای تولید عبارات تخصیص مقدار را در نظر بگیرید. برای جمله `A = B + C * A` دو درخت تجزیه متفاوت وجود دارد.



ابهام

- ابهام در یک گرامر ایجاد مشکل می‌کند، چرا که کامپایلرها معمولاً معانی جملات را از روی ساختار درخت به دست می‌آورند. در مثال قبل، کامپایلر برای محاسبه عبارت تخصیص مقدار، طبق درخت تجزیه، کد ماشین مورد نظر را تولید می‌کند. وقتی دو درخت تجزیه وجود داشته باشند، در واقع دو معنی برای یک عبارت وجود دارد و کامپایلر نمی‌تواند تصمیم بگیرد کدام معنی را انتخاب کند.
- به طور کلی اثبات شده‌است که هیچ الگوریتمی برای تعیین مبهم بودن یک گرامر وجود ندارد.
- اما برخی از ویژگی‌های یک گرامر می‌توانند تعیین کنندهٔ مبهم بودن گرامر باشند. اگر یک گرامر جمله‌ای را با دو اشتقاق چپ متفاوت یا دو اشتقاق راست متفاوت به دست بیاورد، آن گرامر مبهم است.
- در بسیاری از موارد یک گرامر را می‌توان به نحوی نوشت که مبهم نباشد. اگر نتوان یک گرامر را به نحوی نوشت که مبهم نباشد، زبان آن گرامر یک زبان ذاتاً مبهم است.

تقدم عملگرها

- وقتی در یک عبارت محاسباتی، چند عملگر وجود داشته باشد، یک مشکل معنایی که به وجود می‌آید، ترتیب ارزیابی عملگرهاست. برای مثال، در عبارت $x + y * z$ آیا باید ابتدا عملگرجمع ارزیابی شود و یا عملگر ضرب؟
- بدین منظور، تقدم عملگرها تعریف می‌شوند. برای مثال، اگر تقدم ضرب بیشتر از جمع تعریف شود، آنگاه عملگر ضرب باید قبل از جمع ارزیابی شود.
- برای گرامر مبهم قبلی می‌توانیم یک گرامر غیر مبهم بنویسیم به طوری که درخت تجزیه ابتدا عملگر جمع و سپس عملگر ضرب را تجزیه کند. بدین ترتیب وقتی از برگ‌های درخت تجزیه برای ارزیابی یک عبارت آغاز می‌کنیم ابتدا عملگر ضرب را اعمال می‌کنیم.

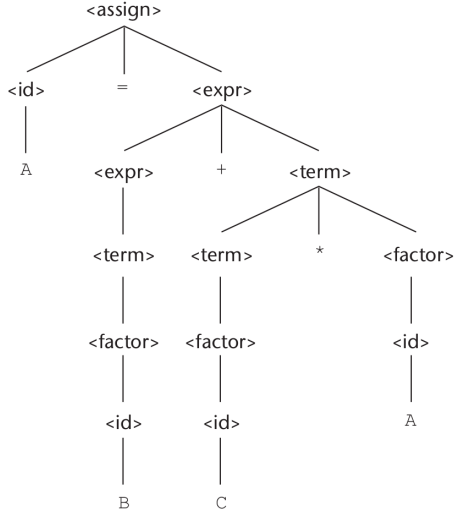
تقدم عملگرها

- گرامر غیر مبهم زیر معادل گرامر مبهم قبلی است.

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <term> | <term>
<term> → <term> ∗ <factor> | <factor>
<factor> → (<term>) | <id>

تقدم عملگرها

- بدین ترتیب برای عبارت `A = B + C * A` تنها یک درخت تجزیه به صورت زیر وجود خواهد داشت.



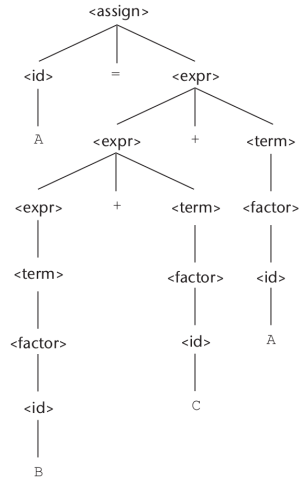
وابستگی عملگرها

- وقتی در یک عبارت دو عملگر وجود داشته باشد که تقدم برابر داشته باشند، به قوانین معنایی نیاز داریم تا بدانیم کدام عملگر باید زودتر اجرا شود. وابستگی عملگرها [1] مشخص می‌کند که در شرایطی که تقدم یکسان است به کدام عملگر اولویت بالاتری داده می‌شود.

[1] operator associativity

وابستگی عملگرها

- برای مثال عبارت A = B + C + A را در نظر بگیرید. درخت تجزیه برای این عبارت طبق گرامر غیر مبهمی که قبلاً ارائه شده به صورت زیر است.



وابستگی عملگرها

- در این درخت تجزیه عملگر جمع اول زودتر محاسبه می‌شود. این عملیات صحیح است اگر وابستگی عملگر جمع از سمت چپ باشد که معمولاً هم همینطور است.
- در ریاضیات می‌گوییم عملگر جمع خاصیت شرکت پذیری [1] دارد، بدین معنی که وابستگی از چپ با وابستگی از راست معادل است، یعنی `(A+B) + C = A + (B+C)`
- در کامپیوتر اما در عملیات جمع هم وابستگی می‌تواند مهم باشد، بدین معنی که محاسبات از چپ و راست می‌توانند نتایج متفاوتی تولید کنند. به طور مثال فرض کنید چند عدد را می‌خواهیم جمع کنیم و نتیجه را با دقت ۷ رقم اعشار محاسبه کنیم. اگر اولین عدد $۱۰^{۷}$ باشد و بقیه اعداد ۱ باشند اضافه کردن اعداد ۱ تأثیری در نتیجه ندارد چرا که در رقم ۸ ام اضافه می‌شوند، و نتیجه در هر بار افزایش به میزان یک واحد برابر با $۱۰^{۷} \times ۱.۰۰۰۰۰۰۰$ خواهد بود.
- اما اگر ۱۰ عدد ۱ را با هم جمع کنیم و در نهایت با $۱۰^{۷}$ جمع کنیم نتیجه $۱۰^{۷} \times ۱.۰۰۰۰۰۰۱$ خواهد شد.

[1] associative property

وابستگی عملگرها

- در طراحی گرامر یک زبان برنامه نویسی وابستگی عملگرها باید در نظر گرفته شود.
- برای مثال در زبان سی تقدم عملگر * و ++ یکسان است، اما وابستگی آنها از راست به چپ است، بدین معنا که اگر این دو عملگر در کنار یکدیگر قرار بگیرند، کامپایلر ابتدا عملگر سمت راست را محاسبه می‌کند.
- عبارت ++p* ابتدا مقدار اشاره‌گر را افزایش می‌دهد و سپس مقدار آن را ارزیابی می‌کند، زیرا تقدم این دو عملگر یکسان و وابستگی آنها از راست به چپ است.
- اگر بخواهیم ابتدا مقدار اشاره گر را ارزیابی و سپس به آن یک واحد بیافزاییم، از عبارت ++(p*) استفاده می‌کنیم.

وابستگی عملگرها

- وقتی در یک گرامر، یک مفهوم یا متغیر lhs در یک قانون، در طرف چپ rhs باشد می‌گوییم این قانون بازگشتی چپ [1] است. یک قانون بازگشتی چپ تولید وابستگی از چپ می‌کند.
- به همین ترتیب اگر مفهوم lhs در یک قانون، در طرف راست rhs باشد می‌گوییم قانون بازگشتی راست [2] است. یک قانون بازگشتی راست تولید وابستگی از راست می‌کند.

<factor> → <expr> ∗∗ <factor> | <expr>
<expr> → (<expr>) | id

[1] left recursive
[2] right recursive

گرامرهای غیر مبهم برای if-else

- قانون گرامر if را به صورت زیر در نظر بگیرید.

<if-stmt> → if(<logic-expr>) <stmt> | if(<logic-expr>) <stmt> else <stmt>

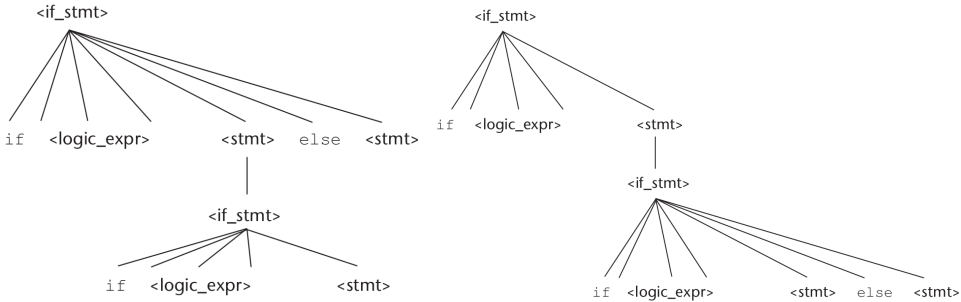
- حال فرض کنید می‌خواهیم if های تو در تو داشته باشیم. اگر قانون <if-stmt> → <stmt> را اضافه کنیم، این گرامر تبدیل به یک گرامر مبهم می‌شود.
- یکی از صورت‌های جمله‌ای که توسط این گرامر ساخته می‌شود برابر است با

if(<logic-expr>) if(<logic-expr>) <stmt> else <stmt>

- ابهام به وجود آمده این است که مشخص نیست else مربوط به کدام if است.

گرامرهای غیر مبهم برای if-else

- دو درخت تجزیه برای این صورت جمله‌ای به شکل زیر هستند.



گرامرهای غیر مبهم برای if-else

- کد زیر را در نظر بگیرید.

```
۱ if (done == true)
۲ if (denom == 0)
۳     quotient = 0 ;
۴     else quotient = num/denom ;
```

- اگر از درخت تجزیه اول (سمت چپ) استفاده شود، else وقتی اجرا می‌شود که مقدار done نادرست باشد.

گرامرهای غیر مبهم برای if-else

- حال می‌خواهیم یک گرامر غیر مبهم برای if بنویسیم. قانون عبارات شرطی در همهٔ زبان‌های برنامه‌نویسی این است که else با نزدیک‌ترین if قبل از آن تطبیق داده می‌شود. بنابراین بین دو عبارت if و else نمی‌توان یک عبارت if بدون else گذاشت چرا که در غیر اینصورت else با if دوم تطبیق داده می‌شود.
- برای حل این مشکل دو حالت در نظر می‌گیریم. حالتی که if بدون else باشد که باید در انتها یک بلوک if-else قرار بگیرد و حالتی که حالتی که if همراه با else باشد که در اینصورت می‌توانیم if-else های تو در تو داشته باشیم.

<stmt> → <matched> | <unmatched>
<matched> → if (<logic-expr>) <matched> else <matched> | <non-if-stmt>
<unmatched> → if (<logic-expr>) <stmt> | if (<logic-expr>) <matched> else <unmatched>

گرامرهای مستقل از متن تعمیم یافته

- تعدادی روش جهت تعمیم گرامرهای مستقل از متن برای بهبود خوانایی آنها پیشنهاد داده شده‌اند.
- در تعمیم اول، در سمت راست قانون می‌توانیم یک قسمت اختیاری قرار دهیم. هر عبارتی که در بین دو علامت براکت [ ] قرار بگیرد اختیاری است و می‌تواند وجود داشته باشد یا تهی باشد.
- برای مثال در عبارت زیر قسمت آخر اختیاری است.

<if-stmt> → if (<expression>) <stmt> else [else <stmt>]

گرامرهای مستقل از متن تعمیم یافته

- در تعمیم دوم، در سمت راست عبارت می‌توانیم قسمتی را بین دو علامت آکولاد { } قرار دهیم، که بدین معنی است که عبارت بین آکولاد می‌تواند به هر تعداد بار دلخواه تکرار شود. بنابراین با استفاده از این روش می‌توانیم قانون‌های بازگشتی را ساده‌تر بنویسیم. برای مثال

<id-list> → <id> { , <id> }

گرامرهای مستقل از متن تعمیم یافته

- در تعمیم سوم، وقتی قسمتی از یک عبارت می‌تواند به چند حالت مختلف وجود داشته باشد. آن حالت‌ها را با استفاده از علامت | در بین دو علامت پرانتز () از یکدیگر جدا می‌کنیم.
- برای مثال

<term> → <term> (* | / | %) <factor>

- گرامر مستقل از متن قانون بالا، به صورت زیر نوشته می‌شود.

<term> → <term> * <factor> | <term> / <factor> | <term> % <factor>

- بنابراین در گرامر مستقل از متن تعمیم یافته علامت‌های براکت، آکولاد و پرانتز جزء زبان گرامر هستند و ترمینال محسوب نمی‌شوند.

گرامرهای مستقل از متن تعمیم یافته

- برای ساخت عبارت‌های ریاضی می‌توانیم از گرامر مستقل از متن تعمیم یافته زیر استفاده کنیم.

<expr> → <term> { (+ | -) <term> }
<term> → <factor> { (* | /) <factor> }
<factor> → <exp> { ** <exp> }
<exp> → (<expr>) | <id>

گرامرهای مستقل از متن تعمیم یافته

- قبلا گفتیم که گرامرها یک دستگاه تولید کننده زبان هستند ولی می‌توانیم از آنها به عنوان تشخیص دهنده نیز استفاده کنیم. به ازای یک جمله باید الگوریتمی بنویسیم که بررسی کند آیا آن جمله توسط گرامر داده شده قابل تولید است یا خیر.

گرامر صفت

- یک گرامر صفت [1] وسیله ای است که با استفاده از آن ساختارهای دیگری که توسط گرامر مستقل از متن قابل توصیف نیستند، توصیف می‌شوند. در واقع یک گرامر صفت نوعی تعمیم برای گرامر مستقل از متن است.
- از گرامرهای صفت برای توصیف معانی جمله‌های تولید شده توسط گرامر استفاده می‌کنیم.

[1] attribute grammar

گرامر صفت

- برخی از ویژگی‌های زبان‌های برنامه‌نویسی نمی‌توانند توسط گرامرهای مستقل از متن بیان شوند. یکی از این ویژگی‌ها سازگاری نوع داده‌ها [1] است.
- برای مثال در زبان جاوا متغیرهای نوع اعشاری نمی‌توانند به متغیرهای نوع صحیح نسبت داده شوند، ولی برعکس آن امکان پذیر است. البته توصیف این ویژگی با استفاده از گرامر مستقل از متن امکان پذیر است، ولی باید تعدادی قوانین و نمادهای غیر پایانی به گرامر بیافزاییم که در این صورت گرامر زبان جاوا بسیار پیچیده و غیر قابل استفاده خواهد شد.

[1] type compatibility

گرامر صفت

- یک مثال دیگر از ویژگی‌های جاوا که ثابت شده است نمی‌توان آن را با گرامر مستقل از متن بیان کرد این است که همه متغیرها باید قبل از استفاده تعریف شده باشند.
- به این دلایل به قوانینی نیاز داریم که علاوه بر نحو، معنای عبارات را نیز توصیف کنند. به این دسته از قوانین، قوانین معناشناسی ایستا [1] می‌گوییم.
- در واقع قوانین معناشناسی ایستا مربوط به قوانین معنایی برنامه هستند که در زمان کامپایل قابل بررسی هستند.

[1] static semantic rules

گرامر صفت

- یکی از ابزارهایی که برای توصیف معناشناسی ایستا به کار می‌رود، گرامر صفت است که توسط دونالد کنوث برای توصیف نحو و معناشناسی ایستا ابداع شد.
- گرامرهای صفت تقریبا در همهٔ کامپایلرها به صورت غیر رسمی استفاده شده‌اند.
- معناشناسی پویا مربوط به معانی عبارات است که بعدها به آن اشاره خواهیم کرد.

گرامر صفت

- گرامرهای صفت در واقع گرامرهای مستقل از متن هستند که به آنها تعدادی صفت، توابع محاسبه صفت [1] و توابع مسندی [2] اضافه شده است.
- صفت‌ها به نمادهای گرامر (نمادهای پایانی و غیر پایانی) مربوط می‌شوند. توابع محاسبه صفت که توابع معنایی نیز نامیده می‌شوند، به قوانین گرامر مربوط می‌شوند. از این قوانین برای محاسبه صفت‌ها استفاده می‌شود. توابع مسندی، معنای قوانین را بیان می‌کنند و محدودیت‌هایی بر روی قوانین گرامر اعمال می‌کنند.

[1] attribute computation function
[2] predicate function

تعریف گرامر صفت

- به ازای هر نماد X در یک گرامر مجموعه‌ای از صفت‌ها به نام A(X) وجود دارد. مجموعهٔ A(X) به دو مجموعهٔ مجزا افراز می‌شود. مجموعه‌های S(X) و I(X) که صفت‌های ساخته شده [1] و صفت‌های ارث برده شده [2] نامیده می‌شوند.

[1] Synthesized attributes
[2] Inherited attributes

تعریف گرامر صفت

- به ازای هر قانون گرامر، مجموعه‌ای از توابع معنایی وجود دارد.
- برای قانون

$X_0 \to X_1 \cdots X_n$

صفت‌های ساخته شده به صورت $S(X_0) = f(A(X_1), \cdots, A(X_n))$ محاسبه می‌شوند. بنابراین در یک درخت تجزیه صفت‌های ساخته شده با محاسبهٔ صفت‌های فرزندان به دست می‌آید.

- صفت‌های به ارث برده شده به صورت $I(X_j) = f(A(X_0), \cdots, A(X_{j-1}), A(X_{j+1}), \cdots, A(X_n))$ محاسبه می‌شوند. پس صفت‌های به ارث برده شده به صفت‌های پدر و همزادها [3] بستگی دارد.
- صفت‌های ساخته شده معانی را در درخت تجزیه به بالا منتقل می‌کنند در حالی که صفت‌های به ارث برده شده معانی را در درخت تجزیه به پایین منتقل می‌کنند.

[3] sibiling

تعریف گرامر صفت

- یک تابع مسندی یک تابع منطقی است که مقدار آن درست یا نادرست است. این تابع محدودیت‌هایی بر روی قوانین گرامر اعمال می‌کند. نادرست بودن یک تابع مسندی، نشان دهندهٔ نقص در معنای عبارت است.
- یک درخت تجزیه برای یک گرامر صفت درخت تجزیه‌ای است که هر کدام از رئوس آن دارای مجموعه‌ای از صفت‌ها باشد که این مجموعه می‌تواند تهی نیز باشد.

تعریف گرامر صفت

- صفت های ذاتی [1] صفت‌های ساخته شده مربوط به برگ‌های درخت تجزیه هستند. برای مثال نوع یک متغیر در یک برنامه یک صفت ذاتی است که می‌توان آن را از جدول نمادها دریافت کرد. هنگامی که یک درخت تجزیه ساخته می‌شود، اولین ویژگی‌هایی که قابل محاسبه هستند، ویژگی‌های ذاتی برگ‌های درخت هستند. پس از آن، با استفاده از توابع صفت می‌توان صفت‌های رئوس دیگر را محاسبه نمود.

[1] intrinsic attributes

مثال گرامر صفت

- برای مثال در زبان آدا [1] ، نام یک تابع باید در پایان تعریف تابع نیز نوشته شود. این قید را نمی‌توان با استفاده از گرامر مستقل از متن بیان کرد. با استفاده از گرامر صفت این قانون را به صورت زیر بیان می‌کنیم.

Syntax rule : <proc-def> → procedure <proc-name>[1]
<proc-body> end <proc-name>[2]
Predicate : <proc-name>[1] string == <proc-name>[2] string

- دقت کنید هرگاه یک نماد غیر پایانی در سمت راست یک گرامر صفت تکرار شود، به ازای هر تکرار یک اندیس در بین دو براکت قرار می‌دهیم.

- در این گرامر صفت بیان کردیم که نام یک تابع که یک رشته بعد از کلمهٔ کلیدی procedure است، باید با نام تابع که انتهای تعریف تابع، بعد از کلمهٔ کلیدی end نوشته می‌شود، همخوانی داشته باشد.

[1] Ada

مثال گرامر صفت

- حال یک مثال دیگر را در نظر می‌گیریم. در این مثال می‌خواهیم در یک زبان برنامه نویسی، عبارت‌های تخصیص مقدار داشته باشیم. نام متغیرها می‌تواند A یا B یا C باشد و نوع متغیرها می‌تواند int یا real باشد. سمت راست یک عبارت تخصیص مقدار می‌تواند یک متغیر و یا جمع چندین مقدار باشد. وقتی دو متغیر سمت راست از نوع‌های متفاوت باشند، مقدار محاسبه شده real است. اما وقتی دو متغیر سمت راست از یک نوع باشند، مقدار محاسبه شده از نوع آن متغیرهاست. نوع محاسبه شده در سمت راست عملیات انتساب باید با نوع متغیر سمت چپ عملیات انتساب یکسان باشد.

- با استفاده از گرامر مستقل از متن، این گرامر را به صورت زیر می‌نویسیم :

<assign> → <var> = <expr>
<expr> → <var> + <var> | <var>
<var> → A | B | C

مثال گرامر صفت

- حال برای متغیرهای این گرامر دو متغیر صفت در نظر می‌گیریم. نوع واقعی (actual-type) و نوع مورد انتظار (expected-type).
- نوع واقعی : هرکدام از متغیرهای <var> و <expr> در گرامر، یک صفت ساخته شده دارند که برای ذخیره نوع آنها (که int یا real است) به کار می‌رود. برای نماد غیر پایانی <var> صفت آن ذاتی است و برای نماد غیر پایانی <expr> صفت آن (که نوع داده‌ای آن است) از روی صفت فرزندان آن به دست می‌آید.
- نوع مورد انتظار : نوع مورد انتظار، یک صفت به ارث برده شده برای نماد غیر پایانی <expr> است. در واقع انتظار می‌رود نوع <expr> در یک عبارت انتساب، با نوع متغیر <var> یکسان باشد.

مثال گرامر صفت

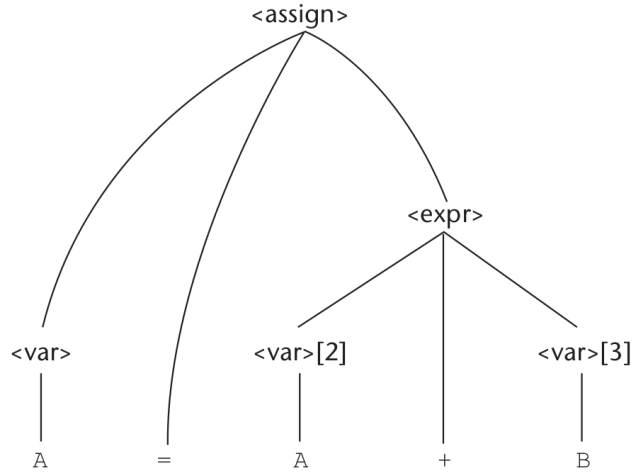
- گرامر صفت برای مثال قبلی را به صورت زیر می‌نویسیم.

1. Syntax rule : <assign> → <var> = <expr>
Semantic rule : <expr> . expected-type ← <var> . actual-type
2. Syntax rule : <expr> → <var> [2]+ <var> [3]
Semantic rule : <expr> . actual-type ← if (<var> [2] . actual-type = int)
and (<var> [3] . actual-type = int)
then int
else real
and if
Predicte : <expr> . actual-type == <expr> . expected-type
3. Syntax rule : <expr> → <var>
Semantic rule : <expr> . actual-type ← <var> . actual-type
Predicte : <expr> . actual-type == <expr> . expected-type
4. Syntax rule : <var> → A | B | C
Semantic rule : <var> . actual-type ← look-up (<var> . string)

- تابع look-up در واقع به ازای نام یک متغیر، نوع آن را باز می‌گرداند.

مثال گرامر صفت

- درخت تجزیه برای عبارت A = A + B در شکل زیر نشان داده شده است.

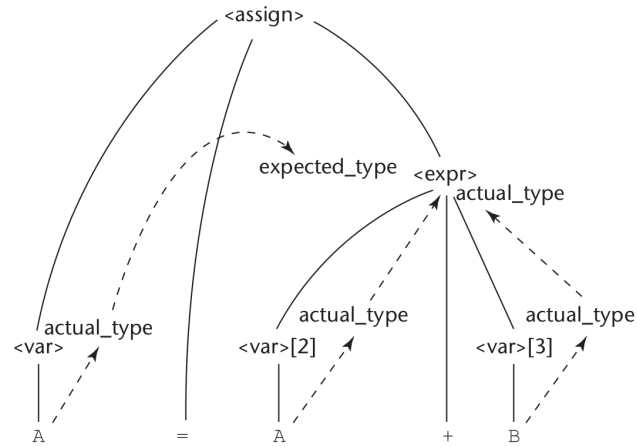


مثال گرامر صفت

- حال فرایند محاسبه صفت‌ها در یک درخت تجزیه را در نظر بگیرید. اگر همهٔ صفت‌ها، صفت‌های به ارث برده شده بودند، با پیمایش درخت از بالا به پایین می‌توانیم صفت همه رئوس را بدست آوریم. اگر همه صفت‌ها، صفت‌های ساخته شده بودند با پیمایش درخت از پایین به بالا می‌توانیم همه صفت‌ها را محاسبه کنیم. اما در واقع همیشه ترکیبی از صفت‌های به ارث برده شده و ساخته شده داریم پس پیمایش از هر دو طرف صورت می‌گیرد.

مثال گرامر صفت

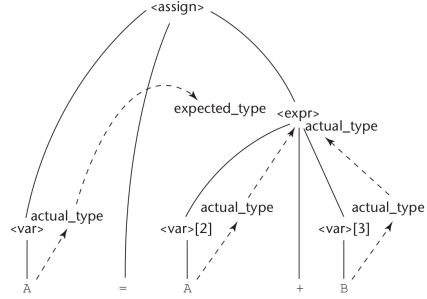
- شکل زیر نحوهٔ محاسبه صفت‌ها در درخت تجزیه را نشان می‌دهد. نوع واقعی (actual-type) یک صفت ساخته شده است، درحالی‌که نوع مورد انتظار (expected-type) یک صفت به ارث برده شده است.



مثال گرامر صفت

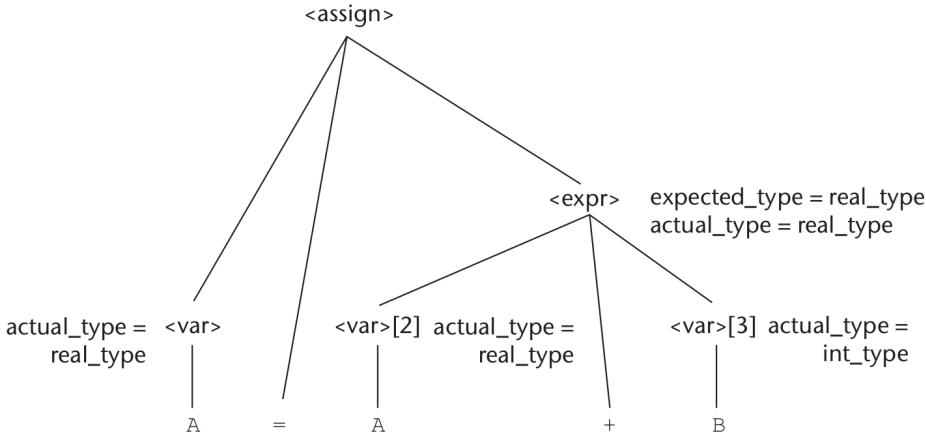
- برای محاسبه صفت‌ها در درخت تجزیه برای عبارت A = A + B داریم :

1. <var>. actual-type ← look-up (A) (Rule 4)
2. <expr>. expected-type ← <var>. actual-type (Rule 1)
3. <var> [2]. actual-type ← look-up (A) (Rule 4)
<var> [3]. actual-type ← look-up (B) (Rule 4)
4. <expr>. actual-type ← either int or real (Rule 2)
5. <expr>. expected-type == <expr>. actual-type is either true or false (Rule 2)



مثال گرامر صفت

- با فرض اینکه متغیر A از نوع real و متغیر B از نوع int باشد، درخت نهایی به صورت زیر خواهد بود.



معناشناسی پویا

- حال می‌خواهیم در مورد توصیف معنای برنامه صحبت کنیم، که به آن معناشناسی پویا [1] گفته می‌شود.
- از این پس منظور، از کلمه معناشناسی همان معناشناسی پویا است.
- توصیف معنای برنامه، به کاربران کمک خواهد کرد که معانی برنامه‌ها را بهتر متوجه شوند و همچنین به توسعه دهندگان کامپایلر کمک می‌کند تا بتوانند کامپایلر را به درستی پیاده سازی کنند و ابهامات و ناسازگاری‌های ممکن را رفع نمایند.
- اگر توصیف کاملی از نحو ومعنای برنامه وجود داشته باشد، آنگاه می‌توانیم ابزاری تولید کنیم که به طور خودکار کامپایلر تولید کند.
- معمولا برای توصیف معنا در یک زبان از زبان انگلیسی استفاده می‌شود که به دلیل غیر دقیق بودن، معمولاً یک توسعه دهنده کامپایلر باید در بسیاری موارد با آزمون و خطا یک کامپایلر را توسعه دهد و معمولاً برای زبان‌های رایج توصیف دقیقی وجود ندارد تا بتوان به طور خودکار کامپایلر آن را تهیه کرد. یکی از زبان‌هایی که برای معنای آن توصیف دقیقی داده شده است زبان اسکیم [2] است.

[1] dynamic semantic

[2] Scheme

معناشناسی عملیاتی

- در معناشناسی عملیاتی [1] معنای عبارات یک برنامه با استفاده از تأثیر اجرای آنها بر روی ماشین توصیف می‌شود.
- تأثیر بر روی ماشین به معنی دنباله‌ای از تغییرات بر روی حالت ماشین است و حالت ماشین مجموعه‌ای از مقادیر بر روی حافظهٔ آن است.

[1] operational semantics

معناشناسی عملیاتی

- اولین گام در ساختن معناشناسی عملیاتی ساختن زبانی میانی است که برای توصیف به کار رود. مهم‌ترین معیاری که برای این زبان باید در نظر گرفته شود وضوح [1] آن است.
- هر ساختاری در این زبان باید روشن و غیر مبهم باشد. نیاز به چنین زبانی به این دلیل است که زبان ماشین بسیار پیچیده و ناخوانا است و زبان مورد نظر برای توصیف ناشناخته است.

[1] clarity

معناشناسی عملیاتی

- برای مثال حلقه for در زبان سی را که به صورت زیر نوشته می‌شود.

```
for ( expr1 ; expr2 ; expr3 )  {  ... }
```

- می‌توانیم به صورت زیر توصیف کنیم.

```
            expr1;
loop :   if expr2 == 0 goto out;
         ...
         expr3;
         goto loop;
out :    ...
```

معناشناسی عملیاتی

- در چنین زبانی معمولا از ساختارهای سادهٔ زیر استفاده می‌کنیم.

```
id = var
id = id + 1
id = id - 1
goto  label
if var rel-op var goto label
```

- در اینجا از عملگرهای رابطه ای [1] مانند `=` , `<` , `>` , `=>` , `=<` , `!=` استفاده می‌کنیم.
- می‌توانیم این زبان را تعمیم دهیم و از عملگرهای حسابی ساده ریاضی مانند جمع و تفریق و ضرب و تقسیم و همچنین عملگرهای منطقی مانند و فصل و عطف و نقیض نیز استفاده کنیم.
- در معناشناسی عملیاتی برای توصیف یک زبان برنامه نویسی از یک زبان برنامه نویسی دیگر استفاده می‌کنیم. خواهیم دید که در روش‌های دیگر برای توصیف زبان می‌توانیم از زبان ریاضی استفاده کنیم.

[1] relational operator

معناشناسی دلالتی

- معناشناسی دلالتی [1] دقیق‌ترین و معروف‌ترین روش رسمی برای توصیف معنای برنامه‌هاست.
- این روش بر مبنای نظریه توابع گشتی است.
- توصیف معناشناسی دلالتی به طور کامل بسیار زمان بر است بنابراین در اینجا به قسمتی از آن و تعدادی مثال بسنده می‌کنیم.
- در توصیف معنای برنامه توسط معناشناسی دلالتی باید برای هر یک از ساختارهای برنامه یک ساختار ریاضی تعریف شود و توابعی تعریف شود که ساختارهای برنامه به ساختارهای ریاضی نگاشت کنند.
- به این روش معناشناسی دلالتی گفته می‌شود چرا که ساختارهای ریاضی دلیل استفاده مفاهیم در زبان را وصف می‌کنند.

[1] denotational semantics

معناشناسی دلالتی

- نگاشت ها در معناشناسی دلالتی مانند همهٔ نگاشت‌ها یک دامنه و یک برد دارند. به دامنهٔ این نگاشت‌ها دامنه نحوی [1] گفته می‌شود چرا که ساختارهای نحوی زبان را در بر می‌گیرند و به برد این نگاشت‌ها دامنه معنایی [2] گفته می‌شود.
- بنابراین در معناشناسی عملیاتی یک زبان را به یک زبان سطح پایین‌تر ترجمه می‌کنیم و در معناشناسی دلالتی زبان را به ساختارهای ریاضی ترجمه می‌کنیم.

[1] syntactic domain
[2] semantic domain

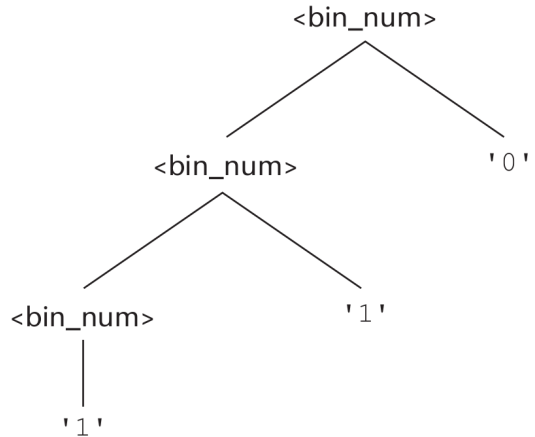
معناشناسی دلالتی

- در اینجا یک قسمت بسیار ساده از یک زبان را در نظر می‌گیریم و آن را توسط معناشناسی دلالتی توصیف می‌کنیم.
- گرامری را در نظر بگیرید که یک عدد دودویی را به صورت رشته تولید می‌کند.

<bin-num> → '0' | '1' | <bin-num> '0' | <bin-num> '1'

معناشناسی دلالتی

- درخت تجزیه برای جمله 110 از این گرامر در شکل زیر نشان داده شده است.



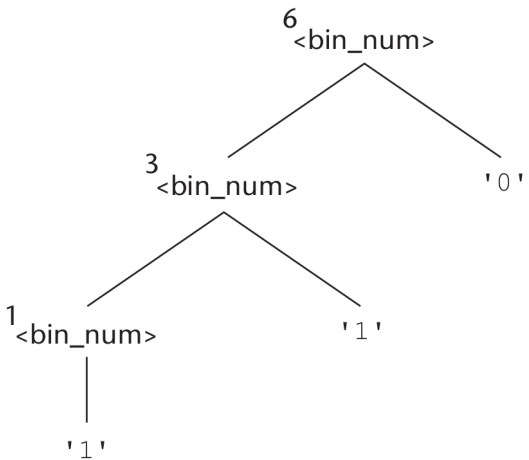
معناشناسی دلالتی

- دامنه نحوی در نگاشت معناشناسی دلالتی مجموعه همهٔ رشته‌های دودویی است. دامنه معنایی در این نگاشت مجموعه همه اعداد صحیح مثبت است که با N نمایش داده می‌شوند.
- اگر تابعی را تعریف کنیم که به ازای هر یک از قوانین گرامر با استفاده از مفهوم سمت چپ قانون معنی مفهوم سمت راست قانون را بیان کند، آنگاه می‌توانیم معنی متناظر با همه جملات زبان را به دست آوریم. در اینجا ساختار ریاضی که برای معنی جملات زبان به کار می‌بریم اعداد هستند.
- تابع نگاشت M_{bin} یک ساختار نحوی را با استفاده از قوانین گرامر به یک عدد نگاشت می‌کند.

M_{bin} ('0') = 0
M_{bin} ('1') = 1
M_{bin} (<bin-num> '0') = 2 * M_{bin} (<bin-num>)
M_{bin} (<bin-num> '1') = 2 * M_{bin} (<bin-num>) + 1

معناشناسی دلالتی

- با استفاده از این معناشناسی دلالتی، می‌توانیم معنای همه رئوس درخت تجزیه برای جمله '110' را تعیین کنیم.



معناشناسی دلالتی

- به طور مشابه می‌توانیم معنای گرامری که اعداد صحیح دهدهی تولید می‌کند را با استفاده از معناشناسی دلالتی توصیف کنیم.

<dec-num> → '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
| <dec-num> ('0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9')

- معنای این گرامر به روش معناشناسی دلالتی به صورت زیر است.

M_{dec} ('0') = 0 , M_{dec} ('1') = 1 ⋯ , M_{dec} ('9') = 9
M_{dec} (dec-num'0') = 10 * M_{dec} (<dec-num>)
M_{dec} (dec-num'1') = 10 * M_{dec} (<dec-num>) + 1
...
M_{dec} (dec-num'9') = 10 * M_{dec} (<dec-num>) + 9

معناشناسی دلالتی

- در معناشناسی دلالتی، همانند معناشناسی عملیاتی، از توصیف تغییر حالت برنامه استفاده می‌کنیم. به طور دقیق‌تر در معناشناسی دلالتی از تغییر مقادیر متغیرها به زبان ریاضی استفاده می‌کنیم.
- فرض کنید حالت یک برنامه مقادیر متغیرهای آن باشد. به عبارت دیگر داشته باشیم :

$S = \{ <i_1 , V_1> , <i_2 , V_2> , \cdots <i_n , V_n> \}$

- هر کدام از i ها یکی از متغیرها هستند که مقدار آن توسط V متناظر با آن نشان داده شده است. اگر متغیری مقدار نداشته باشد می‌توانیم از مقدار undef استفاده کنیم.
- فرض کنید تابع VARMAP مقدار یک متغیر در یک حالت دلخواه را باز می‌گرداند. به عبارت دیگر VARMAP (i_j , s) برابر است با V_j.

معناشناسی دلالتی

- حال فرض کنید گرامری برای عبارات محاسباتی در یک زبان برنامه نویسی داشته باشیم.

<dec-num> → <dec-num> | <var> | <binary-expr>
<binary-expr> → <left-expr> <operator> <right-expr>
<left-expr> → <dec-num> | <var>
<right-expr> → <dec-num> | <var>
<operator> → + | *

- تنها خطایی که می‌تواند وجود داشته باشد این است که مقداری تعریف نشده باشد. اگر مجموعه همه اعداد صحیح را Z و مقدار خطا را برابر با error در نظر بگیریم، آنگاه $Z \cup \{\text{error}\}$ دامنه معنایی برای عبارات این زبان است.

معناشناسی دلالتی

- در اینجا برای عملگر تساوی ریاضی از علامت $=\Delta$ استفاده می‌کنیم، چرا که علامت = معمولا برای انتساب مقدار در زبان برنامه نویسی استفاده می‌شود. همچنین علامت $\Rightarrow$ به معنی بازگرداندن مقدار یا نتیجه عبارت است.

معناشناسی دلالتی

M_e(<expr> , s) Δ = case <expr> of
<dec-num> $\Rightarrow$ M_{dec} (<dec-num> , s)
<var> $\Rightarrow$ if VARMAP (<var> , s) == undef
then error
else VARMAP (<var> , s)
<binary-expr> $\Rightarrow$
if (M_e (<binary-expr>.<left-expr>, s))== undef OR
M_e (<binary-expr>.<right-expr>, s) == undef
then error
else if (<binary-expr>.<operator> == '+')
then M_e (<binary-expr>.<left-expr> , s) +
M_e (<binary-expr>.<right-expr> , s)
else M_e (<binary-expr>.<left-expr> , s) *
M_e (<binary-expr>.<right-expr> , s)

- مقدار محاسبه شده برای M_e مقدار یک عبارت را محاسبه می‌کند و اگر بخواهیم معنی یک عبارت انتساب را بیان کنیم باید حالت جدید ماشین را محاسبه کنیم.

$$
\begin{aligned}
M_a\,(x = E\,,\, s)\;\Delta = {} & \text{if } M_e\,(E\,,\, s) == \text{error} \\
& \text{then error} \\
& \text{else } s' = \{\, \langle i_1\,,\, v_1' \rangle, \langle i_2\,,\, v_2' \rangle, \cdots, \langle i_n\,,\, v_n' \rangle \,\}\,, \text{ where} \\
& \quad \text{for } j = 1,\, 2,\, \cdots,\, n \\
& \quad\quad \text{if } i_j == x \\
& \quad\quad\quad \text{then } v_j' = M_e\,(E\,,\, s) \\
& \quad\quad\quad \text{else } v_j' = \text{VARMAP}(i_j,\, s)
\end{aligned}
$$

- دقت کنید که $i_j == x$ مقایسه اسامی است نه مقایسه مقادیر.

معناشناسی دلالتی

- تلاش‌های بسیاری برای تعریف معنای زبان‌های برنامه نویسی توسط معناشناسی دلالتی صورت گرفته است برای اینکه یک کامپایلر از روی تعریف نحو و معنا به طور خودکار ساخته شود اما این تلاش‌ها به نتیجه‌ای نرسیده است. با این وجود تعریف زبان به وسیله معناشناسی دلالتی تعریف دقیقی از زبان به دست می‌دهد و همچنین پیچیده شدن بیش از حد تعاریف نشان از این خواهد بود که زبان برنامه نویسی مورد توصیف برای کاربران نیز پیچیده خواهد شد.

معناشناسی اصل موضوعی

- معناشناسی اصل موضوعی [1] به این نام خوانده می‌شود چرا که بر مبنای منطق ریاضی [2] و استنتاج گزاره‌ها بر اساس اصول موضوع [3] است.
- در معناشناسی اصل موضوعی معنی یک برنامه بر اساس ارتباط متغیرها و ثابت‌های آن مشخص می‌شود.

[1] Axiomatic semantics
[2] mathematical logic
[3] axioms

معناشناسی اصل موضوعی

- در معناشناسی اصل موضوعی به ازای عبارت S ، یک پیش شرط P و یک پس شرط Q می‌نویسیم.
{P} S {Q}
- اگر بخواهیم اثبات کنیم یک برنامه درست است، باید اثبات کنیم به ازای مجموعه‌ای از ورودی‌ها خروجی‌های مناسب تولید می‌شوند.
- اگر به ازای پیش شرط یک برنامه پس شرط برنامه در محدودهٔ مقادیر مورد انتظار نباشد، برنامه درست نیست.

معناشناسی اصل موضوعی

- به عنوان مثال عبارت زیر را در نظر بگیرید.

$$x = x + y - 3 \ \{ \ x > 10 \ \}$$

- می‌توانیم پیش شرط را بدین صورت محاسبه کنیم :

$$x + y - 3 > 10 \ , \ y > 13 - x$$

- بنابراین می‌نویسیم :

$$\{ \ y > 13 - x \ \} \ x = x + y - 3 \ \{ \ x > 10 \ \}$$

معناشناسی اصل موضوعی

- فرض کنید می‌خواهیم درستی برنامه زیر را اثبات کنیم.

```
{ x = A AND y=B } t = x ; x = y ; y = t ; { x = B AND y = A }
```

- از آخرین دستور شروع می‌کنیم و پیش شرط دستور سوم را محاسبه می‌کنیم. این پیش شرط برابر است با

```
{ x = B AND t = A }
```

- سپس پیش شرط دستور سوم را به عنوان پس شرط دستور دوم در نظر می‌گیریم. پیش شرط دستور دوم برابر خواهد بود با

```
{ y = B AND t = A }
```

- سپس پیش شرط دستور دوم را به عنوان پس شرط دستور اول در نظر می‌گیریم. پیش شرط دستور اول برابر خواهد بود با

```
{ y = B AND x = A }
```

- بنابراین این برنامه درست است.

معناشناسی اصل موضوعی

- برای همهٔ ساختارهای یک برنامه از جمله دستورات شرطی و حلقه‌ها می‌توانیم پیش‌شرط و پس‌شرط‌ها را بر اساس گزاره‌های منطقی محاسبه کنیم که به علت طویل بودن محاسبات در اینجا از آنها صرف نظر می‌کنیم.

متغیرها و نوع‌های داده‌ای

نام‌ها

- در این فصل در مورد نام‌ها و متغیرها و کلمات کلیدی در زبان‌های برنامه نویسی صحبت خواهیم کرد.
- سپس در مورد انواع متغیرها و حوزه‌های تعریف صحبت می‌کنیم.

نام‌ها

- یک متغیر در واقع یک مفهوم انتزاعی [1] برای یک سلول حافظه است. در برخی مواقع متغیر دقیقا همان مقداری را دارد که در خانهٔ حافظه قرار می‌گیرد، مانند یک متغیر عدد صحیح و گاهی مقادیر ذخیره شده در متغیر باید به نحوی در حافظه نگاشت شوند چرا که مقدار متغیر با آنچه در حافظه ذخیره می‌شود متفاوت است، مانند یک آرایه از کاراکترها.
- از مهم‌ترین ویژگی‌های یک متغیر می‌توان به حوزه تعریف [2] و طول عمر [3] آن اشاره کرد.

[1] abstraction
[2] scope
[3] lifetime

نام‌ها

- یک نام [1] یا شناسه [2] رشته ای است که برای تمیز دادن یک موجودیت (مانند تابع یا متغیر) در یک برنامه به کار می‌رود.
- هر زبان محدودیتی بر روی نام‌ها دارد. معمولاً در بیشتر زبان نام‌ها باید با حرف شروع شوند و نام‌ها حساس [3] به بزرگ و کوچک‌اند.
- معمولا اسامی کلیدی زبان را نمی‌توان به عنوان نام انتخاب کرد. زبان کوبول تعداد کلمات کلیدی زیادی دارد مانند LENGTH و COUNT که منجر به محدودیت برای برنامه نویس می‌شود.
- در زبان سی ++، برای نام‌ها می‌توان فضای نام [4] تعریف کرد که باعث می‌شود یک نام را بتوان در چند مکان متفاوت به کار برد.

---

[1] name
[2] identifier
[3] case sensitive
[4] name space

متغیرها

- یک متغیر را می‌توان با چند ویژگی مشخص کرد : نام، آدرس، مقدار، نوع، طول عمر و حوزهٔ تعریف.
- آدرس یک متغیر در واقع آدرس حافظه‌ای در ماشین است که به آن متغیر اختصاص داده شده است. در طول اجرای یک برنامه یک متغیر ممکن است چندین بار تخصیص داده شده و آزاد شود و در هر بار تخصیص، آدرس آن می‌تواند متفاوت باشد. مثلا متغیر محلی در یک تابع در هر بار فراخوانی یک آدرس جدید می‌گیرد.

متغیرها

- آدرس متغیر گاهی مقدار چپ [1] نامیده می‌شود چون وقتی متغیر در سمت چپ یک عبارت باشد به آدرس آن نیاز داریم.
- ممکن است چند متغیر یک آدرس واحد داشته باشند که در اینصورت به اسامی دیگری که به یک آدرس واحد اشاره می‌کنند نام مستعار [2] گفته می‌شود. در زبان سی++، با استفاده از اشاره‌گرها و متغیرهای مرجع می‌توان نام مستعار تعریف کرد.
- یک متغیر همچنین دارای یک نوع [3] است که محدوده مقادیری که در آن می‌توانند قرار بگیرند و عملگرهایی که بر روی آن متغیر می‌توانند اعمال شوند را تعیین می‌کند.
- مقدار یک متغیر محتوایی است که در آن آدرس حافظه متناظر با آن ذخیره شده است. مقدار یک متغیر را گاهی مقدار راست [4] می‌نامیم زیرا هر گاه متغیر در سمت راست قرار گیرد به مقدار آن نیاز داریم.

[1] l-value
[2] alias
[3] type
[4] r-value

انقیاد

- انقیاد [1] به معنی پیوند دادن یک موجودیت با ویژگی آن است. برای مثال پیوند دادن نوع یک متغیر با آن متغیر انقیاد نوع، و انتساب مقدار به یک متغیر انقیاد مقدار نامیده می‌شود.
- زمانی را که انقیاد صورت می‌گیرد، زمان انقیاد [2] می‌گویند.
- برای مثال نماد ستاره (*) در زمان طراحی زبان [3] به عمل ضرب مقید شده است.
  نوع یک متغیر به مقادیر ممکن آن زمان پیاده سازی زبان [4] مقید شده است.
  یک متغیر به نوع آن در زبان جاوا در زمان کامپایل [5] مقید شده است.
  و در نهایت یک متغیر به سلول حافظه در زبان جاوا در زمان اجرا [6] مقید می‌شود.

---

[1] binding
[2] binding time
[3] language design time
[4] language implementation time
[5] compile time
[6] run time or execution time

انقیاد

- یک انقیاد ایستا [1] نامیده می‌شود اگر قبل از اجرای برنامه رخ دهد و در زمان اجرای برنامه بدون تغییر باقی بماند. اما اگر انقیاد در زمان اجرا صورت بگیرد و قابل تغییر باشد، به آن انقیاد پویا [2] می‌گوییم.

[1] static binding
[2] dynamic binding

انقیاد

- انقیاد نوع نیز می‌تواند ایستا یا پویا باشد. انقیاد نوع در جاوا ایستا و در پایتون پویا است.
- متغیرها همچنین می‌توانند به دو صورت تعریف شوند : صریح [1] و ضمنی [2].
- در تعریف صریح، نوع متغیر تعیین می‌شود، اما در تعریف ضمنی نوع متغیر در اولین مقداردهی به متغیر تعیین می‌شود.
- در بیشتر زبان‌های قدیمی مانند جاوا انقیاد نوع ایستا و تعریف متغیر به صورت صریح است.

[1] explicit declaration
[2] implicit declaration

انقیاد

- در برخی از زبان‌ها نام متغییر تعیین کننده نوع آن نیز هست. مثلا در زبان پرل متغیرهای اسکالر [1] با علامت $ و آرایه‌ها [2] با @ آغاز می‌شوند.
- تعریف ضمنی می‌تواند برای انقیاد ایستا نیز صورت بگیرد. مثلا در زبان سی++ با استفاده از کلمه auto می‌توان یک متغیر را به طور ضمنی تعریف کرد :

auto a = 12 ; auto b = " blue" ;

- در اینجا a از نوع int و b از نوع string در زمان کامپایل به صورت انقیاد ایستا تعیین می‌شود.

[1] scalar
[2] arrays

انقیاد

- در انقیاد نوع پویا، نوع متغیر در زمان اجرا با اولین انتساب مقدار به آن تعیین می‌شود. در چنین مواردی انقیاد متغیر به مکان حافظه نیز باید به صورت پویا باشد چون هر نوع مقدار حافظه متفاوتی اشغال می‌کند. همچنین در انقیاد پویا، نوع متغیر نیز در زمان اجرا می‌تواند تغییر کند.
- انقیاد پویا باعث می‌شود برنامه انعطاف بیشتری داشته باشد. برای مثال فرض کنید برنامه‌ای توسط پایتون نوشته ایم که مقادیری را از ورودی می‌خواند و محاسباتی را انجام می‌دهد. بسته به نوع مقادیر ورودی نوع متغیرها در این برنامه می‌توانند تغییر کنند، اما در یک برنامه سی اگر نوع متغیرها صحیح تعریف شده باشند نمی‌توان ورودی اعشاری به برنامه داد.

انقیاد

- در زبان لیسپ که یک برنامه تابعی قدیمی است انقیاد پویا است اما در بیشتر زبان‌های قدیمی انقیاد به صورت ایستا بوده است. در زبان‌های پایتون، روبی، جاوا اسکریپت و پی‌اچ‌پی انقیاد به صورت پویا است. برای مثال در پایتون می‌توانیم بنویسیم :

```
۱ a = [12.2 , 19.5]
۲ a = "hello"
```

- بدین صورت a ابتدا از نوع لیست تعریف می‌شود و سپس نوع آن به رشته تغییر پیدا می‌کند.
- در زبان روبی که یک زبان شیء‌گرای خالص است، همهٔ متغیرها ارجاعی و از نوع شیء عمومی هستند و به کلاس‌های مختلف ارجاع می‌دهند.

انقیاد

- سه نقطه ضعف برای انقیاد پویا وجود دارد :
۱. خطاهای نوع نمی‌توانند در زمان کامپایل تعیین شوند بنابراین برنامه‌ها در زبان‌ها با انقیاد پویا کمتر قابل اعتماد هستند.
۲. ممکن است به خاطر خطای برنامه نویس نوع متغیر تغییر کند در حالی که برنامه نویس انتظار نداشته است نوع تغییر کند.
۳. انقیاد پویا هزینهٔ زمانی دارد، زیرا بررسی نوع در زمان اجرا باید صورت بگیرد که باعث کندی برنامه می‌شود.
- معمولا زبان‌های با انقیاد پویا توسط مفسر پیاده سازی می‌شوند. یک کامپایلر نمی‌توان کدی را ترجمه کند که در آن نوع‌ها نامشخص هستند.

انقیاد

- فضایی در حافظه که به یک متغیر مقید می‌شود از مخزنی از حافظه‌های موجود گرفته می‌شود. این فرایند را تخصیص حافظه [1] می‌نامیم و فرایند آزاد سازی حافظه [2] فرایند گرفتن سلول حافظه از متغیر و بازگرداندن آن به مخزن فضاهای موجود در حافظه است.
- طول عمر [3] یک متغیر مدت زمانی است که در آن متغیر به فضای حافظه مقید شده است.

[1] memory allocation
[2] memory deallocation
[3] lifetime

انقیاد

- متغیرها را از لحاظ طول عمر به چهار دسته می‌توان تقسیم کرد : (۱) ایستا [1] ، (۲) پویا در پشته [2] ، (۳) پویا در هیپ به طور صریح [3] ، (۴) پویا در هیپ به طور ضمنی [4]

[1] static
[2] stack-dynamic
[3] explicit heap-dynamic
[4] implicit heap-dynamic

انقیاد

- یک متغیر ایستا [1] متغیری است که به سلول حافظه قبل از اجرای برنامه مقید شده باشد و تا وقتی که برنامه خاتمه یابد به همان سلول حافظه مقید بماند.
- یکی از کاربردهای متغیر ایستا زمانی است که بخواهیم متغیری در یک تابع یا کلاس مقدار خود را پس از خروج از تابع یا پس از تخریب اشیا از دست ندهد. این متغیرها در قسمت داده‌ها [2] در حافظه تخصیص داده می‌شوند.
- یکی از مزیت‌های استفاده از متغیر ایستا سرعت دسترسی به آن است. در زبان سی++ برای متغیرهای ایستا از کلمهٔ کلیدی `static` استفاده می‌کنیم.
- وقتی متغیر ایستا در یک کلاس جاوا یا سی++ تعریف شود، آن متغیر متعلق به کلاس است نه اشیای آن کلاس.

[1] static variable
[2] data segment

انقیاد

- یک متغیر پویا در پشته [1] متغیری است که انقیاد حافظه آن در زمان اجرا در لحظه تعریف متغیر رخ می‌دهد.
- این متغیرها در فضای پشتهٔ [2] برنامه تخصیص داده می‌شوند.
- یکی از مزیت‌های استفاده از متغیرهای پویا در پشته، استفاده از آنها در توابع بازگشتی است. در هر فراخوانی یک تابع بازگشتی، همهٔ متغیرهای تابع در فضای پشته کپی می‌شوند. مزیت دیگر این متغیرها این است که هر تابع یا کلاس، متغیرهای خود را در فضای پشته خود تعریف می‌کند که باعث امنیت بیشتر برنامه می‌شود.
- سرعت تخصیص فضا برای متغیرهای پویا نسبت به متغیرهای ایستا کمتر است.
- در زبان جاوا و سی++ همهٔ متغیرها به طور پیش فرض متغیر پویا هستند که در پشته تعریف می‌شوند.

[1] stack-dynamic variable
[2] stack

انقیاد

- یک متغیر صریح پویا در هیپ [1] خانهٔ حافظه بدون نام است که فضای حافظه در آن به طور صریح توسط برنامه نویس تخصیص داده شده و آزاد می‌شود.
- این متغیر در فضایی در حافظه به نام هیپ [2] قرار می‌گیرند، که تنها توسط اشاره‌گر و مرجع قابل دسترسی هستند.
- فضای هیپ فضایی در حافظه است که ساختار آن کاملا نامنظم است به دلیل اینکه فضای بیشتری را در حافظه اشغال می‌کند.
- در زبان سی++، این متغیرها توسط عملگر `new` تخصیص و توسط عملگر `delete` آزاد می‌شوند.

[1] explicit heap-dynamic variable
[2] heap

انقیاد

- وقتی فضای هیپ تخصیص داده می‌شود، آدرس آن بازگردانده می‌شود. زمان انقیاد حافظه این متغیرها در هنگام اجرا است.
- در زبان جاوا همه متغیرها به جز متغیرهای اصلی، شیء هستند. همهٔ اشیا در جاوا متغیر پویا هستند که در هیپ تخصیص داده می‌شوند و توسط متغیر مرجع آنها قابل دسترسی هستند. در جاوا راهی برای آزاد سازی حافظه توسط برنامه نویس وجود ندارد، بلکه فضاها به طور خودکار آزادسازی می‌شوند.
- معمولا ساختارهای داده مثل لیست‌های پیوندی که به فضای زیادی نیاز دارند و مقدار حافظه مورد نیاز آنها از ابتدا نامعلوم است از متغیرهای پویا در هیپ استفاده می‌کنند.
- نقطه ضعف این متغیرها هزینه زمانی برای تخصیص حافظه و همچنین سختی آنها در مدیریت اشاره‌گرها و مرجع‌هاست.

انقیاد

- یک متغیر ضمنی پویا در هیپ [1] متغیری است که به طور ضمنی بدون دخالت برنامه نویس در زمان مقدار دهی، به حافظه مقید می‌شود.
- اگر این متغیر قبلاً نیز در برنامه استفاده شده باشد، در هر مقدار دهی جدید مجدداً به یک فضا در حافظه هیپ مقید می‌شود.
- به طور مثال در زبان پایتون با تعریف `Var = [2,3]` فضایی در حافظه هیپ تخصیص داده می‌شود و چنانچه در همان برنامه مجدداً با دستور `Var = 'hello'` یا `Var = [1,2]` مواجه شویم، فضای جدیدی در حافظه تخصیص داده می‌شود.
- مزیت این متغیر انعطاف پذیری آن جهت نوشتن برنامه‌هایی با نوع عمومی است. نقطه ضعف این روش هزینه بالای اجرا است.
- یک متغیر همچنین می‌تواند ثابت تعریف شود بدین معنی که مقدار آن در طول برنامه غیر قابل تغییر است. متغیرهای ثابت در سی++ با کلمه کلیدی `const` و در جاوا با `final` مشخص می‌شوند.

[1] implicit heap-dynamic variable

حوزهٔ تعریف

- حوزهٔ تعریف [1] یک متغیر محدوده‌ای از دستورات است که برای آنها آن متغیر قابل مشاهده [2] است.
- یک متغیر برای یک دستور قابل مشاهده است اگر آن دستور بتواند متغیر را مقداردهی یا مقدار گیری کند.
- قسمتی از یک برنامه که با یک علامت شروع و پایان مشخص شده است را یک بلوک [3] می‌گوییم.
- یک متغیر را برای یک بلوک محلی [4] می‌نامیم، اگر در آن بلوک از کد تعریف شده باشد. یک متغیر را برای یک بلوک غیر محلی [5] می‌نامیم اگر متغیر در آن بلوک از برنامه تعریف نشده، ولی قابل مشاهده باشد.

---

[1] scope
[2] visible
[3] block
[4] local
[5] nonlocal

حوزهٔ تعریف

- حوزهٔ تعریف به دو دستهٔ ایستا و پویا تقسیم می‌شود.
- برای پیدا کردن مقدار متغیری که حوزهٔ تعریف آن ایستا است، کامپایلر ابتدا در بلوک فعلی به دنبال مقدار متغیر می‌گردد. اگر متغیر یافت نشد، بلوک‌های پدر را برای پیدا کردن تعریف متغیر و مقدار آن جستجو می کند.
- اگر حوزهٔ تعریف متغیری پویا باشد، مقدار متغیر بستگی به اجرا و پشته فراخوانی پیدا می‌کند. آخرین مقداری که در پشتهٔ فراخوانی به آن متغیر داده شده است، مقداری است که برای آن متغیر در نظر گرفته می‌شود.

حوزهٔ تعریف

- در مثال زیر اگر حوزهٔ تعریف x ایستا باشد، آنگاه خروجی برنامه ۱۰ است، اما اگر حوزهٔ تعریف پویا باشد، خروجی برنامه ۲۰ خواهد بود.

```
۱  int x = 10;
۲  int f() {
۳      return x;
۴  }
۵  int g() {
۶      int x = 20;
۷      return f();
۸  }
۹  int main() {
۱۰     printf("%d \n", g());
۱۱     return 0;
۱۲ }
```

حوزهٔ تعریف

- حوزهٔ تعریف ایستا [1] اولین بار در زبان الگول معرفی شد و بیشتر زبان‌های دستوری [2] بعد از الگول این مفهوم را از الگول گرفته‌اند.
- دو دسته از زبان‌ها وجود دارند که در آنها حوزهٔ تعریف ایستا به کار می‌رود : زبان‌هایی که در آنها می‌توان زیر برنامه‌های تو در تو نوشت و زبان‌هایی که زیر برنامه‌ها نمی‌توانند در آن تو در تو باشند.
- در دسته اول زبان‌های جاوا اسکریپت و لیسپ معمولی و آدا و پایتون قرار دارند و در دسته دوم زبان‌هایی به سبک سی.

[1] static scope
[2] imperative languages

حوزهٔ تعریف

- در زبان‌هایی با حوزهٔ تعریف ایستا، وقتی با متغیری برخورد می‌کنیم ابتدا در همان بلوکی که متغیر استفاده شده به دنبال تعریف آن می‌گردیم. اگر متغیر در آن بلوک تعریف نشده بود یا به عبارت دیگر یک متغیر غیر محلی بوده آنگاه در بلوک پدر [1] به دنبال تعریف آن متغیر می‌گردیم و اگر در بلوک پدر متغیر تعریف نشده بود، در بلوک پدرپدر به دنبال آن می‌گردیم و این روند را ادامه می‌دهیم تا یا تعریف متغیر را بیابیم و یا خطای متغیر تعریف نشده را گزارش کنیم.

[1] parent block

حوزهٔ تعریف

- برنامه زیر را در زبان پایتون در نظر بگیرید :

```
۱  def func() :
۲      def sub1() :
۳          x = 7
۴          sub2()
۵      def sub2() :
۶          y = x
۷          print("y = ",y)
۸      x = 3
۹      sub1()
۱۰
۱۱ func()
```

- متغیر x در تابع sub2 تعریف نشده است بنابراین باید در بلوک پدر یعنی در تابع func به دنبال آن بگردیم. در این بلوک x برابر با ۳ قرار گرفته است. توجه کنید که تابع sub2 از متغیر x در تابع sub1 استفاده نمی‌کند زیرا این تابع پدر تابع sub2 نیست.

حوزهٔ تعریف

- برنامه زیر را در نظر بگیرید :

```
۱ void sub() {
۲     int count;
۳     ...
۴     while (...) {
۵         count++;
۶         ...
۷     }
۸ }
```

- متغیر count در بلوک تابع sub تعریف شده است و بلوک حلقه از این متغیر استفاده می‌کند. در واقع بلوک حلقه از متغیری استفاده می‌کند که در بلوک پدر تعریف شده و غیرمحلی است.

حوزهٔ تعریف

- حال برنامه زیر را در نظر بگیرید :

```
۱ void sub() {
۲     int count;
۳     ...
۴     while (...) {
۵         int count;
۶         count++;
۷         ...
۸     }
۹ }
```

- متغیر count که در حلقه while تعریف شده است، در بلوک حلقه استفاده می‌شود، اما متغیر count که در بلوک تابع sub تعریف شده در حلقه قابل مشاهده نیست، زیرا متغیری همنام در حلقه تعریف شده است.

حوزهٔ تعریف

- زبان جاوا تعریف نام‌های تکراری در بلوک‌های تو در تو را ممنوع کرده است، چرا که قابلیت اطمینان برنامه با مجاز کردن آنها پایین می‌آید.
- در زبان سی می‌توان با بازکردن آکولاد یک بلوک فرزند ایجاد کرد ولی در زبان پایتون این قابلیت وجود ندارد. همچنین در زبان پایتون گرچه حوزهٔ تعریف ایستا در توابع تو در تو استفاده می‌شود ولی در بلوک‌های تو در تو حوزهٔ تعریف ایستا استفاده نمی‌شود.

حوزهٔ تعریف

- کد زیر در زبان پایتون را در نظر بگیرید :

```
for i in range (2) :
     x = 5
     print(x)
print(x)
```

- گرچه متغیر x در بلوک for تعریف شده اما بیرون از بلوک نیز قابل مشاهده است.

حوزهٔ تعریف

- در زبان‌های تابعی حوزهٔ تعریف یک متغیر با استفاده از کلمه let تعیین می‌شود. برای مثال در زبان ام‌ال می‌توان به صورت زیر چند متغیر تعریف کرده و از آن متغیرها در یک عبارت استفاده نمود :

```
let
    val   top = a + b
    val   bottom = c - d
in
    top / bottom
end;
```

حوزهٔ تعریف

- در برخی از زبان‌ها وقتی متغیری در بلوکی تعریف می‌شود آن متغیر در همه بلوک قابل مشاهده است اما مقدار آن اگر بعد از استفاده تعریف شده باشد `undefined` است.
- برای مثال برنامه زیر در جاوا اسکریپت را در نظر بگیرید :

```
1 console.log(x)
2 var x = 10
```

- مقدار x برابر با undefined چاپ خواهد شد، ولی اگر x تعریف نشده باشد با پیام خطای مفسر رو به رو می‌شویم.

حوزهٔ تعریف

- برخی از زبان‌ها مانند سی و سی++ و پایتون اجازه می‌دهند که متغیرها در خارج از توابع و کلاس‌ها نیز تعریف شوند. این متغیرها را متغیرهای عمومی [1] می‌نامیم که توسط همهٔ توابع قابل مشاهده‌اند.
- در زبان سی و سی++، می‌توان علاوه بر تعریف [2] متغیرها، آن‌ها را اعلام [3] نمود.
- در زمان کامپایل، اعلام متغیر توسط برنامه نویس به کامپایلر اعلام میکند که متغیری از نوع داده‌ای اعلام شده در کد وجود دارد، اما این متغیر ممکن است هنوز تعریف نشده باشد. اعلام متغیرها معمولاً وقتی به کار می‌رود که یک فایل دیگر تعریف شده باشد و بخواهیم از آن در یک فایل دیگر استفاده کنیم. همچنین اگر متغیری بعد از استفاده از آن تعریف شده باشد، باید قبل از استفاده آن را اعلام کنیم. در زمان اجرا، با تعریف متغیر فضای حافظه به آن تخصیص داده می‌شود، اما با اعلام متغیر تنها انقیاد نوع صورت می‌گیرد.
- در زبان سی، یک متغیر را می‌توان توسط کلمهٔ کلیدی `extern` اعلام کرد.

[1] global variable
[2] define
[3] declare

حوزهٔ تعریف

- در زبان سی++ اگر یک متغیر عمومی و یک متغیر محلی هم نام باشیم، متغیر عمومی غیر قابل مشاهده است، اما می‌توان با استفاده از عملگر حوزهٔ تعریف [1] (::) به آن دسترسی پیدا کرد.
- برای مثال برای دسترسی به متغیر عمومی x در تابعی که متغیر x را تعریف کرده از عبارت ::x استفاده می‌کنیم.
- در زبان پایتون، می‌توانیم از یک متغیر عمومی در یک تابع استفاده کنیم، اما اگر متغیر در تابع دوباره تعریف شود، آن متغیر تبدیل به یک متغیر محلی می‌شود و متغیر عمومی غیر قابل مشاهده می شود. برای اعلام متغیر به عنوان یک متغیر عمومی در یک تابع از کلمهٔ `global` استفاده می‌کنیم.

[1] scope operator

حوزهٔ تعریف

- برنامه زیر را در نظر بگیرید :

```
day = " Monday "
def  today() :
       print (" Today is ", day)
today()
```

- این برنامه بدون خطا اجرا می‌شود و خروجی آن برابر است با " Today is Monday ".

- حال برنامه زیر را در نظر بگیرید :

```
۱ day = " Monday "
۲ def  today() :
۳        print (" Today is ", day)
۴        day = " Tuesday"
۵        print (" Tomorrow is ", day)
۶ today()
```

- در اینجا با پیام خطا رو به رو می‌شویم چرا که day به یک متغیر محلی در تابع today() تبدیل شده است و اولین دسترسی به آن بدون مقدار است.

حوزهٔ تعریف

- برای حل این مشکل باید متغیر day را درون تابع به صورت عمومی تعریف کنیم.

```
۱ day = " Monday "
۲ def  today() :
۳         global    day
۴         print (" Today is ", day)
۵         day = " Tuesday"
۶         print (" Tomorrow is ", day)
۷ today()
```

حوزهٔ تعریف

- متغیرها می‌توانند در توابع تو در تو نیز در پایتون تعریف شوند. اگر یک متغیر در یک تابع فرزند همنام یک متغیر در تابع پدر تعریف شده باشد و بخواهیم از متغیر تابع پدر استفاده کنیم، از کلمه کلیدی `nonlocal` استفاده می‌کنیم.
- متغیرهای عمومی می‌توانند خطرساز باشند چرا که توابع گوناگون آنها را تغییر می‌دهند و ممکن است طراح برنامه نتواند همهٔ حالت‌هایی که متغیر عمومی ممکن است در آنها تغییر کنند را در نظر بگیرد و این موجب خروجی نادرست در برنامه شود.

حوزهٔ تعریف

- در برخی از زبان‌ها مانند لیسپ، حوزهٔ تعریف می‌تواند پویا باشد بدین معنی که مقدار متغیر و حوزه تعریف آن بستگی به نحوه اجرا و ترتیب اجرای توابع پیدا می‌کند.
- برنامه زیر را در زبان لیسپ در نظر بگیرید :

```
۱ (setf    r    100)
۲ (defun    fun1(r)    (print  r)    (fun2))
۳ (defun    fun2()    (print  r))
```

- حال با اجرای (fun1 5) ابتدا مقدار ۵ و سپس مقدار ۱۰۰ چاپ می‌شود. این همان چیزی است که از حوزهٔ تعریف ایستا انتظار داریم.

حوزهٔ تعریف

- حال برنامه زیر را در نظر بگیرید :

```
(defparameter    x    100)
(defun    fun1(x)    (print  x)    (fun2))
(defun    fun2()    (print  x))
```

- توسط کلمهٔ کلیدی defparameter یک متغیر با حوزهٔ تعریف پویا تعریف می‌شود.
- این بار با اجرای (fun1 5) مقدار ۵ دو بار چاپ می‌شود.
- با فراخوانی تابع fun2 درون تابع fun1 متغیر x تبدیل به یک متغیر محلی می‌شود.

حوزهٔ تعریف

- یکی از معایب حوزه تعریف پویا این است که قابلیت اطمینان و خوانایی برنامه‌ها توسط آن پایین می‌آید، زیرا متغیرهای محلی در توابع ممکن است در توابع دیگر قابل مشاهده شوند.
- به همین دلیل حوزهٔ تعریف پویا در بسیاری از زبان‌های برنامه نویسی استفاده نمی‌شود.

نوع‌های داده‌ای

- در این فصل در مورد نوع‌های داده‌ای از جمله نوع‌های داده‌ای اصلی، آرایه‌ها، لیست‌ها و غیره صحبت خواهیم کرد.

- یک نوع داده‌ای [1] مجموعه‌ای از داده‌ها از جنس یکسان و عملگرهای ممکن برای انجام محاسبات بر روی آن داده‌ها را تعیین می‌کند. برنامه‌های کامپیوتری برای حل مسائل دنیای واقعی، باید بتوانند عناصر و اشیای دنیای واقعی را مدلسازی کنند و برای این مدلسازی نیاز به ساختارهای داده‌ای دارند. زبان‌های مختلف ساختارها و نوع‌های داده‌ای متفاوتی را ارائه می‌دهند. هرچه نوع‌های داده‌ای فراهم شده توسط یک زبان بیشتر باشند، مسائل را می‌توان ساده‌تر با استفاده از آن حل نمود.

- زبان‌های ابتدایی مانند کوبول و فورترن نوع‌های داده‌ای بسیار محدودی داشتند.

[1] data type

نوع‌های داده‌ای

- زبان پی‌ال [1] زبانی بود که نوع‌های داده‌ای بسیار متنوعی را ارائه کرد. از طرف دیگر سازندگان زبان الگول تصمیم گرفتند نوع داده‌ای را محدود کرده و عملگرهایی را برای ایجاد امکان ساخت نوع‌های داده‌ای متنوع ارائه کنند. بنابراین اولین بار در این زبان نوع‌های داده‌ای تعریف شده توسط کاربر [2] به وجود آمدند. به این ترتیب خوانایی برنامه بسیار بالا می‌رفت. همچنین با استفاده از این داده‌ها، تغییر دادن برنامه‌های بزرگ و پیچیده نیز آسان‌تر می‌شد و کامپایلر نیز می‌توانست برای نوع داده‌ای جدید بررسی نوع [3] انجام دهد.
- سامانهٔ نوع [4] در یک زبان برنامه نویسی تعریف می‌کند که نوع‌های مختلف را چگونه می‌توان مقدار دهی و به یکدیگر منتسب کرد.

[1] PL/I
[2] User-defined data types
[3] type checking
[4] type system

نوع‌های داده‌ای

- در کنار نوع‌های داده‌ای اصلی مانند اعداد صحیح و اعشاری و کاراکتر و غیره، دو نوع داده‌ای مهم که در بیشتر زبان‌ها وجود دارند، آرایه‌ها [1] و رکوردها [2] هستند. یک آرایه مجموعه‌ای از مقادیر هم‌جنس است، در حالی که یک رکورد مجموعه‌ای از نوع‌های داده‌ای غیرهم‌جنس است. در چنین زبان‌هایی می‌توان آرایه‌ای تعریف کرد از نمونه‌هایی از رکوردهای هم‌جنس و یا می‌توان رکوردهایی را تعریف کرد که یک یا چند عنصر از آنها آرایه باشند.
- لیست‌ها نوع داده‌ای دیگری هستند که می‌توانند شامل مقادیر غیر هم‌جنس باشند. در زبان‌های تابعی مانند لیسپ، لیست‌ها بسیار مورد استفاده بودند. در سال‌های اخیر لیست‌ها در زبان‌هایی مانند پایتون نیز پیاده سازی شده‌اند.
- عملگرهای تبدیل نوع، به عملگرهایی در یک زبان گفته می‌شود که یک نوع را به نوع دیگر تبدیل می‌کنند مثلاً در زبان سی عملگر ستاره نوع داده‌ای اصلی را به اشاره‌گر و عملگر براکت یک نوع داده‌ای را به آرایه تبدیل می‌کند.

[1] array
[2] record

نوع داده‌ای اصلی

- نوع داده‌هایی که خود از نوع داده‌های ساده‌تر تشکیل نشده‌اند را نوع داده‌های اصلی [1] می‌نامیم.
- برخی از زبان‌های برنامه نویسی ابتدا تنها نوع داده‌ای عددی داشتند. مهم‌ترین نوع دادهٔ عددی نوع عدد صحیح یا `integer` است.
- در زبان جاوا بسته به اندازه عدد صحیح مورد نیاز می‌توان از نوع‌های `byte, short, int, long` استفاده کرد.
- در برخی زبان‌ها مانند پایتون می‌توان اعداد صحیح با طول نامحدود تعریف کرد. چنین اعدادی توسط سخت افزار پشتیبانی نمی‌شوند بلکه نیاز به طراحی در سطح زبان برنامه نویسی می‌باشد. یک عدد صحیح طولانی را می‌توان با حرف L در زبان پایتون نشان داد.
- برخی از زبان‌ها مانند سی++ برای اعداد مثبت و منفی عملگر نوع تعریف کرده‌اند، بدین ترتیب می‌توان یک نوع داده‌ای شامل اعداد مثبت با استفاده از کلمه کلیدی `unsigned` تعریف کرد.

[1] primitive data types

نوع داده‌ای اصلی

- نوع داده‌ای ممیز شناور [1] برای نمایش اعداد گویا و حقیقی به کار می‌رود، با این تفاوت که به دلیل محدودیت حافظه این اعداد را می‌توان تنها با تقریب در حافظه ذخیره کرد. برای ذخیره این اعداد باید قسمت اعشاری عدد که بعد از ممیز مشخص می‌شود و همچنین مرتبه بزرگی عدد را که توسط مقدار توانی مشخص می‌شود را ذخیره سازی کرد.
- بیشتر زبان‌های برنامه نویسی دو نوع داده اعشاری با دقت متفاوت به نام `float` و `double` را پشتیبانی می‌کنند که اولی در چهار بایت و دومی در هشت بایت ذخیره می‌شود.
- در نوع داده‌ای `float` ، یک بیت برای علامت، ۸ بیت برای توان و ۲۳ بیت برای قسمت اعشاری به کار می‌رود. در نوع داده‌ای `double` که به معنای ممیز شناور با دقت دو برابر [2] است، یک بیت برای علامت، ۱۱ بیت برای توان و ۵۲ بیت برای قسمت اعشاری به کار می‌رود.

[1] floating-point
[2] double-precision floating-point

نوع داده‌ای اصلی

- برخی از زبان‌های برنامه نویسی مانند پایتون، نوع داده‌ای مختلط [1] را نیز پشتیبانی می‌کنند. برای مثال در پایتون می‌توان یک عدد مختلط را به صورت `(1 + 2j)` تعریف کرد.
- برخی از زبان‌ها مانند زبان کوبول که برای استفاده‌های تجاری به وجود آمده است و نیاز به نگهداری دقیق اعداد دهدهی اعشاری دارند، یک نوع داده‌ای ویژه به نام `decimal` پشتیبانی می‌کنند. مزیت این نوع داده‌ای این است که اعداد اعشاری را دقیق به همان صورتی که هستند ذخیره می‌کند چرا که اعداد ممیز شناور ممکن است در تبدیل دودویی به دهدهی وقتی را از دست بدهند. برای ذخیره سازی دقیق این نوع داده‌ای اعداد را مانند رشته ذخیره می‌کند.

[1] complex

نوع داده‌ای اصلی

- نوع داده‌ای بولی [1] برای ذخیرهٔ مقادیر درست [2] و نادرست [3] به کار می‌رود.
- در برخی زبان‌ها مانند سی که نوع داده‌ای بولی را پشتیبانی نمی‌کنند، عدد صفر معادل مقدار نادرست و اعداد غیر صفر معادل درست به کار می‌روند.
- در زبان سی++ نوع داده‌ای بولی با کلمه `bool` تعریف می‌شود، برای نوع داده‌ای بولی تنها به یک بیت نیاز است اما به دلیل اینکه دسترسی به یک بیت راندمان پایینی دارد، برای ذخیره آنها از یک بایت استفاده می‌شود.

[1] boolean
[2] true
[3] false

نوع داده‌ای اصلی

- نوع داده‌ای کاراکتر یا حرف [1] برای ذخیره حروف الفبا با یک کدگذاری مشخص به کار می‌رود. حروف استاندارد اسکی [2] به یک بایت برای ذخیره سازی نیاز دارند. برای حروف الفبا از زبان‌های مختلف می‌توان از استانداردهای یونیکد [3] از جمله UTF-۳۲ استفاده کرد که به چهار بایت فضا نیاز دارد.

[1] character
[2] ASCII (American Standard Code for Information Interchange)
[3] Unicode Consorsium

نوع داده‌ای رشته‌ای

- نوع داده‌ای رشتهٔ کاراکتری [1] یا رشته برای ذخیرهٔ دنباله‌ای از حروف به کار می‌رود. کاربرد این نوع داده‌ای ذخیره سازی کلمات، جملات و متون است.
- برخی از زبان‌ها مانند سی و سی++، رشته ها به صورت آرایه‌ای از حروف تعریف می‌شوند و توابعی در کتابخانه‌های جانبی برای اعمال عملگرهایی مانند الحاق [2] ، انتساب [3] و کپی زیر رشته [4] تعریف شده است.
- در برخی زبان‌ها مانند جاوا یا کتابخانه استاندارد سی++، رشته‌ها به صورت کلاس تعریف می‌شوند و عملگرها برای این کلاس‌ها سربارگذاری [5] شده‌اند.

[1] character string type
[2] concatenation
[3] assigniment
[4] substring
[5] overload

نوع داده‌ای رشته‌ای

- در برخی زبان‌ها مانند پایتون رشته‌ها به عنوان یک داده اصلی تعریف می‌شوند که همه عملگرهای مورد نیاز برای آنها تعریف شده است.
- در زبان‌هایی که طول رشته در آنها ثابت است، برای ذخیره سازی رشته تنها نیاز به آدرس شروع رشته در حافظه و طول رشته داریم. در زبان‌هایی که در طول رشته می‌تواند متغیر باشد، باید طول فعلی و طول ماکزیمم مشخص باشند.
- رشته‌هایی با طول متغیر را می‌توان توسط لیست‌های پیوندی ذخیره کرد. نقطه ضعف این روش پیچیدگی آن برای رشته‌های طولانی و در نتیجه راندمان پایین آن است. روش دیگر استفاده از آرایه‌ای از کاراکترهاست که نقطه ضعف آن محدودیت حافظه برای رشته‌های طولانی‌است. روشی که معمولاً استفاده می‌شود این است که رشته در یک آرایه نگهداری می‌شود و هنگامی که طول آرایه نیاز به افزایش داشت فضای جدیدی در هیپ با حافظه مورد نیاز تخصیص داده شده و رشته از مکان قبلی به مکان فعلی منتقل می‌شود.

نوع داده‌ای شمارشی

- نوع داده‌ای شمارشی [1] نوعی است که توسط آن می‌توان یک مقدار از بین چند مقدار نامگذاری شده را انتخاب کرد. به عبارت دیگر این نوع داده‌ای مقادیر ثابت نامگذاری شده [2] را تعریف می‌کند.

[1] enumeration type
[2] named constant

نوع داده‌ای شمارشی

- برای مثال در زبان سی می‌توان نوع داده‌ای `day` را به صورت
enum day { Mon, Tue, Wed, Thu, Fri, Sat, Sun } تعریف کرد. مقادیر نوع داده‌ای شمارشی معمولاً به صورت اعداد صحیح ذخیره می‌شوند. در زبان سی++ می‌توان بر روی داده‌های شمارشی عملگر نیز تعریف کرد و عملیات بر روی متغیرهای از نوع شمارشی اعمال کرد.
برای این کار باید از `enum class` استفاده کرد. همچنین با استفاده از `enum class` در زمان کامپایل می‌توان نوع دادهٔ شمارشی را بررسی کرد. اگر داشته باشیم :
enum class color = { red, blue, yellow } و enum class rgb = { red, green, blue }
آنگاه کامپایلر برای `rgb c = blue` پیام خطا صادر می‌کند و باید مشخص کرد این مقدار از چه نوعی است.

نوع داده آرایه

- نوع دادهٔ آرایه [1] برای نگهداری مقادیر داده‌ای هم نوع استفاده می‌شود، به طوری که هر مقدار در آرایه توسط مکان آن در آرایه نسبت به اول آرایه قابل دسترسی است.
- معمولا مقادیر آرایه توسط عملگر زیرنویس [2] یا اندیس [3] می‌توان دسترسی پیدا کرد.
- اندیس یک آرایه در بیشتر زبان‌ها از جمله سی++، جاوا و پایتون با براکت مشخص می‌شود.
- بسیاری از زبان‌ها بررسی دسترسی در آرایه [4] ندارند، اما جاوا بازهٔ دسترسی را بررسی می‌کند.

[1] array
[2] subscript operator
[3] index
[4] array range check

نوع داده آرایه

- چهار نوع مختلف از آرایه‌ها وجود دارند : (۱) آرایه‌های ایستا [1] ، (۲) آرایه‌های پویای ثابت روی پشته [2] ، (۳) آرایه‌های پویای ثابت بر روی هیپ [3] و (۴) آرایه‌های پویای بر روی هیپ [4].
- آرایه‌های ایستا قبل از اجرای برنامه اندازه معین دارند و قبل از شروع برنامه بر روی قسمت داده [5] حافظه مقید می‌شوند. در زبان سی++، این دسته از آرایه‌ها به صورت `static type name[N]` تعریف می‌شوند. به طوری که N یک عدد صحیح است.

[1] static arrays
[2] fixed stack-dynamic arrays
[3] fixed heap-dynamic arrays
[4] heap-dynamic arrays
[5] data segment

نوع داده آرایه

- آرایه‌های پویای ثابت بر روی پشته قبل از اجرای برنامه اندازه معین دارند و در هنگام تعریف بر روی پشتهٔ حافظه مقید می‌شوند. این آرایه‌ها در سی++ به صورت `type name[N]` تعریف می‌شوند.
- آرایه‌های پویای ثابت بر روی هیپ، قبل از اجرای برنامه اندازه معین ندارند و در زمان اجرا اندازه آنها تعیین و بر روی پشته مقید می‌شوند. وقتی این آرایه‌ها به حافظه مقید شدند اندازهٔ آنها غیر قابل تغییر است. این آرایه‌ها در زبان سی++ به صورت `type[] name = new type[n]` تعریف می‌شوند، به طوری که n یک متغیر یا ثابت است.
- آرایه‌های پویای بر روی هیپ، اندازه آنها در زمان اجرا تعیین می‌شود و همچنین انقیاد حافظه آنها در زمان اجرا صورت می‌گیرد. همچنین می‌توان چندین بار در زمان اجرا فضای حافظه آنها را آزاد و دوباره تخصیص داد. وکتورها در زبان سی++ در این دسته از آرایه‌ها قرار می‌گیرند. در زبان سی تخصیص حافظه بر روی هیپ در زمان اجرا با استفاده از دستور malloc و آزادسازی حافظه با استفاده از free انجام می‌شود. در سی++ نیز تخصیص با new و آزادسازی با delete صورت می‌گیرد.

نوع داده آرایه

- در زبان پایتون از نوع داده‌ای لیست می‌توان به عنوان آرایه استفاده کرد. می‌توان آرایه را به صورت زیر تعریف کرد و به عناصر آن مقدار افزود.

```
array = [1,2,3]
array.append(4)
```

در زبان سی و سی++ و جاوا، به طور پیش فرض بر روی آرایه‌ها عملگر تعریف نشده است، اما برای لیست‌ها در پایتون عملگرهایی تعریف شده است.

نوع داده آرایه

- مثلا عملگر + دو لیست را به یکدیگر الحاق می‌کند و توسط عملگر in می‌توان بررسی کرد آیا مقداری در آرایه وجود دارد یا خیر. عملگر == بررسی می‌کند آیا دو لیست از نظر اندازه و مقدار با یکدیگر برابرند یا خیر.

```
۱ a = [1,2]
۲ b = [3,4]
۳ c = a + b
۴ if 4 in c or a==b :
۵     print ("ok")
```

نوع داده آرایه

- در برخی از زبان‌ها مانند روبی، مفهومی به نام برش آرایه [1] وجود دارد که توسط آن می‌توان قسمتی از یک آرایه را به عنوان یک آرایه دیگر استفاده کرد.
- در برش آرایه، توسط عملگر : در داخل براکت می‌توان شروع و پایان برش را تعیین کرد. قطعه کد زیر چند مثال از برش آرایه‌ها آورده شده است.

```
۱ vector = [2,4,6,8,10,12,14,16]
۲ mat = [[1,2,3],[4,5,6],[7,8,9]]
۳ v1 = vector[3:6] # v1=[8,10,12]
۴ m1 = mat[0][0:2] # m1=[1,2]
۵ m2 = mat[:1] # m2=[[1,2,3]]
```

[1] slice of array

نوع داده آرایه

- به طور کل قوانین برش در پایتون به صورت زیر می‌باشند.

```
a[start : stop] # [a[start], ... , a[stop-1]]
a[start :] # [a[start], ... , a[len(a)-1]]
a[: stop] # [a[0], ... , a[stop-1]]
a[:] # [a[0], ... , a[len(a)-1]]
```

- همچنین می‌توان علاوه بر شروع و پایان اندیس برش، مقدار افزایش اندیس در هرگام برش را نیز به صورت زیر تعیین کرد.

```
a[start : stop : step] # itms from start index incremented
                       # by step not after stop -1
```

- به طور مثال :

```
v2 = vector [0:4:2] # v2 = [2,6]
```

نوع داده آرایه

- مقدار منفی در اندیس‌های آرایه و همچنین برش، به معنی شمارش از آخر است.

```
vector [-2] # 14
vector [-2 :] # [14,16]
vector [: -2] # [2,4,6,8,10,12]
```

- مقدار منفی در پارامتر سوم برش به معنی شمارش معکوس است.

```
vector [:: -1] # [16,14,12,10,8,6,4,2]
vector [1 :: -1] # [4,2]
vector [: -3 : -1] # [16,14]
```

نوع داده آرایه

- برای پیاده سازی آرایه در یک زبان برنامه نویسی نیاز به دسترسی به آدرس حافظه هر یک از عناصر آن داریم. آدرس یک سلول از حافظه را می‌توانیم با استفاده از رابطه زیر به دست آوریم :

address (array[k]) = address (array[0]) + k * element-size

- در زبان‌هایی که محدوده دسترسی اندیس را بررسی می‌کنند، کامپایلر نیاز به نگهداری اطلاعات مربوط به آدرس آرایه و نوع آرایه و اندازهٔ آرایه دارد ولی در صورتی که کامپایلر محدوده دسترسی را بررسی نکند، نیازی به نگهداری اندازه آرایه نیست.

- برای پیاده سازی آرایه‌های چند بعدی نیاز به محاسبه آدرس حافظه یک درایه برای دسترسی به آن را داریم، زیرا حافظه یک بعدی است. برای مثال در یک آرایهٔ دو بعدی که همهٔ سطرهای آن طول یکسان دارند، داریم :

address (m[i][j]) = address (m[0][0]) + (i * n + j) * element-size

آرایه‌های انجمنی

- یک آرایه انجمنی [1] مجموعه‌ای است از مقادیر که برای دسترسی به آنها از مقادیری به نام کلید استفاده می‌کنیم. به عبارت دیگر هر یک از عناصر یک رابطه انجمنی جفتی است که قسمت اول آن کلید و قسمت دوم آن مقدار نامیده می‌شود. برای دسترسی به یک مقدار باید از کلید مربوط به آن استفاده کرد.
- در آرایه‌های غیر انجمنی در واقع کلیدها، اندیس‌هایی هستند که مکان یک مقدار را در آرایه تعیین می‌کنند.
- آرایه‌های انجمنی در زبان‌ها برل، پایتون و روبی پیاده سازی شده‌اند.

[1] associative array

آرایه‌های انجمنی

- در زبان پایتون به آرایه‌های انجمنی، دیکشنری [1] گفته می‌شود.
- در زبان پایتون کلیدهای یک دیکشنری می‌توانند تنها رشته‌ها و اعداد باشند در حالی که در روبی کلیدها می‌توانند از هر نوع کلاسی باشند.
- برای پیاده سازی آرایه انجمنی در پرل، به ازای هر کلید یک مقدار هش [2] ۳۲ بیتی محاسبه می‌شود.
- در زبان پایتون یک متغیر از نوع دیکشنری به صورت زیر تعریف می‌شود.

```
۱ d = {1:'one', 'two': 2}
۲ d[1] #  'one'
۳ d['two'] # 2
```

[1] dictionary
[2] hash value

نوع داده‌ای رکورد

- یک رکورد [1] تعدادی متغیر که هر کدام می‌توانند از یک نوع متفاوت باشند را تجمیع می‌کند. هر عنصر از یک رکورد با یک نام و یک نوع مشخص می‌شود و مکان آن در حافظه نسبت به ابتدای رکورد با محاسبه اندازهٔ عناصر قبلی آن قابل محاسبه است.
- در سی و سی++، برای تعریف یک رکورد از نوع داده‌ای ساختمان یا استراکت [2] استفاده می‌شود.
- عناصر یک رکورد برخلاف آرایه که با اندیس مشخص می‌شوند با نام و نوع عنصر مشخص می‌شوند. هر عنصر یک رکورد، فیلد نامیده می‌شود که در بیشتر زبان‌ها با عملگر نقطه (.) قابل دسترسی هستند.

[1] record
[2] struct

نوع داده‌ای چندتایی

- یک نوع داده‌ای چندتایی [1] نوع داده‌ای است برای نگهداری مقادیر از انواع متفاوت. به عناصر چندتایی با اندیس یا شماره آنها در چندتایی می‌توان دسترسی پیدا کرد.
- بنابراین یک چندتایی از لحاظ این که به عناصرش با اندیس می‌توان دسترسی پیدا کرد شبیه آرایه است و تفاوت آن با آرایه این است که عناصر آن می‌توانند از نوع‌های متفاوت باشند.
- تفاوت دیگر چندتایی و آرایه این است که عناصر آرایه قابل تغییر [2] هستند، در حالی که عناصر چندتایی غیر قابل تغییر [3] اند.
- یک مورد استفاده از چندتایی وقتی است که می‌خواهیم تعدادی مقدار به تابعی دیگر ارسال کنیم ولی نمی‌خواهیم تابع بتواند مقادیر متغیر ارسال شده را تغییر دهد.

[1] tuple
[2] mutable
[3] immutable

نوع داده‌ای چندتایی

- در زبان پایتون چندتایی را با استفاده از پرانتز تعریف می‌کنیم.

```
t = (3 , 1.2 , 'hello')
t [1] # 1.2
t [0] = 2 # error
```

- عملگر + بر روی چندتایی تعریف شده است که دو چندتایی را با هم الحاق می‌کند.

نوع داده‌ای لیست

- لیست در اولین زبان تابعی یعنی لیسپ به وجود آمد و از اهمیت ویژه‌ای در همهٔ زبان‌های تابعی برخوردار است.
- لیست مجموعه‌ای است از عناصر با نوع‌های متفاوت به طوری که مقدار عناصر آن قابل تغییر هستند.
بنابراین لیست شبیه چندتایی است با این تفاوت که مقادیر عناصر آن را می‌توان تغییر داد و شبیه آرایه است با این تفاوت که نوع عناصر آن ممکن است یکسان نباشد.
- در زبان پایتون یک لیست به صورت زیر تعریف می‌شود :

```
۱ l = [1 , 2.3 , 'apple' ]
۲ l [1] = 'grape'
```

- یک عنصر از لیست را می‌توان توسط عملگر del حذف کرد.

نوع داده‌ای لیست

- یک متغیر از نوع چندتایی را می‌توان توسط تابع list به لیست تبدیل کرد. همچنین یک متغیر از نوع لیست را می‌توان توسط تابع tuple به چندتایی تبدیل نمود.
- پایتون روشی مختصر برای توصیف لیست ارائه می‌کند که روش شمول کامل [1] نامیده می‌شود.

```
# [expression for var in list if condition]
a = [x * x  for  x  in range(12)  if  x % 3  ==  0]
# a = [0 , 9 , 36 , 81]
```

[1] list comprehension

نوع داده‌ای اجتماع

- نوع داده‌ای اجتماع [1] ، نوعی است که متغیر آن در زمان‌های متفاوت می‌تواند نوع‌های متفاوت داشته باشد. برای مثال فرض کنید به متغیری نیاز داشته باشیم که گاهی در آن عدد صحیح قرار می‌گیرد و گاهی عدد اعشاری. اگر بخواهیم دو متغیر تعریف کنیم در هر بازهٔ زمانی یکی از آنها بدون استفاده می‌ماند. اگر این دو متغیر را در یک اجتماع قرار دهیم برای هر دوی آنها یک فضای حافظه برابر با حافظه مورد نیاز برای متغیر بزرگ‌تر تخصیص داده می‌شود.

[1] union

نوع داده‌ای اجتماع

- در سی و سی++ این نوع با کلمه union تعریف می‌شود.

```
union Number {
      int i ;
      float f ;
};
union Number n;
n.i = 2
n.f = 3.1 // n.i is erased
```

- در پیاده سازی اجتماع همه عناصر آن یک آدرس حافظه می‌گیرند.

نوع‌های اشاره‌گر و مرجع

- یک اشاره‌گر [1] نوعی است که متغیر آن آدرس یک سلول از حافظه را نگهداری می‌کند. همچنین مقدار آن می‌تواند تهی باشد که در این صورت به هیچ مکانی در حافظه اشاره نمی‌کند.

- اشاره‌گرها استفاده‌های متعددی دارند. یکی از موارد استفاده آنها در فراخوانی توابع است. ارسال آدرس متغیرها به توابع به جای ارسال مقدار آنها موجب بهبود سرعت اجرای برنامه می‌شود. مورد استفاده دیگر اشاره‌گرها تخصیص حافظه پویا در فضای هیپ است. به فضای حافظهٔ تخصیص داده شده توسط یک اشاره‌گر می‌توان دسترسی پیدا کرد.

- برای مثال فرض کنید بخواهیم یک لیست پیوندی بسازیم که تعداد عناصر آن مشخص نباشد. به ازای هر عنصر در لیست پیوندی باید یک فضای جدید در حافظه هیپ تخصیص دهیم به طوری که هر عنصر لیست به عنصر بعدی خود اشاره می‌کند.

- یک متغیر از نوع اشاره‌گر یک آدرس را نگهداری می‌کند و در زبان سی و سی++ می‌توان توسط عملگر ستاره (*) به مقدار یک مکان حافظه دسترسی پیدا کرد.

[1] pointer

نوع‌های اشاره‌گر و مرجع

- زبان‌هایی که نوع اشاره‌گر را پیاده سازی می‌کنند به یک عملگر یا تابع برای تخصیص حافظه پویا و انتساب آن به اشاره‌گر نیاز دارند که این عملگر در زبان سی++ توسط کلمهٔ new پیاده‌سازی شده است. همچنین با استفاده از عملگر delete می‌توان حافظهٔ تخصیص داده شده را آزاد کرد.
- عملگری که برای دریافت مقدار مکان حافظه‌ای که توسط اشاره‌گر قابل دسترسی است استفاده می‌شود عملگر رفع ارجاع [1] نام دارد. این عملگر در زبان سی++ توسط توسط ستاره (*) پیاده سازی شده است.
- عملگر مورد نیاز دیگر، جهت دریافت آدرس یک متغیر برای ذخیره‌سازی آدرس در اشاره‌گر است. در زبان سی++ با استفاده از عملگر امپرسند (&) می‌توان آدرس یک متغیر را به دست آورد.
- اولین زبانی که نوع اشاره‌گر را پیاده سازی کرد زبان پی‌ال۱ بود.

[1] dereference

نوع‌های اشاره‌گر و مرجع

- اشاره‌گرها می‌توانند خطراتی را نیز به همراه داشته باشند که قابلیت اطمینان برنامه را پایین می‌آورند.
- برای مثال اگر اشاره‌گری به یک فضای حافظه در هیپ اشاره کند و فضا توسط دستوری آزاد شود ولی اشاره‌گر همچنان به آن فضای حافظه اشاره کند، اشاره‌گری داریم که مقدار ناصحیح دارد و در برخی زبان‌ها ممکن است دسترسی به این اشاره‌گر معلق [1] موجب توقف برنامه شود. همچنین اگر در همان فضای حافظه مجدداً حافظه تخصیص داده شود، اشاره‌گر مذکور ممکن است مقدار ناصحیح از حافظه بخواند.
- مشکل دیگر اشاره‌گرها این است که ممکن است برنامه نویس بدون آزاد سازی فضای حافظه‌ای که یک اشاره‌گر به آن اشاره می‌کند، اشاره‌گر را به مکان دیگری اشاره دهد. در این صورت مکان حافظه در هیپ غیر قابل دسترسی می‌شود. به این پدیده نشست حافظه [2] گفته می‌شود که ممکن است پس از انباشته شده زیاد مکان‌های تخصیص داده شده موجب پر شدن حافظه و توقف برنامه شود.
- همچنین اگر چند اشاره‌گر به یک فضای حافظه اشاره کنند، ممکن است به اشتباه بخواهیم یک فضای حافظه را چند بار آزاد کنیم.

[1] dangling pointer
[2] memory leakage

نوع‌های اشاره‌گر و مرجع

- نوع داده‌ای مرجع شبیه اشاره‌گر است با این تفاوت که یک متغیر مرجع آدرس نگهداری نمی‌کند بلکه نام مستعاری است برای یکی از خانه‌های حافظه.
- بنابراین متغیر مرجع وقتی برای اولین بار به خانه‌ای در حافظه اشاره کرد، در طول برنامه فقط به همان خانهٔ حافظه می‌تواند اشاره کند.
- در زبان سی++ می‌توان یک متغیر مرجع را با استفاده از عملگر امپرسند [1] (&) به صورت زیر تعریف کرد.

```
۱ int x = 2;
۲ int &r = x;
۳ r++; // r = 3 and x = 3
```

[1] ampersand

نوع‌های اشاره‌گر و مرجع

- نوع داده‌ای مرجع برای ارسال متغیر به یک تابع استفاده می‌شود هنگامی که بخواهیم مقدار آن متغیر توسط تابع تغییر کند.

```
۱ void swap (int& x, int& y) {
۲     int tmp = x;
۳     x = y;
۴     y = tmp;
۵ }
```

نوع‌های اشاره‌گر و مرجع

- مورد استفاده دیگر وقتی است که می‌خواهیم متغیری که فضای زیادی در حافظه اشغال می‌کند (برای مثال یک نوع داده‌ای تعریف شده توسط کاربر) را به تابعی ارسال کنیم و نمی‌خواهیم تابع از آن متغیر کپی بگیرد بلکه می‌خواهیم تنها آدرس مکان حافظه را به تابع ارسال شود. به این فرایند فراخوانی تابع فراخوانی با ارجاع می‌گوییم. مزیت استفاده از متغیر مرجع در اینجا نسبت به اشاره‌گر این است که عملیات توسط آن ساده‌تر انجام می‌شود.

```
*zptr = *xptr + *yptr;
zref = xref + yref;
```

- در زبان جاوا همه اشیای کلاس‌ها متغیر مرجع هستند.

```
A a1 = new A();
A a2 = a1;
a1.x = 1; // a2.x = 1
```

نوع‌های اشاره‌گر و مرجع

- یکی از روش‌های پیاده سازی اشاره‌گرها، روش قفل و کلید [1] نامیده می‌شود.
- در این روش، وقتی اشاره‌گر به یک مکان در حافظه هیپ اشاره می‌کند، اشاره‌گر علاوه بر آدرس حافظه یک کلید را ذخیره می‌کند که یک عدد صحیح است. در مکان حافظه‌ای که آن اشاره‌گر به آن اشاره می‌کند نیز همان مقدار ذخیره می‌شود که به آن قفل گفته می‌شود. در هنگام دسترسی یک اشاره‌گر به مکان حافظه، اگر قفل و کلید همخوانی داشته باشند دسترسی مجاز است. هنگامی که فضایی در حافظه آزاد می‌شود، مقدار قفل آن فضا تغییر می‌کند، بنابراین همهٔ اشاره‌گرهایی که به آن مکان اشاره می‌کنند هنگام دسترسی با پیام خطا مواجه می‌شوند چرا که قفل و کلید دیگر همخوانی ندارند.
- با استفاده از این روش مشکل اشاره‌گر معلق رفع می‌شود.
- در برخی زبان‌ها مانند جاوا اجازه آزادسازی حافظه به برنامه نویس داده نمی‌شود و بنابراین مشکل اشاره‌گر معلق وجود نخواهد داشت.

[1] locks and keys approach

نوع‌های اشاره‌گر و مرجع

- در زبان‌هایی مانند جاوا که جمع‌آوری فضاهای آزاد شده یا بازیافت حافظه (زباله‌روبی) [1] به طور خودکار انجام می‌شود، باید الگوریتم بهینه برای این کار نیز پیاده سازی شود.
- دو روش برای بازیافت حافظه وجود دارد که روش اول شمارنده ارجاع [2] و روش دوم علامت گذاری و جاروب [3] نامیده می‌شوند.
- در روش شمارندهٔ ارجاع به ازای هر سلول حافظه شمارنده‌ای در نظر گرفته می‌شود که تعداد اشاره‌گرهایی که به هر مکان از حافظه اشاره می‌کنند را می‌شمارد. هرگاه تعداد اشاره‌گرها یا متغیرهای مرجع که به یک سلول حافظه اشاره می‌کنند به صفر رسید، سلول حافظه به مجموعه سلول‌های قابل استفاده باز می‌گردد. از آنجایی که تعداد سلول‌های حافظه زیاد است تعداد شمارنده‌هایی که باید نگهداری شوند سربار زیادی به سیستم تحمیل می‌کند. در زمان اجرا نیز این شمارش سربار زمانی ایجاد می‌کند و باعث کاهش سرعت می‌شود.

[1] garbage collection
[2] reference counter
[3] mark and sweep

نوع‌های اشاره‌گر و مرجع

- در روش نشانه گذاری و جاروب، ابتدا سلول‌های حافظه تخصیص داده می‌شوند تا هنگامی که حافظه پر شود. در این لحظه بازیافت کنندهٔ حافظه همهٔ مرجع‌ها و اشاره‌گرها در برنامه را در حافظه دنبال می‌کند و همهٔ فضاهایی که توسط آن اشاره‌گرها استفاده می‌شوند را علامت گذاری می‌کند. سپس همهٔ سلول‌های حافظه که علامت گذاری نشده‌اند جاروب می‌شوند یا به عبارتی در مجموعهٔ سلول‌های قابل تخصیص قرار می‌گیرند.

بررسی نوع

- بررسی نوع [1] فعالیتی است که به موجب آن اطمینان حاصل می‌شود که همهٔ عملگرها با عملوندهای آنها همخوانی دارند. در اینجا توابع را نیز عملگر در نظر میگیریم و ورودی و خروجی توابع را نیز عملوند برای توابع.
- یک نوع سازگار [2] نوعی است که برای یک عملگر تحت قوانین حاکم بر آن زبان معتبر باشد. برای مثال وقتی یک عدد صحیح و اعشاری در زبان جاوا جمع می‌شوند، عدد صحیح به اعشاری تبدیل می‌شود بنابراین این دو نوع عملوند تحت قوانین جاوا با عملگر سازگارند.

[1] type checking
[2] compatible

بررسی نوع

- خطای نوع [1] هنگامی رخ می‌دهد که عملگر با عملوندهایش سازگاری نداشته باشد.
- اگر انقیاد نوع ایستا باشد آنگاه بررسی نوع می‌تواند در زمان کامپایل انجام شود اما اگر انقیاد نوع پویا باشد بررسی نوع در زمان اجرا خواهد بود که بررسی نوع پویا [2] نامیده می‌شود
- بررسی نوع در زبان پایتون پویاست که باعث می‌شود خطاهای نوع قبل از اجرا مشخص نشوند اما از طرفی در این زبان انعطاف پذیری برنامه افزایش یافته است.
- یک زبان برنامه نویسی در دسته زبان‌های نوع دهی قوی [3] است اگر همهٔ خطاهای نوع قبل از اجرا تشخیص داده شوند.

[1] type error
[2] dynamic type checking
[3] strongly typed

راست

- راست [1] یک زبان برنامه نویسی است که پارادایم‌های برنامه نویسی رویه‌ای، تابعی و همروند را پشتیبانی می‌کند.
- بیشترین تمرکز زبان برنامه نویسی راست بر روی راندمان [2] ، قابلیت اطمینان [3] و همروندی [4] است.

[1] rust
[2] efficiency
[3] reliability
[4] concurrency

راست

- زبان راست یک زبان کامپایل شونده است و کامپایلر آن به نحوی طراحی شده است که برنامه‌های آن از راندمان بالایی برخوردارند. علاوه بر این کامپایلر اطمینان حاصل می‌کند که برنامهٔ نوشته شده در زمان اجرا دسترسی غیر مجاز به حافظه ندارد. در زبان راست، بازیافت کنندهٔ حافظه [1] وجود ندارد اما ساختارهایی وجود دارد که به کامپایلر کمک می‌کند بتواند در زمان کامپایل از نشتی‌های احتمالی حافظه مطلع شده و پیام خطا صادر کند. بنابراین برنامه‌های نوشته شده در زبان راست از قابلیت اطمینان بالایی برخوردارند. همچنین ساختارهایی برای برنامه نویسی همروند به این زبان افزوده شده است.
- در دسامبر ۲۰۲۲، زبان راست به عنوان اولین زبان جدید در کنار زبان‌های قدیمی سی و اسمبلی در توسعهٔ هستهٔ لینوکس مورد استفاده قرار گرفت.

[1] garbage collector

راست : نمادها و متغیرها

- نماد [1] نامی است که یک مقدار را نشان می‌دهد. نمادها در زبان راست غیر قابل تغییر [2] هستند.
- یک نماد با کلمهٔ `let` تعریف می‌شود. اگر مقدار یک نماد را تغییر دهیم، کامپایلر پیام خطا صادر می‌کند.

```
۱ fn main() {
۲     let x = 5 ;
۳     println! ("The value  of x is : {x}") ;
۴     x = 6 ; // error
۵ }
```

[1] symbol
[2] immutable

راست : نمادها و متغیرها

- با استفاده از کلمه mut مخفف کلمه قابل تغییر [1] می‌توان یک نماد قابل تغییر تعریف کرد. نماد قابل تغییر در واقع یک متغیر است.

```
let mut x = 5 ;
x = 6
```

- با استفاده از کلمه const می‌توان یک ثابت تغییر کرد. مقدار یک ثابت قابل تغییر است و تفاوت آن با نماد این است که نمی‌توان آن را مجدداً تعریف کرد.

```
const PI = 3.141592 ;
PI = 3.1415 // error
const PI : f16 = 3.1415 // error
let pi = 3.141592
let pi = 3.1415 // OK
```

[1] mutable

راست : نمادها و متغیرها

- یک نماد را می‌توان در یک بلوک تعریف کرد و حوزهٔ تعریف آن نماد فقط مختص بلوک مورد نظر است.

```
۱ let x = 5 ;
۲ let x = x + 1 ;
۳ {
۴     let x = x * 2 ; // x = 12
۵ }
۶ println! ("The value of x is : {x}")
۷ // x = 6
```

راست : نمادها و متغیرها

- وقتی یک نماد بازتعریف می‌شود، نوع آن می‌تواند متفاوت باشد.

```
۱ let s = "021" ;
۲ let s = s.len() ; // s = 3
۳ let mut m = "012" ;
۴ m = m.len() ; //error
```

راست : نوع‌های داده‌ای

- نوع‌های داده‌ای عددی در زبان راست می‌توانند صحیح بدون علامت و یا اعشاری باشند. عدد صحیح علامت دار با `i` ، عدد صحیح بدون علامت با `u` و عدد اعشاری با `f` نشان داده می‌شوند.
- اگر نوع یک نماد توسط برنامه نویس تعریف نشود، کامپایلر توسط اولین مقداردهی نماد، نوع آن را مشخص می‌کند.

```
۱ let x = 2.0 // f64
۲ let y : f32 = 3.0 // f32
۳ let z = 4 // i32
۴ let w : i128 = 5 // i128 (128-bit integer)
۵ let m : u64 = 6 // u64 (64-bit unsigned integer)
```

راست : نوع‌های داده‌ای

- نوع داده‌ای منطقی توسط کلمهٔ `bool` تعریف می‌شود.

```
let t =  true ;
let f : bool = false ;
```

- نوع داده‌ای کاراکتر توسط کلمهٔ `char` تعریف می‌شود.

```
let c = 'z' ;
let z : char = 'Z' ;
```

راست : نوع‌های داده‌ای

- نوع داده‌ای چندتایی مجموعه‌ای است از چند مقدار از نوع‌های دلخواه.

```
1 let  x : (i32 , f64 , char) = (500 , 604 , 'y') ;
```

راست : نوع‌های داده‌ای

- نوع داده‌ای آرایه نوعی است که مجموعه‌ای از مقادیر از یک نوع یکسان را نگهداری می‌کند.

```
1 let a = [1, 2, 3, 4, 5]
```

راست : نوع‌های داده‌ای

- آرایه را می‌توان با نوع عناصر و تعداد عناصر آن نیز تعریف کرد.

```
let a : [i32 ; 5] = [1, 2, 3, 4, 5]
let a0 = a[0] ; a0 = 1
```

- همچنین می‌توان مقادیر یک آرایه را با استفاده از یک روش میانبر تعریف کرد.

```
let a = [3 ; 5] // [3, 3, 3, 3, 3]
```

- دسترسی به عناصر یک آرایه در بیرون از محدوده منجر به خطای زمان اجرا می‌شود.

راست : توابع

- یک تابع را می‌توان با پارامترهای ورودی آن و نوع خروجی آن تعریف کرد.

```
fn plus_one (x : i32) -> i32 {
        x+1
}
```

- عبارتی که در یک بلوک بدون نقطه ویرگول یا سمی‌کالن [1] اعلام شده است، از بلوک بازگردانده می‌شود.
- برای مثال می‌توانیم بنویسیم :

```
let y = {
            let x = 3 ;
            x + 1
} ;
// y = 4
```

[1] semicolon

راست : ساختار کنترلی

- ساختار کنترلی شرطی به صورت زیر استفاده می‌شود.

```
۱ if n % 2 == 0 {
۲       // even
۳ } else {
۴       // odd
۵ }
۶ let number = if n % 2 ==0 {6} else {5} ;
```

راست : ساختار کنترلی

- از کلمهٔ loop می‌توان برای ایجاد یک حلقه تکرار بدون شرط استفاده کرد.

```
۱ loop {
۲       println! ("again!");
۳ }
```

- یک حلقه می‌تواند یک مقدار را نیز بازگرداند.

```
۱ let mut counter = 0 ;
۲ let result = loop {
۳           counter += 1;
۴           if counter == 10 {
۵                   break counter * 2 ;
۶           }
۷ };
۸ println! ("The result is {result}") ; // 20
```

راست : ساختار کنترلی

- همچنین با استفاده از ساختار while می‌توان یک حلقه ایجاد کرد.

```
let a = [10, 20, 30, 40, 50]
let mut index = 0 ;
while index < 5 {
      println! ("The value is : {}", a[index]);
      index += 1 ;
}
```

راست : ساختار کنترلی

- همچنین با استفاده از دستور for می‌توان به صورت پیمایش در یک آرایه و یا پیمایش در یک بازهٔ عددی حلقه ایجاد کرد.

```
۱ let a = [10, 20, 30, 40, 50]
۲ for element in a {
۳         println! ("The value is : {element}");
۴ }
۵ for index in (0 .. 4) {
۶       println! ("The value is : {}", a[index]);
۷ }
```

راست : مالکیت

- مالکیت [1] یکی از مفاهیم اصلی و ویژهٔ زبان راست است که پیامدهای مهمی در استفاده از آن دارد. مفهوم مالکیت به زبان راست کمک می‌کند ایمنی استفاده از حافظه [2] را تضمین کند، بدون اینکه نیازی به بازیافت کنندهٔ حافظه [3] داشته باشد.

[1] ownership
[2] memory safety guarantee
[3] garbage collector

راست : مالکیت

- مالکیت در زبان راست به مجموعهٔ قوانینی گفته می‌شود که تعیین می‌کنند مدیریت حافظه چگونه انجام می‌شود.
- برخی از زبان‌ها مانند جاوا یک مکانیزم بازیافت حافظه دارند که به طور منظم بررسی می‌کند به کدام قسمت‌های حافظه دیگری نیازی نیست و آن مکان‌های حافظه بی استفاده را برای استفاده مجدد آزادسازی و بازیافت می‌کند. در برخی از زبان‌های دیگر مانند سی++ مدیریت حافظه باید توسط برنامه نویس به طور صریح انجام شود. مشکل زبان‌های دسته اول سرعت اجرای پایین آنهاست و مشکل زبان‌های دسته دوم پایین بودن قابلیت اطمینان برنامه‌های آنهاست چرا که برنامه نویس ممکن است به درستی حافظه را تخصیص و آزادسازی نکند.
- زبان راست راه‌حل سومی را پیشنهاد می‌کند. حافظه توسط سیستمی به نام سیستم مالکیت مدیریت می‌شود. این سیستم مالکیت قوانینی را تعریف می‌کند که در زمان کامپایل قابل بررسی هستند، پس کامپایلر می‌تواند اطمینان حاصل کند که این قوانین به درستی اعمال شده‌اند و در زمان اجرا به حافظه دسترسی ایمن وجود دارد. اگر یکی از قوانین مالکیت نقض شده باشد، برنامه راست کامپایل نمی‌شود. این مکانیزم سرعت اجرای برنامه را کاهش نمی‌دهد.

راست : مالکیت

- مفهوم مالکیت، یک مفهوم جدید در زبان‌های برنامه نویسی است که توسط زبان راست ابداع شده است.
- در بسیاری از زبان‌های برنامه نویسی نیازی به فکر کردن به حافظه پشته و هیپ نیست، زیرا حافظه توسط زبان برنامه نویسی مدیریت می‌شود. در قوانین مالکیت زبان راست، برنامه نویس آگاهانه از پشته و هیپ استفاده می‌کند.

راست : مالکیت

- حافظهٔ پشته مناسب برای فراخوانی توابع است، زیرا وقتی یک تابع فراخوانی می‌شود، فقط به متغیرهای آن تابع می‌توان دسترسی پیدا کرد و به محض اتمام اجرای تابع، متغیرهای آن از روی برداشته می‌شوند و دیگر قابل دسترسی نخواهند بود.
- حافظه هیپ نظم خاصی ندارد. وقتی برنامه به حافظه پویا نیاز دارد، مقدار حافظه مورد نیاز را درخواست می‌کند و تخصیص دهندهٔ حافظه [1] فضایی را در حافظه پیدا کرده و به اشاره‌گری به فضای تخصیص داده شده به حافظه می‌دهد.
- دسترسی به حافظه پشته سریع‌تر از دسترسی به هیپ است زیرا نیاز به جستجوی فضای خالی وجود ندارد.

[1] memory allocator

راست : مالکیت

- قوانین مالکیت در راست به صورت زیر هستند :
۱. هر مقداری در زبان راست یک مالک [1] دارد.
۲. هر مقداری در هر لحظه فقط یک مالک دارد.
۳. وقتی از حوزهٔ تعریف مالک خارج می‌شویم، مقدار آن از بین می‌رود.

[1] owner

راست : مالکیت

- برای توضیح مفهوم مالکیت از نوع داده‌ای رشته استفاده می‌کنیم.
- یک رشته می‌تواند به صورت زیر ساخته و مورد استفاده قرار بگیرد.

```
۱ let mut s = String :: from ("hello") ;
۲ s.push_str (", world!"); //append
۳ println! ("{}", s);
```

راست : مالکیت

- اگر یک رشته را بدون استفاده از نوع `String` تعریف کنیم، در واقع رشته مورد نظر بر روی حافظه قرار نمی‌گیرد و درون فایل اجرایی (قسمت کد) قرار می‌گیرد، زیرا چنین رشته‌هایی یک بار مصرف هستند. برای مثال در کد زیر رشته در قسمت کد برنامه قرار می‌گیرد.

```
1 let s = "hello"
```

راست : مالکیت

- با استفاده از تابع `from` از نوع `String` رشته مورد نظر بر روی هیپ ساخته می‌شود. از آنجایی که رشته بر روی هیپ قرار می‌گیرد، اندازهٔ آن به مقدار دلخواه می‌تواند افزایش پیدا کند.
- از آنجایی که حافظه بر روی هیپ قرار دارد بنابراین باید در هنگام نیاز تخصیص داده شود و در هنگام عدم نیاز حافظهٔ آن به فضاهای آزاد هیپ بازگردانده شود.
- تخصیص حافظه در تابع `from` برای نوع `String` انجام می‌شود، اما چگونه حافظه آزادسازی می‌شود؟
- در زبان‌هایی که از سازوکار بازیافت حافظه استفاده می‌کنند، بازیافت کننده به طور خودکار وقتی به حافظه نیاز نیست آن را آزاد می‌کند، اما در زبان‌هایی که بازیافت کنندهٔ حافظه وجود ندارد، این کار باید توسط برنامه نویس انجام شود. آزادسازی حافظه به طور دستی معمولا کار سختی است. اگر آزادسازی حافظه فراموش شود با نشت حافظه مواجه می‌شویم که در بلند مدت منجر به پر شدن حافظه می‌شود. اگر آزادسازی حافظه زود انجام شود دسترسی‌های بعدی آن با خط مواجه می‌شوند. اگر آزادسازی حافظه دوبار انجام شود، با خطای دسترسی مواجه می‌شویم.

راست : مالکیت

- در زبان راست هرگاه از حوزهٔ تعریف متغیری خارج شویم فضای حافظهٔ آن آزاد می‌شود.
- بنابراین در خارج از آکولاد درکد زیر s غیر قابل دسترس است و فضای حافظه آن آزاد می‌شود.

```
{
    let s = String :: from ("hello") ;
    // do stuff with s
} // the scope is over, and s and
// its memory are no longer valid
```

- در پایان حوزهٔ تعریف، راست به طور خودکار تابع drop را فراخوانی و حافظه را به تخصیص دهندهٔ حافظه پس می‌دهد.

راست : مالکیت

- کد زیر را در نظر بگیرید :

```
let x = 5 ;
let y = x ;
```

- نمادها و متغیرهای نوع صحیح بر روی حافظه پشته ساخته می‌شود، بنابراین در واقع مقدار ۵ در متغیر x کپی و سپس مقدار x که ۵ است در y کپی می‌شود.

راست : مالکیت

- حال کد زیر را در نظر بگیرید :

```
let s1 = String :: from ("hello");
let s2 = s1;
```

- ممکن است انتظار داشته باشیم که به طور مشابه به مقدار s1 در s2 کپی شود، اما چون s1 به یک فضا در هیپ اشاره می‌کند این اتفاق نمی‌افتد.
- یک رشته از نوع String از سه بخش تشکیل شده است. اشاره‌گری به مکان حافظه در هیپ، اندازه رشته و ظرفیت رشته که اعداد صحیح هستند. این ساختار که شامل یک اشاره‌گر و دو مقدار است بر روی پشته قرار می‌گیرد، اما محتوای رشته بر روی هیپ قرار می‌گیرد.
- وقتی رشته s2 برابر با رشته s1 قرار می‌گیرد، در واقع آدرس اشاره‌گر و اندازه رشته متعلق به s1 در s2 کپی می‌شود. پس s2 به همان فضای هیپ اشاره می‌کند که s1 اشاره می‌کند.

راست : مالکیت

- گفتیم وقتی از حوزهٔ تعریف یک متغیر خارج می‌شویم آزادسازی حافظه انجام می‌شود، اما باید توجه داشت که اشاره‌گر متعلق به کدام متغیر است و آزادسازی حافظه برای کدام متغیر انجام می‌شود.
- وقتی می‌نویسیم `let s2 = s1` در واقع `s1` بی اعتبار می‌شود. بنابراین در کد زیر با پیام خطای کامپایلر مواجه می‌شویم :

```
۱ let s1 = String :: from ("hello");
۲ let s2 = s1;
۳ println! ("{} , world!", s1); // error
```

راست : مالکیت

- در زبان‌های برنامه نویسی دیگر، وقتی مقدار یک اشاره‌گر کپی می‌شود می‌گوییم کپی سطحی [1] انجام شده است و هنگامی که مکان حافظه در هیپ برای اشاره گری کپی می‌شود می‌گوییم کپی عمیق [2] انجام شده است از آنجایی که در کپی سطحی در راست مقدار اول بی اعتبار می‌شود به آن عمل جابجایی [3] گفته می‌شود.
- پس در کد قبلی تنها s2 معتبر است و هرگاه از حوزه تعریف s2 خارج شویم، حافظه آزاد می‌شود.

[1] shallow copy
[2] deep copy
[3] move

راست : مالکیت

- اما اگر بخواهیم مقدار یک رشته را کپی عمیق کنیم بعنی فضای جدیدی در حافظه هیپ تخصیص دهیم از تابع clone استفاده می‌کنیم.

```
۱ let s1 = String :: from ("hello");
۲ let s2 = s1 . clone () ;
۳ println! ("s1 = {} , s2 = {}", s1, s2);
```

راست : مالکیت

- در فراخوانی تابع نیز وقتی یک اشاره‌گر به عنوان آرگومان به تابعی ارسال شود، تابع مالکیت متغیر را به دست می‌آورد، بنابراین با خارج شدن از تابع باید فضای آن آزاد شود.

```
fn main() {
    let s = String::from("hello");
    takes_ownership(s);  // s's value moves into
                         // the function. and so is
                        // no longer valid.
}
fn takes_ownership(st: String) {
     println!("{}", st);
} // st goes out of scope and drop is called.
```

- اگر بخواهیم به متغیر s بعد از فراخوانی تابع دسترسی پیدا کنیم، با خطای کامپایل مواجه می‌شویم. چنین پیام‌های خطا قابلیت اطمینان برنامه‌های راست را افزایش می‌دهد.

راست : مالکیت

- همچنین بازگرداندن یک مقدار از یک تابع مالکیت را منتقل می‌کند.

```
۱ fn main() {
۲     let s1 = String::from("hello");
۳     let s2 = takes_given_ownership(s1);
۴ } // frop is called for s2
۵ fn takes_given_ownership(s: String) -> String {
۶      s
۷ } // s moves out of function
```

راست : مالکیت

- حال ممکن است بخواهیم یک تابع را فراخوانی کنیم اما نخواهیم مالکیت را متغیرهایی که به عنوان آرگومان ارسال می‌شوند را از دست بدهیم. راه حل اول این است که مالکیت را بعد از دست دادن، پس بگیریم :

```
fn main() {
    let s1 = String::from("hello");
    let (s2, len) = calculate_length(s1);
    println!("The length of '{}' is {}.", s2, len);
}
fn calculate_length(s: String) -> (String, usize) {
    let length = s.len(); // len() returns the length of a String
    (s, length)
}
```

- اما اگر بخواهیم همیشه این کار را انجام دهیم برنامه نویسی بسیار سخت می‌شود.

- خوشبختانه در راست مفهوم دیگری به نام مرجع وجود دارد که توسط آن می‌توان از انتقال مالکیت جلوگیری کرد.

راست : مرجع‌ها

- یک متغیر مرجع در زبان راست، همانند متغیر مرجع در سی++ یک نام مستعار برای یک مکان حافظه است.
- وقتی یک متغیر با استفاده از یک متغیر مرجع به عنوان پارامتر به یک تابع ارسال شود، مالکیت منتقل نمی‌شود.
- در برنامه زیر طول یک رشته محاسبه می‌شود بدون اینکه مالکیت متغیر s1 منتقل شود.

```
۱ fn main() {
۲     let s1 = String::from("hello");
۳     let len = calculate_length(&s1);
۴     println!("The length of '{}' is {}.", s1, len);
۵ }
۶ fn calculate_length(s: &String) -> usize {
۷     s.len()
۸ }
```

راست : مرجع‌ها

- یک متغیر مرجع همانند سی++ با علامت امپرسند (&) تعریف می‌شود و همچنین آدرس یک متغیر با عملگر (&) به دست می‌آید.
- بنابراین `&s1` به مکان حافظه `s1` اشاره می‌کند و `s` در پارامتر تابع یک متغیر از نوع مرجع است. از آنجایی که یک متغیر مرجع هیچ مالکیتی بر داده‌ای که به آن اشاره می‌کند ندارد بنابراین وقتی از تابع خارج شویم، فضای حافظه‌ای که `s` به آن اشاره می‌کند آزاد نمی‌شود.
- به این عملیات قرض گرفتن [1] گفته می‌شود، چرا که یک متغیر مرجع مالکیت حافظه را به دست نمی‌آورد، اما آن را برای استفاده برای مدت زمانی قرض می‌گیرد. در واقع در عملیات قرض گرفتن متغیر مرجع مالکیت موقت به دست می‌آورد و پس از اتمام کار خود مالکیت را پس می‌دهد.
- همانطور که مقدار یک متغیر غیر قابل تغییر را نمی‌توان تغییر داد، متغیر مرجع نیز که به یک نماد غیر قابل تغییر اشاره می‌کند، قابل تغییر نیست.

[1] borrowing

راست : مرجع‌ها

- یک متغیر مرجع را می‌توان قابل تغییر [1] تعریف کرد. یک مرجع قابل تغییر به یک متغیر قابل تغییر اشاره می‌کند و مقدار را می‌توان تغییر داد.
- برای مثال :

```
۱ fn main() {
۲     let mut s = String::from("hello");
۳     change(&mut s);
۴ }
۵ fn change(some_string: &mut String) {
۶     some_string.push_str(", world");
۷ }
```

[1] mutable reference

راست : مرجع‌ها

- یک متغیر قابل تغییر را در یک زمان فقط به یک مرجع قابل تغییر می توان قرض داد. اگر یک متغیر به بیش از دو مرجع قابل تغییر قرض داده شود، کامپایلر پیام خطا صادر می‌کند.
- بنابراین کامپایل برنامه زیر پیام خطا صادر می‌کند.

```
۱ let mut s = String::from("hello");
۲ let r1 = &mut s;
۳ let r2 = &mut s; // error
۴ println!("{}, {}", r1, r2);
```

راست : مرجع‌ها

- در واقع در این کد خواسته‌ایم یک متغیر قابل تغییر را در یک زمان به دو مرجع قرض دهیم، که این عملیات در زبان راست ممنوع است.

- این محدودیت به کامپایلر کمک می‌کند که از وضعیت رقابت داده [1] در زمان کامپایل جلوگیری کند. اگر دو یا چند مرجع بتوانند به طور همزمان به یک داده دسترسی داشته باشند ممکن است هر دو به طور همزمان داده را تغییر دهند و رفتار سیستم غیر قابل پیش بینی می‌شود و پیدا کردن خطای خروجی بسیار مشکل می‌شود. راست از این خطاهای احتمالی در زمان کامپایل جلوگیری می‌کند.

- البته اجازه داریم یک مرجع قابل تغییر را در یک بلوک جداگانه تعریف کنیم

```
let mut s = String::from("hello");
    {
        let r1 = &mut s;
    } // r1 goes out of scope here, so we can make a new reference
      //with no problems.
let r2 = &mut s;
```

[1] data race

راست : مرجع‌ها

- چند مرجع غیر قابل تغییر به طور همزمان می‌توانند برای اشاره به یک داده تعریف شوند، ولی مرجع قابل تغییر حتی با مرجع غیر قابل تغییر هم نمی‌تواند به طور همزمان تعریف شود.

```
۱ let mut s = String::from("hello");
۲ let r1 = &s; // no problem
۳ let r2 = &s; // no problem
۴ let r3 = &mut s; // error
۵ println!("{}, {}, and {}", r1, r2, r3);
```

راست : مرجع‌ها

- دلیل این امر این است که مرجع‌های غیر قابل تغییر انتظار ندارند مقداری که به آن اشاره می‌کنند ناگهان تغییر کندو اما چند مرجع غیر قابل تغییر به طور همزمان می‌توانند تعریف شوند چون هیچ‌کدام مقدار داده را تغییر نمی‌دهند.

راست : مرجع‌ها

- دقت کنید که حوزه تعریف یک متغیر مرجع از زمانی است که تعریف می‌شود تا زمانی که استفاده می‌شود.
  بنابراین اگر یک متغیر مرجع تعریف و سپس استفاده شود، یک متغیر مرجع قابل تغییر بعد از آن می‌تواند تعریف شود و به همان داده متغیر قبلی اشاره کند.
- بنابراین کد زیر بدون خطا کامپایل می‌شود.

```
۱ let mut s = String::from("hello");
۲ let r1 = &s; // no problem
۳ let r2 = &s; // no problem
۴ println!("{} and {}", r1, r2);
۵ // variables r1 and r2 will not be used after this point
۶ let r3 = &mut s; // no problem
۷ println!("{}", r3);
```

راست : مرجع‌ها

- قوانین مالکیت و قرض گرفتن ممکن است کمی پیچیده به نظر برسند، اما این قوانین کمک می‌کنند که کامپایلر خطاهای احتمالی را در زمان کامپایل پیدا کرده و از بروز آنها جلوگیری کند. بدون این قوانین ممکن است برنامه به راحتی کامپایل شود ولی پیدا کردن خطا به طور دستی توسط برنامه نویس می‌تواند بسیار پیچیده و دشوار شود.

راست : مرجع‌ها

- در زبان‌هایی که اشاره‌گر در آنها وجود دارد، ممکن است به راحتی خطای اشاره‌گر معلق [1] به وجود بیاید، بدین معنی که یک اشاره‌گر به مکانی در حافظه اشاره کند که توسط یک متغیر دیگر آزاد شده باشد.
- اما در زبان راست کامپایلر اطمینان حاصل می‌کند که هیچ‌گاه مرجع معلق به وجود نمی‌آید. اگر مرجعی به یک متغیر وجود داشته باشد، کامپایلر اطمینان حاصل می‌کند که متغیر مربوطه قبل از مرجع از حوزه تعریف خارج نمی‌شود و فضای حافظه آن آزاد نمی‌شود.

[1] dangling pointer

راست : مرجع‌ها

- کد زیر را در نظر بگیرید. وقتی از تابع خارج می‌شویم متغیر s از بین می‌رود، اما برنامه نویس مرجعی از آن بازگردانده است که کامپایلر پیام خطا صادر می‌کند.

```
۱ fn main() {
۲     let reference_to_nothing = dangle();
۳ }
۴ fn dangle() -> &String { // dangle returns a reference to a String
۵     let s = String::from("hello"); // s is a new String
۶     &s // error: we return a reference to the String, s
۷ } // Here, s goes out of scope, and is dropped. Its memory goes away.
۸   // Danger!
```

راست : مرجع‌ها

- برنامه را می‌توانیم به صورت زیر صحیح کنیم. مالکیت s از درون تابع به بیرون انتقال پیدا می‌کند.

```
fn no_dangle() -> String {
    let s = String::from("hello");
    s
}
```

- بنابراین در هر زمان، یا فقط یک مرجع قابل تغییر می‌تواند به یک داده اشاره کند و یا تعدادی مرجع غیر قابل تغییر.

راست : نوع برش

- برش [1] کمک می‌کند مرجعی به دنباله‌ای از عناصر در یک مجموعه بسازیم. از آنجایی که برش یک اشاره‌گر است، مالکیت داده ندارد.
- فرض کنید می‌خواهیم یک برش یا یک قسمت از یک رشته را توسط یک تابع بازگردانیم.
- برای مثال اگر بخواهیم اولین کلمه از یک رشته را که با خط فاصله از کلمهٔ دوم جدا شده است به دست آوریم، می‌توانیم تابع زیر را بنویسیم :

```
۱ fn first_word(s: &String) -> usize {
۲     let bytes = s.as_bytes();
۳     for (i, &item) in bytes.iter().enumerate() {
۴         if item == b' ' {
۵             return i;
۶         }
۷     }
۸     s.len()
۹ }
```

[1] ..

راست : نوع برش

- خروجی تابع، اندیس اولین خط فاصله در رشته ورودی است. حال فرض کنید پس از یافتن اولین خط فاصله، رشته را به یک رشتهٔ تهی تبدیل کنیم. متغیری که به اندیس اولین خط فاصله اشاره می‌کند، اکنون غیر معتبر است.

```
fn main() {
    let mut s = String::from("hello world");
    let word = first_word(&s); // word will get the value 5
    s.clear(); // this empties the String, making it equal to ""
    // word still has the value 5 here, but there's no more string that
    // we could meaningfully use the value 5 with. word is now totally
}
```

راست : نوع برش

- می‌خواهیم برنامه را به گونه‌ای بنویسیم که در زمان کامپایل این خطا تشخیص داده شود.
- برای این کار از نوع برش استفاده می‌کنیم.

راست : نوع برش

- یک برش از یک آرایه یا رشته به صورت زیر تعریف می‌شود.

```
۱  let s = String::from("hello world");
۲  let hello = &s[0..5];
۳  let world = &s[6..11];
۴  let slice = &s[0..2];
۵  let slice = &s[..2];
۶  let len = s.len();
۷  let slice = &s[3..len];
۸  let slice = &s[3..];
۹  let slice = &s[0..len];
۱۰ let slice = &s[..];
```

راست : نوع برش

- حال تابع `first_world` را با استفاده از برش به صورت زیر تعریف می‌کنیم.

```
۱ fn first_word(s: &String) -> &str {
۲     let bytes = s.as_bytes();
۳     for (i, &item) in bytes.iter().enumerate() {
۴         if item == b' ' {
۵             return &s[0..i];
۶         }
۷     }
۸     &s[..]
۹ }
```

راست : نوع برش

- کامپایلر اطمینان حاصل می‌کند که همه مرجع‌ها به یک متغیر معتبر می‌مانند. پس اگر سعی کنیم این بار رشته را به یک رشتهٔ تهی تبدیل کنیم، از آنجایی که متغیر مرجعی داریم که به یک برش از رشته اشاره می‌کند، کامپایل پیام خطا صادر می‌کند.

```
۱ fn main() {
۲     let mut s = String::from("hello world");
۳     let word = first_word(&s);
۴     s.clear(); // error!
۵     println!("the first word is: {}", word);
۶ }
```

راست : نوع برش

- در واقع در اینجا یکی از قوانین قرض گرفتن به کمک ما می‌آید. به یاد داریم وقتی که یک مرجع به یک متغیر داریم نمی‌توانیم یک مرجع قابل تغییر تعریف کنیم. از آنجایی که تابع `clear()` می‌خواهد رشته را تهی کند باید یک مرجع قابل تغییر از آن بگیرد و این کار امکان پذیر نیست چرا که متغیر `word` به یک مرجع به متغیر `s` است.
- بنابراین کامپایل راست به ما کمک کرد که از یک خطای احتمالی در زمان اجرا جلوگیری کنیم.

راست : نوع برش

- وقتی یک رشته را به صورت خام تعریف می‌کنیم، رشته در داخل کد قرار می‌گیرد و بنابراین نوع آن از نوع برش است، زیرا رشته تعریف شده یک برش از کد است.

```
let s = "Hello, world!";
```

- در اینجا متغیر s از نوع &str است که یک برش از یک رشته است. در واقع &str یک مرجع غیر قابل تغییر است.

راست : نوع برش

- می‌توانستیم ورودی تابع `first_word` را از نوع `&str` در نظر بگیریم تا بتوانیم از برش‌ها نیز برش به دست آوریم.

```
1 fn first_word(s:&str) -> &str {
```

راست : نوع برش

- از آرایه‌ها نیز می‌توانیم به صورت زیر برش تهیه کنیم.

```
let a = [1,2,3,4,5];
let slice = &a[1..3];
```

راست : نوع برش

- به طور خلاصه، با استفاده از مفاهیم مالکیت، قرض دادن و برش می‌توان در زمان کامپایل اطمینان حاصل کرد که برنامه‌های راست به طور امن از حافظه استفاده می‌کنند و برنامه در زمان اجرا با دسترسی غیر مجاز مواجه نمی‌شود.

راست : ساختمان

- یک ساختمان را در زبان راست می‌توان به صورت زیر تعریف کرد.

```
struct User {
    active: bool,
    username: String,
    email: String,
    sign_in_count: u64,
}
```

- تفاوت نوع چندتایی و نوع ساختمان در این است که به اعضای ساختمان می‌توان با نام دسترسی پیدا کرد.

راست : ساختمان

- همچنین برای تعریف یک ساختمان می‌توان با استفاده از نام اعضای ساختمان آن را مقداردهی اولیه کرد.

```
fn main() {
    let mut user1 = User {
        active: true,
        username: String::from("someusername123"),
        email: String::from("someone@example.com"),
        sign_in_count: 1,
    };
    user1.email = String::from("anotheremail@example.com");
}
```

- یک نمونه از یک ساختمان می‌تواند قابل تغییر یا غیر قابل تغییر باشد.

راست : ساختمان

- یک نمونه از یک ساختمان در یک تابع توسط پارامترهای تابع می‌تواند به صورت زیر ساخته شود.

```
fn build_user(email: String, username: String) -> User {
    User {
        active: true,
        username: username,
        email: email,
        sign_in_count: 1,
    }
}
```

راست : ساختمان

- یک میانبر نیز برنامه مقداردهی اولیه نمونه ساختمان به صورت زیر وجود دارد.

```
fn build_user(email: String, username: String) -> User {
    User {
        active: true,
        username,
        email,
        sign_in_count: 1,
    }
}
```

راست : ساختمان

- یک نمونه از ساختمان را می‌توان با استفاده از مقادیر یک نمونه دیگر از ساختمان به صورت زیر ساخت :

```
let user2 = User {
    active: user1.active,
    username: user1.username,
    email: String::from("another@example.com"),
    sign_in_count: user1.sign_in_count,
};
```

راست : ساختمان

- یک میانبر نیز برای ساختن یک نمونه با استفاده از مقادیر یک نمونهٔ دیگر به صورت زیر وجود دارد :

```
let user2 = User {
    email: String::from("another@example.com"),
    ..user1
};
```

راست : ساختمان

- در راست می‌توان ساختمان‌ها را بدون ذکر نام عناصر نیز ایجاد کرد. این نوع ساختمان‌ها وقتی استفاده می‌شوند که می‌خواهیم یک چندتایی تعریف کنیم که دارای یک نام معین باشد.

```
۱ struct Color(i32, i32, i32);
۲ struct Point(i32, i32, i32);
۳ fn main() {
۴     let black = Color(0, 0, 0);
۵     let origin = Point(0, 0, 0);
۶ }
```

راست : ساختمان

- یک ساختمان را می‌توان بدون عضو نیز تعریف کرد. در آینده خواهیم دید چگونه برای یک ساختمان رفتار تعریف می‌کنیم.

```
۱ struct AlwaysEqual;
۲ fn main() {
۳     let subject = AlwaysEqual;
۴ }
```

راست : ساختمان

- یک ساختمان می‌تواند علاوه بر اعضای داده‌ای تعدادی تابع عضو نیز داشته باشد که به آنها متود [1] گفته می‌شود. متودهای یک ساختمان به صورت زیر تعریف می‌شوند.

```
۱  #[derive(Debug)]
۲  struct Rectangle {
۳      width: u32,
۴      height: u32,
۵  }
۶  impl Rectangle {
۷      fn area(&self) -> u32 {
۸          self.width * self.height
۹      }
۱۰ }
```

[1] method

راست : ساختمان

```
fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };
    println!(
        "The area of the rectangle is {} square pixels.",
        rect1.area()
    );
}
```

راست : ساختمان

- یک متود می‌تواند با یک فیلد ساختمان همنام باشد.

```
۱  impl Rectangle {
۲      fn width(&self) -> bool {
۳          self.width > 0
۴      }
۵  }
۶  fn main() {
۷      let rect1 = Rectangle {
۸          width: 30,
۹          height: 50,
۱۰     };
۱۱     if rect1.width() {
۱۲         println!("The rectangle has a nonzero width; it is {}", rect1.w
۱۳     }
۱۴ }
```

راست : ساختمان

- یک متود می‌تواند پارامتر نیز داشته باشد.

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
    fn can_hold(&self, other: &Rectangle) -> bool {
        self.width > other.width && self.height > other.height
    }
}
fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };
```

راست : ساختمان

```
    let rect2 = Rectangle {
        width: 10,
        height: 40,
    };
    let rect3 = Rectangle {
        width: 60,
        height: 45,
    };
    println!("Can rect1 hold rect2? {}", rect1.can_hold(&rect2));
    println!("Can rect1 hold rect3? {}", rect1.can_hold(&rect3));
}
```

راست : ساختمان

- توابع یک ساختمان می‌توانند به عنوان ورودی پارامتر self نداشته باشند. این توابع را توابع مرتبط [1] با ساختمان می‌نامیم. این توابع متعلق به یک ساختمان هستند، برخلاف متودها که متعلق به نمونه‌های ساختمان هستند. اگر بخواهیم در یک تابع مرتبط با یک ساختمان، نوع ساختمان را بازگردانیم از کلمهٔ Self استفاده می‌کنیم.

```
impl Rectangle {
    fn square(size: u32) -> Self {
        Self {
            width: size,
            height: size,
        }
    }
}
```

[1] associated function

راست : ساختمان

- به توابع مرتبط می‌توان توسط عملگر (::) دسترسی پیدا کرد، به طور مثال
`Regtangle :: square(3);` یک نمونه از ساختمان مربع به طول ضلع ۳ باز می‌گرداند.

راست : ساختمان

- متودها و توابع مرتبط با یک ساختمان را می‌توان در چند قطعه جدا نیز تعریف کرد.

```
۱  impl Rectangle {
۲      fn area(&self) -> u32 {
۳          self.width * self.height
۴      }
۵  }
۶  impl Rectangle {
۷      fn can_hold(&self, other: &Rectangle) -> bool {
۸          self.width > other.width && self.height > other.height
۹      }
۱۰ }
```

راست : نوع شمارشی

- نوع داده شمارشی در راست با کلمه enum تعریف می‌شود. از نوع دادهٔ شمارشی برای نام‌گذاری مجموعه‌ای از مقادیر استفاده می‌شود.
- برای مثال :

```
enum IpAddrKind {
    V4,
    V6,
}
struct IpAddr {
    kind: IpAddrKind,
    address: String,
}
let home = IpAddr {
    kind: IpAddrKind::V4,
    address: String::from("127.0.0.1"),
};
```

راست : نوع شمارشی

- اما گاهی نیاز داریم در عناصر نوع داده شمارشی، مقداری نیز قرار دهیم. بدین ترتیب نیاز نداریم هر بار در کنار نوع دادهٔ شمارشی در یک ساختمان نیز تعریف کنیم و مقادیر مورد نیاز را در ساختمان قرار دهیم.
- در راست نوع دادهٔ شمارشی می‌تواند مقداری نیز در خود ذخیره کند. برای مثال :

```
۱ enum IpAddr {
۲     V4(String),
۳     V6(String),
۴ }
۵ let home = IpAddr::V4(String::from("127.0.0.1"));
```

راست : نوع شمارشی

- نوع دادهٔ شمارشی زیر معادل تعریف چهار ساختمان متفاوت است، با این تفاوت که نوع دادهٔ شمارشی همهٔ ساختمان‌های همانند را در یک گروه قرار می‌دهد.

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
// it is equivalent to :
struct QuitMessage; // unit struct
struct MoveMessage {
    x: i32,
    y: i32,
}
struct WriteMessage(String); // tuple struct
struct ChangeColorMessage(i32, i32, i32); // tuple struct
```

راست : نوع شمارشی

- مشکل تعریف چند ساختمان در مثال قبل این است که اگر بخواهیم تابعی تعریف کنیم که با همهٔ این ساختمان‌ها رفتار مشابهی انجام دهد، امکان آن وجود ندارد و باید یک تابع به ازای هر یک ساختمان‌ها تعریف کنیم.

راست : نوع شمارشی

- برای داده‌های شمارشی همانند ساختمان‌ها می‌توانیم متود تعریف کنیم.

```
۱  enum Message {
۲      Quit,
۳      Move { x: i32, y: i32 },
۴      Write(String),
۵      ChangeColor(i32, i32, i32),
۶  }
۷  impl Message {
۸      fn call(&self) {
۹          // method body would be defined here
۱۰     }
۱۱ }
۱۲ let m = Message::Write(String::from("hello"));
۱۳ m.call();
```

راست : نوع شمارشی

- یک نوع دادهٔ شمارشی که توسط کتابخانهٔ استاندارد تعریف شده است، نوع `option` یا انتخاب است. این نوع داده یک سناریوی بسیار پر کاربرد دارد. یک متغیر در بسیاری از مواقع یا مقداری دارد که متناسب با نوع متغیر است یا هیچ مقداری ندارد.
- برای مثال می‌خواهید عنصر اول یک لیست را در یک متغیر ذخیره کنید. این متغیر صحیح یا مقداری می‌گیرد و یا اگر لیست خالی باشد هیچ مقداری نمی‌گیرد.
- در زبان‌های دیگر همهٔ این حالات باید توسط برنامه نویس بررسی شوند ولی در زبان راست کامپایلر اطمینان حاصل می‌کند که همهٔ حالات بررسی شده‌اند و در غیر اینصورت پیام خطا صادر می‌کند.

راست : نوع شمارشی

- در زبان راست همچون بسیاری زبان‌های دیگر مقدار `null` وجود ندارد، چرا که طراحی‌های قبلی وظیفه برنامه نویس بود که بررسی کند آیا مقداری `null` است یا خیر. در طراحی زبان راست از نوع داده‌ای `option` استفاده می‌شود و بدین صورت در زمان کامپایل اطمینان حاصل می‌شود که همهٔ حالات بررسی شده‌اند.
- مشکل مقدار `null` این است که اگر مقدار آن را بدون بررسی به عنوان یک مقدار غیر تهی استفاده کنیم با خطای زمان اجرا مواجه می‌شویم.

راست : نوع شمارشی

- نوع داده‌ای option به صورت زیر تعریف و به‌کار برده می‌شود.

```
enum Option<T> {
    None,
    Some(T),
}
let some_number = Some(5);
let some_char = Some('e');
let absent_number: Option<i32> = None;
```

راست : نوع شمارشی

- حال اگر سعی کنیم یک عدد صحیح را با عدد صحیح دیگری که می‌تواند تهی نیز باشد جمع کنیم با خطای کامپایل مواجه می‌شویم. در زبان‌های دیگر برنامه کامپایل می‌شود و با خطای زمان اجرا مواجه می‌شویم.
بنابراین در زبان راست برنامه نویس با مواجه شدن با خطای کامپایل مجبور می‌شود حالت‌های مختلف را در نظر بگیرید.

```
۱ let x: i8 = 5;
۲ let y: Option<i8> = Some(5);
۳ let sum = x + y; //error
```

راست : نوع شمارشی

- عبارت `match` یک ساختار کنترلی در زبان راست برای تطبیق مقدار یک نوع داده شمارشی است. با استفاده از این ساختار کنترلی می‌توانیم همهٔ حالت‌های یک نوع دادهٔ شمارشی را بررسی کنیم.

راست : ساختار کنترلی تطابق

- در زبان راست یک ساختار کنترلی به نام تطابق یا `match` وجود دارد که به برنامه نویس کمک می‌کند یک مقدار را با چند الگوی متعدد مقایسه کند و سپس بر اسا الگوی تطبیق داده شده، دستورات مناسب را اجرا کند.
- کامپایلر اطمینان حاصل می‌کند که همهٔ الگوهای ممکن بررسی شده‌اند و بنابراین اگر برنامه نویس فراموش کند تعدادی از حالات را بررسی کند، با خطای کامپایلر مواجه می‌شود.

راست : ساختار کنترلی تطابق

- برای مثال فرض کنید یک نوع دادهٔ شمارشی داریم که همهٔ سکه‌های پولی موجود را شمارش می‌کند. حال می‌خواهیم تابعی بنویسیم که ارزش یک سکه را برگرداند. نیاز داریم که این تابع همهٔ حالات را بررسی کند، پس می‌توانیم به صورت از `match` استفاده کنیم.

راست : ساختار کنترلی تطابق

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

راست : ساختار کنترلی تطابق

- ساختار کنترلی `match` شباهت زیادی با `if` دارد، با این تفاوت که در `if` یک شرط منطقی بررسی می‌شود اما در اینجا تطابق یک متغیر با مقدار بررسی می‌شود.
- در یک بلوک تطابق چند شاخه [1] وجود دارد. هر شاخه از دو بخش تشکیل شده است. بخش اول الگو و بخش دوم کد عملیاتی است و این دو بخش با علامت => از یکدیگر جدا می‌شوند.

[1] arm

راست : ساختار کنترلی تطابق

- در قسمت کد عملیاتی اگر چندین دستور وجود داشته باشند، از آکولاد استفاده می‌کنیم.

```
۱  fn value_in_cents(coin: Coin) -> u8 {
۲      match coin {
۳          Coin::Penny => {
۴              println!("Lucky penny!");
۵              1
۶          }
۷          Coin::Nickel => 5,
۸          Coin::Dime => 10,
۹          Coin::Quarter => 25,
۱۰     }
۱۱ }
```

راست : ساختار کنترلی تطابق

- همچنین چنانکه گفتیم هر یک از اعضای یک نوع داده شمارشی می توانند مقدار نیز داشته باشند، پس می‌توانیم ساختار تطابق را به صورت زیر نیز بنویسیم.

```
#[derive(Debug)] // so we can inspect the state in a minute
enum UsState {
    Alabama,
    Alaska,
    // --snip--
}
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}
```

راست : ساختار کنترلی تطابق

```
fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
            println!("State quarter from {:?}!", state);
            25
        }
    }
}
```

راست : ساختار کنترلی تطابق

- پس با استفاده از نوع دادهٔ انتخاب یا `option` و ساختار کنترلی تطابق یا `match` می‌توانیم حالات مختلف یک مقدار که می‌تواند تهی باشد را بررسی کنیم.

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        None => None,
        Some(i) => Some(i + 1),
    }
}
let five = Some(5);
let six = plus_one(five);
let none = plus_one(None);
```

راست : ساختار کنترلی تطابق

- همانطور که اشاره شد همهٔ شاخه‌های ممکن در یک الگو باید بررسی شوند، در غیر اینصورت عبارت تطابق با خطای کامپایل مواجه می‌شود.

```
۱ fn plus_one(x: Option<i32>) -> Option<i32> {
۲     match x {
۳         Some(i) => Some(i + 1),
۴     } // error
۵ }
```

راست : ساختار کنترلی تطابق

- در برخی مواقع پس از این که چند الگو را بررسی کردیم می‌خواهیم با بقیه الگوها به طور مشابه رفتار کنیم. در این مواقع از کلمهٔ `other` استفاده می‌کنیم.

```
let dice_roll = 9;
match dice_roll {
    3 => add_fancy_hat(),
    7 => remove_fancy_hat(),
    other => move_player(other),
}
fn add_fancy_hat() {}
fn remove_fancy_hat() {}
fn move_player(num_spaces: u8) {}
```

راست : ساختار کنترلی تطابق

- همچنین در برخی مواقع با بقیهٔ الگوها می‌خواهیم مشابه رفتار کنیم اما نیازی به مقدار آن الگوها نداریم. در چنین مواقعی از زیر خط (_) استفاده می‌کنیم.

```
let dice_roll = 9;
match dice_roll {
    3 => add_fancy_hat(),
    7 => remove_fancy_hat(),
    _ => reroll(),
}
fn add_fancy_hat() {}
fn remove_fancy_hat() {}
fn reroll() {}
```

راست : ساختار کنترلی تطابق

- و در نهایت گاهی بر روی مابقی الگوها نمی‌خواهیم هیچ عملیاتی انجام دهیم.

```
۱ let dice_roll = 9;
۲ match dice_roll {
۳     3 => add_fancy_hat(),
۴     7 => remove_fancy_hat(),
۵     _ => (),
۶ }
۷ fn add_fancy_hat() {}
۸ fn remove_fancy_hat() {}
```

راست : ساختار کنترلی تطابق

- در بسیاری مواقع بررسی کردن حالت‌های باقیمانده که عملیاتی نمی‌خواهیم بر روی آنها انجام دهیم، برنامه بسیار شلوغ می‌کند. یک ساختار میانبر به نام `if let` برای چنین مواقعی وجود دارد.

```
۱  let config_max = Some(3u8);
۲  match config_max {
۳      Some(max) => println!("The maximum is configured to be {}", max),
۴      _ => (),
۵  }
۶  // it is equivalent to :
۷  let config_max = Some(3u8);
۸  if let Some(max) = config_max {
۹      println!("The maximum is configured to be {}", max);
۱۰ }
```

راست : ساختار کنترلی تطابق

- از ساختار `if let` به صورت زیر می‌توان استفاده کرد.

```
let mut count = 0;
match coin {
    Coin::Quarter(state) => println!("State quarter from {:?}!", state),
    _ => count += 1,
}
// it is equivalent to :
let mut count = 0;
if let Coin::Quarter(state) = coin {
    println!("State quarter from {:?}!", state);
} else {
    count += 1;
}
```

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 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] به زبان منطق ریاضی است برای توصیف محاسبات بر اساس توابع انتزاعی.
- یک دستگاه صوری ساختاری است برای بیان اصول و استنتاج قضایا بر پایهٔ اصول با استفاده از 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 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

- 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

- 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.
- متغیر x مقید است اگر در بدنهٔ تابع $\lambda x.M$ وجود داشته باشد، در غیر اینصورت x یک متغیر آزاد است.

[1] α-equivalent
[2] scope

حساب لامبدا

- 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 $\mathtt{inc}\ 3$ را محاسبه کنیم.
- در واقع باید تابع $\mathtt{inc}$ را بر روی تابع 3 اعمال کنیم.
- ابتدا معادل عبارت های $\mathtt{inc}$ و 3 را می‌نویسیم.

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

حساب لامبدا

- حال محاسبات را به صورت زیر انجام می‌دهیم.

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

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

حساب لامبدا

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

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

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.

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

حساب لامبدا

- حال محاسبات را به صورت زیر انجام می‌دهیم.

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

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

حساب لامبدا

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

حساب لامبدا

- 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 :

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

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.

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

- برای مثال :

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

حساب لامبدا

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

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

- برای مثال :

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

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

- سپس تابع G را به صورت زیر تعریف می‌کنیم :

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

- 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:

$$\mathtt{fact} \equiv YG$$
$$\mathtt{fact}\ n = (YG)n$$

حساب لامبدا

- برای مثال :

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

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.

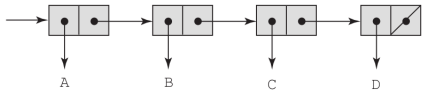
لیسپ

- 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.

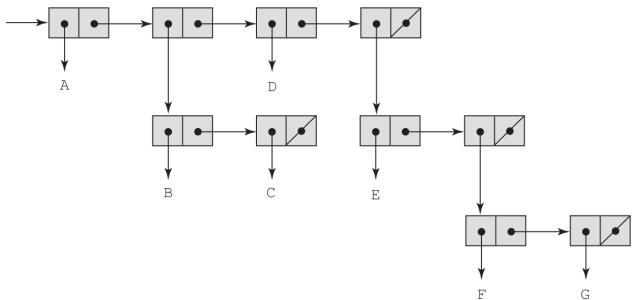
[1] linked list
[2] nil

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)    (*xx))
```

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

```
(( LAMBDA (x)    (*xx)) 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) نامگزاری 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 :

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

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.

```
(DEFINE  (inc n) (+ n 1))
```

- حال فراخوانی زیر را در نظر بگیرید:

```
(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.

```
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) 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.

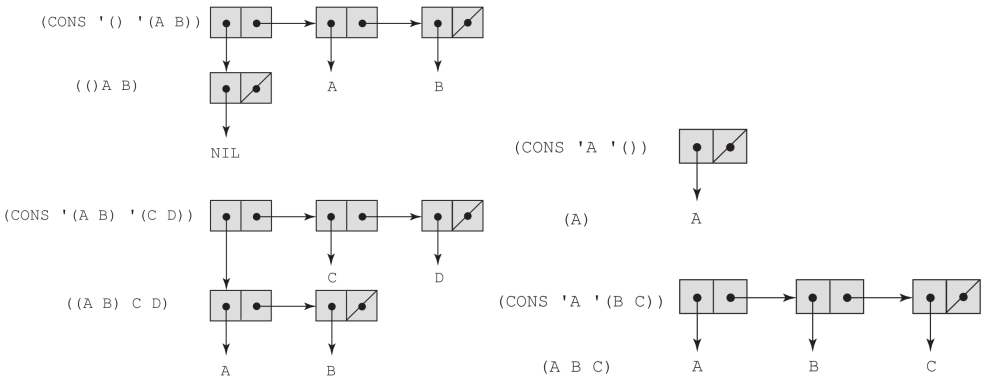
```
۱ (DEFINE (second lst) (CAR (CDR lst)))
```

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 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 ?EQV 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.

```
۱ (DEFINE (adder a_list)
۲   (COND
۳     ((NULL? a_list) 0)
۴     (ELSE (+ (CAR a_list) (adder (CDR a_list))))
۵   )
۶ )
```

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 یک زبان نوع دهی قوی است بدین معنی که 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.

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

[1] ML

ام‌ال

- می‌توانیم نوع بازگشتی 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 پارامتر متنوع 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 :

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

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

```
1 fun cube  x = x * x * x ;
2 val cubList = List.map cube ;
3 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

- زبان هسکل همانند ام‌ال یک زبان نوع دهندهٔ قوی است به این معنی که 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.

هسکل

- 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.

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

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

```
۱ 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.

- 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 می گرداند، 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

پایتون

- 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

پایتون

- 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

- 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.
```

- 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
```

[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.
- بنابراین تابع ورودی در تابع کاهش باید دو 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
```

راست

- یک بستار می‌تواند به متغیر هایی در حوزهٔ تعریف خود دسترسی داشته باشد که این دسترسی به سه نوع می‌توناد وجود داشته باشد : قرض گرفتن بدون تغییر [1] ، قرض گرفتن با تغییر [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 بدون تغییر 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

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

```
۱ fn main() {
۲     let mut list = vec![1, 2, 3];
۳     println!("Before defining closure: {:?}", list);
۴     let mut borrows_mutably = || list.push(7);
۵     borrows_mutably();
۶     println!("After calling closure: {:?}", list);
۷ }
```

[1] captured variable

راست

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

- 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.

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

[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 v1_iter = v1.iter();
۴     assert_eq!(v1_iter.next(), Some(&1));
۵     assert_eq!(v1_iter.next(), Some(&2));
۶     assert_eq!(v1_iter.next(), Some(&3));
۷     assert_eq!(v1_iter.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 v1_iter = v1.iter();
۴     let total: i32 = v1_iter.sum();
۵     assert_eq!(total, 6);
۶ }
```

راست

- 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 let v1: Vec<i32> = vec![1, 2, 3];
۲ let v2: Vec<_> = v1.iter().map(|x| x + 1).collect();
۳ assert_eq!(v2, vec![2, 3, 4]);
```

راست

- 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.

راست

- برای مثال :

```
struct Shoe {
    size: u32,
    style: String,
}
fn shoes_in_size(shoes: Vec<Shoe>, shoe_size: u32) -> Vec<Shoe> {
    shoes.into_iter().filter(|s| s.size == shoe_size).collect()
}
```

راست

```
fn filters_by_size() {
    let shoes = vec![
        Shoe {
                size: 10,
                style: String::from("sneaker"),
        },
        Shoe {
                size: 13,
                style: String::from("sandal"),
        },
        Shoe {
                size: 10,
                style: String::from("boot"),
        },
    ];
    let in_my_size = shoes_in_size(shoes, 10);
}
```

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.

برنامه نویسی رویه‌ای

برنامه نویسی رویه‌ای

- برنامه نویسی رویه‌ای [1] بر اساس مفهوم فراخوانی رویه [2] است. یک برنامه در یک زبان برنامه نویسی رویه‌ای تشکیل شده است از تعدادی رویه که یکدیگر را فراخوانی می‌کنند. رویه‌ها بر خلاف توابع در برنامه نویسی تابعی دارای حالت هستند بدین معنی که خروجی یک رویه می‌تواند بسته به حالت سیستم به ازای یک ورودی یکسان تغییر کند. یک رویه تشکیل شده است از تعدادی انتساب متغیر که مقدار آنها می‌تواند درون رویه یا بیرون رویه تغییر کند و تعداد ساختار کنترلی [3] و حلقه [4] که در مورد آنها صحبت خواهیم کرد.
- بنابراین محاسبات در برنامه نویسی رویه‌ای توسط ارزیابی عبارت و انتساب مقادیر به متغیرها حاصل می‌شود. علاوه بر عبارات با استفاده از ساختارهای کنترلی می‌توان از بین چند مسیر کنترلی برای محاسبات یک مسیر را انتخاب کرد و با استفاده از حلقه‌ها اجرای دسته‌ای از عبارات محاسباتی را تکرار کرد.

[1] procedural programming
[2] procedure call
[3] control structure
[4] loop

ساختارهای کنترلی

- ساختارهای کنترلی برای اولین بار در زبان فورترن به وجود آمدند که برای معماری ماشین آی‌بی‌ام ۷۰۴ به وجود آمد. زبان این ماشین نیز ساختارهای کنترلی را شامل می‌شد که در زبان فورترن به دستور `goto` ترجمه می‌شد.
- در دهۀ ۱۹۶۰ اثبات شد که همۀ برنامه‌ها تنها توسط دو ساختار کنترلی یکی برای انتخاب بین چند مسیر پردازش و دیگری برای تکرار دسته‌ای از محاسبات می‌توانند ساخته شوند و این دو ساختار نیاز همه برنامه‌ها را برآورده می‌کند، علاوه بر اینکه دستور `goto` که برای ارجاع کنترل برنامه به یکی از دستورات برنامه طراحی شده بود می‌توانست خطرساز باشد. دلیل این خطرسازی این بود که برنامه نویس می‌توانست کنترل برنامه را به هر جایی که می‌خواست ارجاع دهد و این باعث پیچیدگی برنامه می‌شد و می‌توانست احتمال بروز خطا در منطق برنامه را افزایش دهد. علاوه بر این، با استفاده از ساختارهای کنترلی جدید، خوانایی برنامه‌ها بیشتر می‌شد.

برنامه نویسی رویه‌ای

- یک ساختار کنترلی انتخاب می‌کند که کدام یک از مجموعه دستورات برای اجرا انتخاب شوند.
- یک عبارت انتخاب [1] وسیله‌ای است برای انتخاب یکی از مسیرها در اجرای یک برنامه.
- در بیشتر زبان‌ها انتخاب کننده دو حالته توسط `if-then-else` و انتخاب کننده چند حالته توسط `switch-case` پیاده سازی شده است.
- یک عبارت تکرار [2] وسیله‌ای است برای تکرار مجموعه‌ای از عبارت به تعداد صفر، یک یا چند بار.
- ساختار تکرار معمولاً حلقه نامیده می‌شود که در بیشتر برنامه‌ها با `for` و `while` و `foreach` پیاده سازی شده است.

[1] selection statement
[2] iterative statement

برنامه نویسی رویه‌ای

- زبان پایتون یک سازوکار جدید برای استفاده در ساختارهای تکرار مهیا کرده است.
- توابع معمولی در پایتون و دیگر زبان‌های برنامه نویسی یک مقدار را محاسبه کرده و بازمی‌گردانند. حال فرض کنید در یک حلقه نیاز داریم تابعی را فراخوانی کنیم که به ازای هر بار فراخوانی یک مقدار جدید را محاسبه کرده، بازمی‌گرداند. بدین ترتیب دنباله‌ای از مقادیر محاسبه شده به ترتیب از تابع بازگردانده می‌شوند.
- چنین ساز و کاری توسط مولدها [1] در پایتون قابل پیاده سازی است.
- در توابع معمولی بعد ازینکه مقداری توسط یک تابع بازگردانده می‌شود، متغیرهای محلی از بین می‌روند و در فراخوانی بعدی تابع، متغیرها دوباره ساخته و مقدار دهی می‌شوند. اما در مولدها پس از اینکه مقداری از یک مولد بازگردانده شد، متغیرهای محلی از بین نمی‌روند و در فراخوانی بعدی، مولد به عملیات ادامه می‌دهد.

[1] generators

برنامه نویسی رویه‌ای

- مولد زیر را در نظر بگیرید :

```
1 def generate_ints(N) :
2     for i in range(N) :
3         yield i
```

- هر تابعی که از کلمه yield استفاده کند، یک مولد است. با بازگرداندن یک مقدار توسط کلمه yield اجرای تابع به اتمام نمی‌رسد بلکه متوقف می‌شود و در فراخوانی بعدی ادامه پیدا می‌کند. فراخوانی‌های بعدی توسط تابع ()next انجام می‌شوند. تابع مولد در واقع یک پیمایشگر بازمی‌گرداند.

```
1 gen = generate_ints (3)
2 next (gen)   # 0
3 next (gen)   # 1
4 for i in generate_ints (5) :
5     print (i)
```

برنامه نویسی رویه‌ای

- به عنوان مثال دیگر فرض کنید مولدی می‌خواهیم که رئوس یک گراف را پیمایش کند و در هر بار فراخوانی، رأس پیمایش شدهٔ بعدی را بازگرداند.
- می‌توانیم مولدی به صورت زیر بنویسیم :

```
۱ def traverse (G) :
۲     # if there is another node :
۳     #       set nd to the next node
۴     # else :
۵     #       return
۶     yield nd
```

- در هر بار فراخوانی توسط تابع `next()` تابع `traverse` یک رأس پیمایش شده را بازمی‌گرداند.

زیر برنامه‌ها

- زیر برنامه‌ها [1] که رویه [2] یا تابع [3] نیز نامیده می‌شوند، به برنامه نویسان کمک می‌کنند تا مجموعه‌ای از محاسبات را جدا کرده و با نامی انتزاعی تعریف کنند تا این دسته از محاسبات بتوانند دوباره مورد استفاده قرار بگیرند. علاوه بر این که زیر برنامه‌ها باعث صرفه جویی در زمان برنامه نویسی و مصرف حافظه در برنامه می‌شوند، خوانایی برنامه را نیز بهبود می‌دهند.

- ویژگی‌های همهٔ زیر برنامه‌ها (به غیر از زیر برنامه‌های موازی که در برنامه نویسی همروند استفاده می‌شوند) به شرح زیر است :
۱. هر زیر برنامه در یک نقطه مقداردهی اجرای (فراخوانی) آن آغاز می‌شود.
۲. اجرای برنامه‌ای که یک زیر برنامه را فراخوانی می‌کند متوقف می‌شود تا زیر برنامه محاسبات خود را انجام دهد، بنابراین در هر لحظه فقط یک زیر برنامه در حال اجراست.
۳. پس از اتمام اجرای زیربرنامه، کنترل اجرای محاسبات به برنامه‌ای که زیربرنامه را فراخوانی کرده است بازگردانده می‌شود.

[1] subprograms
[2] procedure
[3] function

زیر برنامه‌ها

- تعریف زیر برنامه [1] توصیف می‌کند چه عملیاتی توسط یک زیر برنامه انجام می‌شود. در تعریف یک زیر برنامه یک نام انتزاعی برای زیر برنامه تعیین می‌شود و تعدادی متغیر ورودی برای زیر برنامه تعریف می‌شوند. در زبان‌هایی که نوع متغیرها در زمان کامپایل مشخص می‌شوند، نوع ورودی‌ها و خروجی‌ها نیز مشخص می‌شوند.
- فراخوانی زیر برنامه [2] درخواستی است که توسط یک برنامه یا یک زیر برنامهٔ دیگر برای اجرای یک زیر برنامه صادر می‌شود.
- نام یک زیر برنامه به همراه ورودی و خروجی‌های آن اعلام زیر برنامه [3] برنامه نامیده می‌شود.

[1] subprogram definition
[2] subprogram call
[3] declaration

زیر برنامه‌ها

- در برخی زبان‌ها تعریف تابع از اعلام آن جدا می‌شود و اعلام زیر برنامه در یک فایل جداگانه قرار می‌گیرد. دو دلیل برای این کار وجود دارد، یکی اینکه برنامه نظم بیشتری پیدا می‌کند و به اعلام زیر برنامه‌ها (به عنوان مستندات راهنما) راحت‌تر می‌توان دسترسی پیدا کرد. دلیل دوم این است که گاهی یک برنامه تجاری است و تعریف زیر برنامه‌ها نمی‌تواند در اختیار کاربران قرار بگیرد ولی اعلام آن جهت استفاده باید در دسترس باشد.
- در زبان سی و سی++ به اعلام زیر برنامه‌ها پروتوتایپ گفته می‌شود.

زیر برنامه‌ها

- برای اینکه زیر برنامه‌ها بتوانند به متغیرهایی که غیر محلی هستند (توسط خود زیر برنامه تعریف نشده است) دسترسی پیدا کنند، دو روش وجود دارد. در روش اول زیر برنامه به متغیر به طور مستقیم دسترسی دارد، بدین معنی که متغیری به طور عمومی تعریف شده و زیر برنامه به آن به طور مستقیم دسترسی دارد. در روش دوم متغیرهای مورد نیاز زیر برنامه به عنوان ورودی به زیر برنامه فرستاده می‌شوند.
- استفاده از متغیرهای عمومی خوانایی برنامه و همچنین قابلیت اطمینان برنامه را پایین می‌آورند زیرا دنبال کردن و تست برنامه سخت‌تر می‌شود چرا که زیر برنامه فقط به ورودی‌هایش بستگی ندارد بلکه ممکن است زیر برنامه‌های دیگر یک متغیر عمومی را تغییر دهند و نتیجه یک زیر برنامه تغییر کند. در زبان‌های تابعی امکان تعریف متغیر وجود ندارد که باعث افزایش قابلیت اطمینان و ساده‌تر شدن برنامه‌ها برای تحلیل و بررسی می‌شود.

زیر برنامه‌ها

- در برخی از زبان‌ها می‌توان در یک زیر برنامه به عنوان ورودی به جای داده، یک زیربرنامه دریافت کرد. بنابراین در چنین حالتی ورودی زیربرنامه یک متغیر است که می‌تواند به نام یکی از زیربرنامه‌ها ارجاع داده شود.
- ورودی زیر برنامه‌ها در تعریف زیر برنامه پارامتر [1] نیز نامیده می‌شوند. زمان انقیاد حافظهٔ پارامترهای یک زیر برنامه در هنگام فراخوانی زیر برنامه است.
- در زمان فراخوانی یک زیر برنامه نام زیر برنامه به همراه ورودی‌های آن مشخص می‌شوند. ورودی یک برنامه در هنگام فراخوانی آرگومان [2] های زیربرنامه نامیده می‌شوند.

[1] parameter
[2] argument

زیر برنامه‌ها

- در بیشتر زبان‌های برنامه نویسی تناظر آرگومان‌ها و پارامترها یا به عبارت دیگر انقیاد مقدار پارامترها با استفاده از مقدار آرگومان‌ها توسط موقعیت آنها در فراخوانی انجام می‌شود، بدین معنی که مقدار اولین پارامتر به مقدار اولین آرگومان مقید می‌شود و بدین ترتیب الی آخر.
- وقتی تعداد پارامترها زیاد می‌شود، ممکن است این نوع فراخوانی باعث ایجاد خطا در برنامه نویسی شود.

زیر برنامه‌ها

- در برخی زبان‌ها مانند پایتون می‌توان آرگومان‌ها را به پارامترها منسوب‌کرد و بدین صورت ترتیب آرگومان‌ها در هنگام فراخوانی بی‌اهمیت می‌شود.
- برای مثال :

```
1 def logger (time, message) :
2       print ("[" + time + "]" + message)
3 logger (message = "error" , time = "7am")
```

- همچنین می‌توان برای یک پارامتر مقدار پیش‌فرض [1] نیز تعیین کرد.

```
1 def compute_pay (incom , exemption = 1 , tax_rate) :
2         ...
3 pay = compute_pay (2000 , tax_rate = 0.15)
```

- توجه کنید که در اینجا آرگومان اول به پارامتر اول مقید می‌شود و آرگومان دوم به پارامتر سوم که نام پارامتر برای آن مشخص شده است.
- در زبان سی++ که تعیین پارامتر در زمان فراخوانی امکان پذیر نیست، پارامترها با مقادیر پیش فرض باید در پایان لیست پارامترها تعریف شوند.

[1] default

زیر برنامه‌ها

- در یک تقسیم بندی به زیر برنامه‌هایی که مقداری را بازنمی‌گردانند رویه [1] و به زیر برنامه‌هایی که مقدار بازمی‌گردانند تابع [2] گفته می‌شود ولی در بیشتر زبان‌ها همه زیر برنامه‌ها مقدار نیز بازمی‌گردانند.
- تنها زبان‌های قدیمی مانند فورترن و آدا رویه بدین معنا دارد.

[1] procedure
[2] function

زیر برنامه‌ها

- در زبان‌هایی که نوع متغیرها در زمان اجرا تعیین می‌شوند، یک زیر برنامه می‌تواند انواع متفاوتی از ورودی‌ها را بپذیرد و به عبارتی زیر برنامه به صورت عمومی تعریف می‌شود.
- برای مثال تابع زیر می‌تواند رشته یا عدد دریافت کند.

```
1 def add (a, b) : return a+b
2 add(2,3) # 5
3 add(2.3, 4.5) # 6.8
4 add("hello, ", "world") # "hello, world"
```

زیر برنامه‌ها

- در زبان‌هایی که نوع متغیرها در زمان کامپایل تعیین می‌شود، نوع متغیر ورودی تابع نمی‌تواند تغییر کند، مگر اینکه سازوکار برنامه‌نویسی عمومی در آن زبان وجود داشته باشد.
- برای مثال با استفاده از برنامه‌نویسی عمومی تابع زیر می‌تواند رشته یا عدد دریافت کند.

```
۱  template <class T>
۲  T add(T a, T b) {
۳      return a+b;
۴  }
۵  int main() {
۶      int x = add<int>(2,3);
۷      float y = add<float>(2.3, 4.5);
۸      string s = add<string>("hello ", "world");
۹      cout << x << y << s << endl;
۱۰ }
```

زیر برنامه‌ها

- در برخی از زبان‌ها مانند پایتون یک زیر برنامه می‌تواند در یک زیر برنامه تعریف شود، اما در برخی زبان‌ها مانند سی این کار امکان پذیر نیست.
- برای مثال تابع private تنها در تابع f قابل دسترس است.

```
1 def f(...) :
2     ...
3     def private(...) :
4         ...
5     private(...)
6     ...
```

- در زبان سی در یک تابع نمی‌توان تابع تعریف کرد و در نتیجه توابع تعریف شده توسط همهٔ توابع دیگر در دسترس هستند.

زیر برنامه‌ها

- همچنین برخی از زبان‌ها مانند پایتون اجازه می‌دهند یک تابع به عنوان آرگومان به تابع دیگر ارسال شود، اما در برخی زبان‌ها مانند سی این کار تنها توسط اشاره‌گر به تابع امکان پذیر است.

- برای مثال فرض کنید می‌خواهیم برنامه‌ای بنویسیم که یک تابع را دریافت کند و انتگرال آن را در بازهٔ a و b محاسبه کند. در پایتون می‌توانیم این برنامه را به صورت زیر بنویسیم.

```
1  def integral(f, a, b, d=0.001) :
2    return reduce(add, (map(lambda x : f(x)*d, np.arange(a,b,d))))
3
4  cube = lambda x : x*x*x
5
6  integral(cube, 0, 10)
7  # 2499.5000250000016
```

زیر برنامه‌ها

- در زبان سی برای ارسال تابع به تابع باید از اشاره‌گر به تابع [1] استفاده کنیم. برای این کار باید از امضای تابع [2] استفاده کنیم. امضای تابع مشخص می‌کند یک تابع چند ورودی از چه نوع‌های داده دارد و نوع دادهٔ خروجی آن چیست.
- برای مثال یک اشاره‌گر به تابع با دو ورودی عدد صحیح و اعشاری و یک خروجی بولی را به صورت زیر تعریف می‌کنیم. سپس این اشاره‌گر به تابع را می‌توانیم با نام یک تابع مقداردهی کنیم. پس نام توابع در واقع اشاره‌گر به تابع هستند.

```
۱ bool function(int i, double d) {
۲     // ...
۳ }
۴
۵ bool (*ptr) (int, double);
۶ ptr = function;
```

[1] function pointer
[2] function signature

زیر برنامه‌ها

- فرض کنید می‌خواهیم به چند تابع توسط آرایه‌ای از اشاره‌گرها به توابع دسترسی پیدا کنیم.

```
۱  double add(double x, double y) { return x+y; }
۲  double sub(double x, double y) { return x-y; }
۳  double mul(double x, double y) { return x*y; }
۴  double div(double x, double y) { return x/y; }
۵
۶  int main() {
۷      double (*op[])(double, double) = { add, sub, mul, div };
۸      int i;
۹      double a,b;
۱۰     scanf("%d", i);
۱۱     scanf("%f", a); scanf("%f", b);
۱۲     if (i>=0 && i<4)
۱۳         op[i](a, b);
۱۴     return 0;
۱۵ }
```

زیر برنامه‌ها

- همچنین می‌توانیم تابعی تعریف کنیم که در ورودی یک تابع را دریافت می‌کند. برای این کار از اشاره‌گر به تابع استفاده می‌کنیم.

```
۱  double operation(double x, double y, double(*op)(double, double)) {
۲      return op(x,y);
۳  }
۴
۵  int main() {
۶      double res;
۷      res = operation(8, 3, div);
۸
۹      return 0;
۱۰ }
```

زیر برنامه‌ها

- در برخی زبان‌ها امکان سربارگذاری زیر برنامه‌ها [1] وجود دارد، بدین معنی که چند زیر برنامه با نام یکسان می‌توانند پارامترها با نوع‌های متفاوت دریافت کنند.
- در برنامه زیر در فراخوانی اول تابع add تابع اول و در فراخوانی دوم تابع دوم اجرا می‌شود.

```
۱  void add(int a, int b) {
۲          cout << a+b;
۳  }
۴  void add(string a, string b) {
۵          cout << a+b;
۶  }
۷  int main() {
۸          add(5, 3);
۹          add("hello ", "world");
۱۰ }
```

[1] overload subprogram

زیر برنامه‌ها

- یک زیر برنامه عمومی [2] زیر برنامه‌ای است که در آن پارامترهای می‌توانند نوع عمومی داشته باشند. قالب‌ها در زبان سی++ [3] برای طراحی توابع با نوع داده‌ای عمومی تعریف شده‌اند و در زبان‌هایی مانند پایتون که نوع متغیرها به صورت ایستا تعریف نمی‌شود، زیر برنامه‌ها به صورت عادی عمومی هستند.

[2] generic subprogram
[3] template

زیر برنامه‌ها

- متغیرها در زیر برنامه‌ها محلی هستند بدین معنی که تنها در هنگام فراخوانی زیر برنامه به حافظه مقید می‌شوند. گاهی نیاز به متغیرهایی است که در فراخوانی‌های متفاوت یک زیر برنامه مقدار خود را از دست نمی‌دهند و در حافظه باقی می‌مانند. در زبان سی و سی++ تعریف چنین متغیرهایی توسط واژهٔ `static` امکان پذیر است. بدین ترتیب متغیر در حافظه پشته [1] متناظر با زیر برنامه ساخته نمی‌شود بلکه در قسمت داده‌های برنامه [2] تعریف می‌شود.

[1] stack memory
[2] data segment

روش‌های ارسال پارامتر

- روش‌های متعددی برای ارسال پارامتر به یک تابع زیر برنامه وجود دارد که در اینجا به آنها اشاره می‌کنیم.
- پارامترها می‌توانند سه مدل معنایی متفاوت داشته باشند. به عبارت دیگر پارامترها به سه روش متفاوت با آرگومان‌ها رفتار می‌کنند. در روش اول پارامترها تنها آرگومان‌ها را دریافت می‌کنند. چنین پارامترهایی تنها دارای حالت ورودی [1] هستند. در روش دوم پارامترها به آرگومان‌ها مقدار ارسال می‌کنند. این پارامترها تنها دارای حالت خروجی [2] هستند. در روش سوم، پارامترها هم از آرگومان‌ها مقدار دریافت می‌کنند و هم به آنها مقدار ارسال می‌کنند. این پارامترها دارای حالت ورودی و خروجی [3] هستند.

[1] in mod
[2] out mod
[3] in out mod

روش‌های ارسال پارامتر

- وقتی یک آرگومان با مقدار ارسال شده [1] یا به عبارت دیگر یک پارامتر با مقدار دریافت شده است، مقدار آرگومان به عنوان مقدار اولیه برای پارامتر در نظر گرفته می‌شود، اما آرگومان و پارامتر به دو مکان متفاوت در حافظه اشاره می‌کنند، بنابراین مقدار آرگومان در فضای مربوط به پارامتر کپی می‌شود.
- عیب ارسال با مقدار این است که ممکن است حجم یک متغیر زیاد باشد و کپی کردن آن سرعت اجرای برنامه را کاهش می‌دهد.
- متغیرهای اصلی زبان سی با مقدار ارسال می‌شوند.

[1] pass by value

روش‌های ارسال پارامتر

- وقتی یک آرگومان با نتیجه ارسال می‌شود [1] ، هیچ مقداری به زیر برنامه فرستاده نمی‌شود بلکه زیر برنامه مقدار متغیر را تغییر می‌دهد و نتیجه را به برنامه فراخوانی کنندهٔ زیر برنامه می‌فرستد. به عبارت دیگر پس از محاسبهٔ یک پارامتر مقدار آن در مقدار آرگومان کپی می‌شود.
- یکی از معایب ارسال با نتیجه این است که در مقدار دهی‌ها ممکن است تصادم [2] اتفاق بیافتد. فرض کنید یک متغیر دو بار به عنوان دو آرگومان در یک فراخوانی به یک زیر برنامه ارسال می‌شود. برای مثال `sub(p1,p1)` . حال در این فراخوانی `p1` دو بار مقدار دهی می‌شود ولی در نهایت مقداری را به خود می‌گیرد که برای بار آخر به آن داده شده است.

[1] pass by result
[2] collision

روش‌های ارسال پارامتر

- برای مثال تابع زیر در زبان سی شارپ را در نظر بگیرید. دو متغیر ورودی در این تابع ارسال با نتیجه می‌شوند.

```
1  void Set(out int x , out int y ) {
2        x = 17;
3        y = 35;
4  }
5  Set(out a , out a) ;
```

- مقدار a در نهایت برابر با ۳۵ می‌شود، گرچه ابتدا مقدار آن برابر با ۱۷ قرارگرفته است.
- به علت مشکلاتی که همه ارسال با نتیجه ایجاد می‌کند، بسیاری از زبان‌ها این روش را پشتیبانی نمی‌کنند.

روش‌های ارسال پارامتر

- ارسال با مقدار و نتیجه [1] ترکیبی از ارسال با مقدار و ارسال با نتیجه است.
- مقدار آرگومان ابتدا در پارامتر کپی می‌شود و پارامتر مقدار اولیه خود را می‌گیرد و در نهایت مقدار پارامتر در مقدار آرگومان کپی می‌شود.

[1] pass by value result

روش‌های ارسال پارامتر

- ارسال با ارجاع [1] روش دیگری برای پیاده سازی حالت ورودی و خروجی است.
- در این روش به جای کپی کردن مقدار آرگومان در پارامتر و سپس کپی کردن مقدار پارامتر در آرگومان که دارای سربار کپی است، مکان حافظه متغیر آرگومان به عنوان پارامتر به زیر برنامه ارسال می‌شود و زیر برنامه می‌تواند آن مکان حافظه را تغییر دهد.
- ارسال با ارجاع در سی و سی++ با استفاده از اشاره‌گرها انجام می‌شود و در سی++ با متغیر مرجع نیز این کار امکان پذیر است.

[1] pass by reference

روش‌های ارسال پارامتر

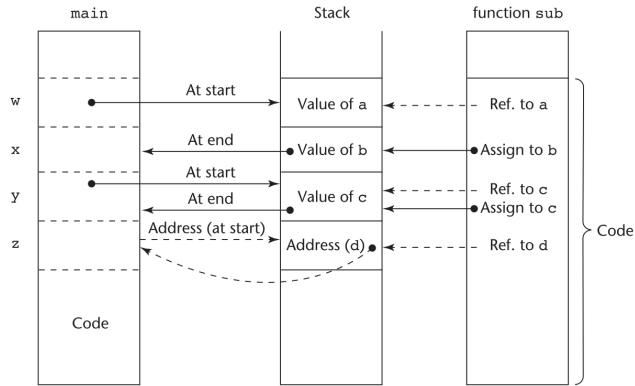
- ارسال با ارجاع چند مزیت دارد. مزیت اول صرفه جویی در زمان اجرا است چرا که در ارسال با ارجاع برای متغیرهای با حجم زیاد در زمان کپی صرفه جویی می‌شود و مزیت دوم صرفه‌جویی در حافظه است چرا که در زمان اجرا نیاز به تخصیص حافظه‌های اضافی وجود ندارد.
- یکی از معایب ارسال با ارجاع این است که ممکن است بخواهیم برای صرفه‌جویی در زمان از ارسال با ارجاع استفاده کنیم، اما نخواهیم زیر برنامه مقدار پارامتر را تغییر دهد. در زبان سی و سی++ برای این کار از اشاره‌گر ثابت استفاده می‌شود و در زبان سی++ با استفاده از مرجع ثابت نیز امکان پذیر است.

روش‌های ارسال پارامتر

- در ارسال پارامتر با مقدار در واقع مقدار آرگومان در پشته مربوط به زیر برنامه کپی می‌شود و زیر برنامه بر روی پشته به مقدار آرگومان دسترسی دارد.
- در ارسال پارامتر با ارجاع آدرس آرگومان در پشته مربوط به زیر برنامه کپی می‌شود و با اعمال تغییرات توسط زیر برنامه در آن مکان حافظه، مقدار متغیر نیز تغییر می‌کند که توسط برنامه فراخوانی کننده قابل مشاهده است.
- در ارسال پارامتر با نتیجه مقدار متغیر توسط زیربرنامه در پشته کپی می‌شود و برنامه مقدار مورد نظر را از پشته دریافت و در آرگومان کپی می‌کند.
- در ارسال پارامتر با مقدار و نتیجه برنامه مقدار آرگومان را در پشته کپی می‌کند، زیر برنامه برنامه محاسبات مورد نیاز را انجام می‌دهد و در نهایت نتیجه را در پشته کپی می‌کند و برنامه فراخوانی کننده نتیجه را در آرگومان کپی می‌کند.

روش‌های ارسال پارامتر

- روش پیاده سازی انواع ارسال پارامتر در شکل زیر نشان داده شده است.



Function header: **void** `sub (int a, int b, int c, int d)`
Function call in `main`: `sub (w,x,y,z)`
(pass `w` by value, `x` by result, `y` by value-result, `z` by reference)

روش‌های ارسال پارامتر

- در زبان سی ارسال با ارجاع توسط اشاره‌گرها پیاده سازی شده است که قبل از آن در زبان الگول نیز وجود داشت.
- اگر یک پارامتر یک اشاره‌گر ثابت باشد، فراخوانی سریع‌تر است چرا که در زمان کپی مقدار آرگومان در مقدار پارامتر صرفه جویی می‌شود.
- در زبان سی++ متغیر مرجع علاوه بر اشاره‌گر وجود دارد که در واقع یک نام مستعار برای یک مکان در حافظه است. پس از مقداردهی اولیه یک متغیر مرجع امکان مقدار دهی مجدد وجود ندارد بدین معنی که یک متغیر مرجع همیشه فقط به یک مکان حافظه اشاره می‌کند.
- در زبان جاوا متغیرها از نوع اصلی با مقدار و اشیاء از نوع کلاس‌ها با ارجاع ارسال می‌شوند، چرا که اشیاء همگی در واقع متغیر مرجع هستند.

روش‌های ارسال پارامتر

- در زبان پایتون برخی از متغیرها قابل تغییر [1] هستند و برخی دیگر غیر قابل تغییر [2] اند.
- لیست‌ها و دیکشنری‌ها و مجموعه‌ها قابل تغییراند ولی رشته‌ها و چندتایی‌ها غیر قابل تغییراند. همچنین نوع‌های عددی مانند اعداد صحیح و اعشاری غیر قابل تغییراند.

[1] mutable
[2] immutable

روش‌های ارسال پارامتر

- اگر یک متغیر قابل تغییر به یک تابع ارسال شود، ارسال با ارجاع است، بنابراین تابع می‌تواند مقدار آن را تغییر دهد.
- اگر تابع متغیر قابل تغییر را به مقدار جدید مقید کند یا به عبارت دیگر انقیاد مجدد [1] صورت بگیرد، و در نتیجه مکان حافظه تغییر کند، برنامه فراخوانی کننده از این انقیاد بی اطلاع خواهد بود. به عبارت دیگر، زیر برنامه‌ها می‌توانند متغیرهای قابل تغییر را تغییر دهند ولی مکان حافظه آنها را نمی‌توانند تغییر دهند.
- اگر یک متغیر غیر قابل تغییر به یک تابع ارسال شود تابع نمی‌تواند مقدار آن را تغییر دهد.

[1] rebind

روش‌های ارسال پارامتر

- برای مثال :

```
۱ def change (lst) :
۲      lst.append ('x')  #  value of list changes
۳      lst = ['y']  #  value of list does not change
۴ change (list)
```

```
۱ def func (s,i) :
۲     s = s + ' x '
۳     i = i + 1
۴ strval = ' hello '
۵ intval = 5
۶ change ( strval, intval )
۷ # value of strval and inval does not change
```

روش‌های ارسال پارامتر

- در زبان‌هایی که نوع آرگومان‌ها در برابر نوع پارامترها بررسی نمی‌شود، ممکن است خطاهایی در برنامه نویسی رخ دهد که برای پیدا کردن مشکل باشد.

- برای مثال فراخوانی تابع `result=sub1(1)` را در نظر بگیرید اگر پارامتر این تابع یک متغیر اعشاری باشد و به جای آن یک عدد صحیح به عنوان آرگومان ارسال شود، هیچ خطایی تشخیص داده نمی‌شود. اما از آنجایی که مقدار یک صحیح و یک اعشاری برابرند، مشکلی ایجاد نمی‌شود. اما اگر یک کاراکتر به عنوان ورودی ارسال شود، گرچه خطایی توسط کامپایلر صادر نمی‌شود و برنامه بدون خطا ادامه پیدا می‌کند اما ممکن است خطایی در محاسبات صورت گیرد که برای پیدا کردن و تصحیح کردن مشکل باشد.

- به عنوان یک مثال دیگر فرض کنید بررسی نوع [1] در پارامترهای توابع وجود نداشته باشد و یک آرگومان float به یک تابع که ورودی double می‌گیرد ارسال شود. اگر ارسال با مقدار باشد، مشکلی رخ نمی‌دهد اما اگر ارسال با ارجاع باشد، تابع در یک متغیر ۴ بایتی ۸ بایت داده ذخیره می‌کند. در خلال اجرای برنامه هیچ خطایی مشاهده نخواهد شد، اما نتیجه برنامه ممکن است با نتیجه دلخواه نابرابر باشد و در چنین مواردی پیدا کردن خطا بسیار مشکل خواهد بود.

[1] type checking

زیر برنامه به عنوان پارامتر

- در برخی مواقع نیاز است یک زیر برنامه به عنوان آرگومان به یک زیر برنامه دیگر ارسال شود.
- به عنوان مثال تابعی را در نظر بگیرید که یک تابع به عنوان ورودی دریافت می‌کند و انتگرال آن را در یک بازهٔ معین محاسبه می‌کند. اگر نتوانیم تابع به عنوان پارامتر ارسال کنیم برای هر تابع ریاضی باید یک تابع انتگرال‌گیر نیز پیاده سازی کنیم، که باعث افزایش هزینه زمانی برای نوشتن برنامه و همچنین افزایش حافظه مورد نیاز برای برنامه می‌شود.
- در زبان سی و سی++، اشاره‌گر به توابع [1] می‌توانند به عنوان آرگومان به توابع ارسال شوند. از آنجایی که در تعریف اشاره‌گر به تابع نوع‌های ورودی و خروجی تابع یا به عبارت دیگر امضای تابع مشخص می‌شود، بنابراین بررسی نوع در زمان کامپایل می‌تواند صورت بگیرد.
- برای مثال `float (*pfun)(int , double)` یک اشاره‌گر در ورودی یک تابع باشد، هر تابعی با این امضا می‌تواند به عنوان آرگومان یک تابع مد نظر ارسال شود.

[1] function pointer

زیر برنامه به عنوان پارامتر

- در زبان پایتون توابع می‌توانند به عنوان آرگومان به توابع دیگر ارسال شوند. به جای تابع در آرگومان می‌توان همچنین از یک عبارت لامبدا استفاده کرد. برای مثال دیدیم که توابع نگاشت و فیلتر و کاهش نیاز به دریافت تابع به عنوان ورودی دارند.

زیر برنامه به عنوان پارامتر

- مقدار بازگشتی توسط یک تابع در زبان سی می‌تواند هر نوعی به غیر از آرایه و تابع باشد. اما در زبان پایتون هر نوعی می‌تواند توسط یک تابع بازگردانده شود. یک تابع می‌تواند تابع نیز بازگرداند.
- در بیشتر زبان‌ها یک تابع می‌تواند تنها یک مقدار بازگرداند. با استفاده از نوع چندتایی در زبان پایتون می‌توان از یک مقدار (به عنوان مقادیر یک چندتایی) بازگرداند.

زیر برنامه به عنوان پارامتر

- یک تابع در برخی زبان‌ها مانند سی++ می‌تواند سربارگذاری [1] شود بدین معنا که می‌توان تعدادی تابع هم‌نام تعریف کرد که پارامترهایی از نوع‌های متفاوت دریافت کنند. همچنین در زبان سی++ می‌توان عملگرها را سربارگذاری کرد، بدین معنی که یک عملگر با توجه به عملوندهای آن عملیاتی متفاوت انجام دهد.
- برای مثال در سی++ می‌توانیم عملگر + را برای اشیای یک کلاس به صورت زیر تعریف کنیم :

```
۱ Complex operator +(Complex c1, Complex c2){
۲     return Complex ( c1.r + c2.r, c1.i + c2.i) ;
۳ }
```

[1] overload

زیر برنامه‌های عمومی

- یکی از معیارهای یک زبان برای ارزیابی، قابلیت آن برنامه برای نوشتن کد به صورت مختصر و بهینه است. در یک زبان هر چقدر عملیات بیشتری در حجم کمتر بتوان نوشت، راندمان برنامه نویسی در آن افزایش می‌یابد.
- برای مثال اگر در یک زبان قابلیتی وجود داشته باشد که عملیات یکسان بر روی نوع‌های متفاوت بتوانند تنها توسط یک تابع پشتیبانی شوند، راندمان برنامه نویسی افزایش می‌یابد.
- در برنامه نویسی شیءگرا خواهیم دید که انواع متفاوت که همگی از یک خانواده هستند می‌توانند توسط قابلیت چند ریختی [1] به یک تابع ارسال شوند.

[1] polymorphism

زیر برنامه‌های عمومی

- قابلیت دیگری که توسط برخی زبان‌ها پشتیبانی می‌شود، دریافت ورودی با نوع پارامتری است، بدین معنا که یک زیر برنامه علاوه بر دریافت ورودی‌ها به عنوان پارامتر، نوع ورودی‌ها را نیز به عنوان پارامتر دریافت می‌کند.
- در زبان سی++ این قابلیت توسط قالب‌ها [1] پشتیبانی می‌شود.
- در برخی زبان‌ها مانند پایتون که نوع متغیرها به طور صریح توصیف نمی‌شود، توابع همگی یک نوع عمومی دریافت می‌کنند.

[1] templates

زیر برنامه‌های عمومی

- برای مثال برای مقایسه دو مقدار از یک نوع عمومی در زبان سی++ می‌توانیم تابعی به صورت زیر بنویسیم :

```
template  < class Type>
Type max (Type first , Type second) {
      return first > second ? first : second ;
}
```

- چنانچه می‌خواستیم این تابع را بدون برنامه نویسی تعریف کنیم، باید به ازای هر نوع یک تابع جدید تعریف می‌کردید. به علاوه ممکن بود پس از تعریف این تابع در یک کتابخانه محاسباتی، یک برنامه نویس به مقایسه نوع‌هایی نیاز داشت که در زمان توسعه کتابخانه موجود نبودند. با استفاده از برنامه نویسی عمومی استفاده کننده کتابخانه محاسباتی نوع‌های جدید را می‌تواند به عنوان پارامتر به توابع کتابخانه ارسال کند، اما بدون برنامه نویسی عمومی لازم بود استفاده کننده کتابخانه از توسعه دهندهٔ کتابخانه بخواهد توابع جدید با نوع‌های جدید را به کتابخانهٔ محاسباتی اضافه کند.

پیوند زیر برنامه

- به عملیات فراخوانی و بازگرداندن مقدار یک زیر برنامه و اتصال آن به جریان برنامه اصلی پیوند زیر برنامه [1] گفته می‌شود.

- عملیاتی که در پیوند زیر برنامه، برای انتقال متغیرها انجام می‌شود، باید توصیف شده و توسط طراح کامپایلر پیاده سازی شوند. برای مثال متغیرهای محلی یک زیر برنامه بر روی پشته تخصیص داده می‌شوند و وضعیت اجرای [2] برنامه قبل از فراخوانی برنامه ذخیره می‌شود تا پس از فراخوانی ادامه پیدا کند. وضعیت اجرای برنامه شامل مقادیر رجیسترها و بیت‌های حالت پردازنده می‌شود. همچنین باید کنترل اجرا به زیر برنامه داده شود و اطمینان حاصل شود که پس از اجرای زیر برنامه کنترل به دستور بعد از فراخوانی بازمی‌گردد. اگر امکان پیاده سازی زیر برنامه‌های تودرتو وجود داشته باشد، باید اطمینان حاصل شود که زیر برنامه‌ها به متغیرهای غیر محلی به درستی دسترسی دارند. در بازگرداندن مقدار ، مقدار خروجی تابع و همچنین متغیرهایی که با ارجاع یا حالت خروجی ارسال شده‌اند باید در دسترس برنامه فراخوانی کننده قرار بگیرند.

[1] subprogram linkage
[2] execution status

پیوند زیر برنامه

- نحوهٔ پیوند یک زیر برنامهٔ ساده را بررسی می‌کنیم. در این زیر برنامهٔ ساده زیر برنامه تودرتو نمی‌تواند وجود داشته باشد و همچنین همهٔ متغیرها ایستا هستند. نسخه‌های اولیهٔ زبان فورترن به این شکل بودند.

- در زمان فراخوانی اعمال زیر باید انجام شوند :
۱. وضعیت اجرای برنامه جاری ذخیره شود.
۲. آرگومان‌ها به پارامترها ارسال شوند.
۳. آدرس بازگشت به زیر برنامهٔ فراخوانی شونده ارسال شود.
۴. کنترل اجرا به زیر برنامهٔ فراخوانی شونده ارسال شود.

- در زمان خاتمه و بازگشت زیر برنامه عملیات زیر باید انجام شوند :
۱. اگر پارامتری با حالت خروجی (برای مثال ارسال با ارجاع) وجود دارد، مقدار آن‌ها باید در دسترس برنامه فراخوانی کننده قرار بگیرد.
۲. اگر زیر برنامه دارای خروجی است، خروجی آن باید در دسترس برنامه فراخوانی کننده قرار بگیرد.
۳. وضعیت اجرای فراخوانی کننده به حالت قبل از فراخوانی باز می‌گردد.
۴. کنترل اجرا به فراخوانی کننده باز می‌گردد.

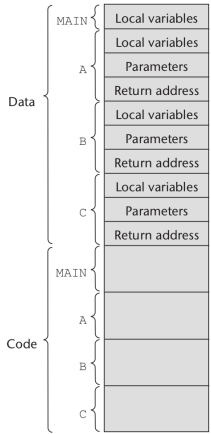
پیوند زیر برنامه

- بنابراین در اجرای یک زیر برنامه، اطلاعات در مورد وضعیت فراخوانی کننده، پارامترها، آدرس بازگشت و مقادیر بازگشتی از زیر برنامه باید ذخیره شوند. این اطلاعات به همراه متغیرهای محلی و کد زیر برنامه مجموعه‌ای از اطلاعات مورد نیاز یک زیر برنامه را تشکیل می‌دهند.
- یک زیر برنامه از دو بخش تشکیل شده است : کد زیر برنامه و متغیرهای محلی و داده‌های مورد نیاز.
- داده‌های یک زیر برنامه به همراه متغیرهای محلی رکورد فعالسازی [1] یک زیر برنامه گفته می‌شوند زیرا داده‌هایی که توصیف شدند در زمان فعالسازی زیر برنامه مورد نیازند. یک نمونه از رکورد فعالسازی [2] در واقع یک مثال یا یک نمونه از مقادیر از رکورد فعالسازی است.
- از آنجایی که در یک زیر برنامهٔ ساده همه مقادیر و داده‌ها ثابت هستند به طور ایستا می‌توانند تخصیص داده شوند. توجه کنید که امکان فراخوانی بازگشتی با استفاده از این روش وجود ندارد.

[1] activation record
[2] activation record instance

پیوند زیر برنامه

- شکل زیر رکورد فعالسازی برای سه زیر برنامه A و B و C را نشان می‌دهد.



پیوند زیر برنامه

- برنامه اجرایی شکل قبل توسط پیوند دهنده یا لینکر [1] ایجاد می‌شود.
- وقتی لینکر اجرا می‌شود، ابتدا همهٔ فایل‌های کامپایل شدهٔ زیر برنامه‌ها در حافظه قرار می‌گیرند. سپس برای هر فراخوانی زیر برنامه، آدرس آن در حافظه باید در مکان فراخوانی قرار بگیرد. همچنین به ازای هر فراخوانی زیر برنامه در درون یک زیر برنامه آدرس مکان ورود به زیر برنامه‌ها توسط لینکر در برنامه اجرایی قرار می‌گیرد.

[1] linker

پیوند زیر برنامهٔ حاوی متغیر پشته

- حال پیوند زیر برنامه را در زبانی بررسی می‌کنیم که متغیرهای محلی به صورت پویا بر روی پشته قرار می‌گیرند. یکی از مهم‌ترین مزیت‌های حافظه پویا بر روی پشته این است که توسط آن می‌توان توابع بازگشتی را پیاده سازی کرد.
- کامپایلر باید کدی تولید کند که تخصیص و آزاد سازی متغیرهای محلی را انجام دهد.
- همچنین در این حالت ممکن است تابعی به صورت بازگشتی فراخوانی شود و بنابراین بیش از یک نمونه از یک زیر برنامه در هر مرحله می‌تواند فعال باشد.
- در زبان‌هایی که دارای متغیرهای پویای بر روی پشته هستند نمونه رکورد فعالسازی به صورت پویا ساخته می‌شود.

پیوند زیر برنامهٔ حاوی متغیر پشته

- یک رکورد فعالسازی شامل آدرس بازگشت، لینک پویا [1] ، پارامترها و متغیرهای محلی زیر برنامه می‌شود.
- آدرس بازگشت اشاره‌گری است که به دستور بعد از فراخوانی زیر برنامه اشاره می‌کند.
- لینک پویا اشاره‌گری است که به نمونه رکورد فعالسازی برنامه یا زیر برنامه فراخوانی کننده اشاره می‌کند. در زبان‌های که حوزهٔ تعریف پویا دارند، از لینک پویا برای دسترسی به متغیرهای محلی توابع فراخوانی کننده یک تابع استفاده می‌شود. در زبان‌های که حوزهٔ تعریف ایستا دارند از این لینک برای دنبال کردن خطاها و گزارش آن توسط کامپایلر به برنامه نویس استفاده می‌شود.
- همچنین مقدار آرگومان‌ها در زمان فراخوانی در رکورد فعالسازی فراخوانی کننده قرار دارند که مقادیر آنها باید در مقدار پارامترهای تابع فراخوانی شوند کپی شود.

[1] dynamic link

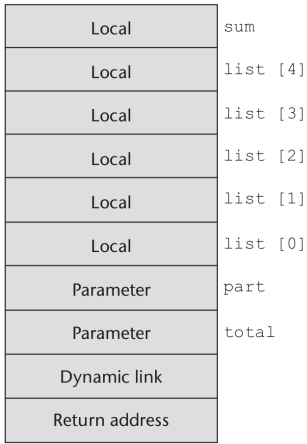
پیوند زیر برنامهٔ حاوی متغیر پشته

- تابع زیر را در نظر بگیرید.

```
۱ void sub (float total, int part) {
۲       int list [5];
۳       float sum;
۴ }
```

پیوند زیر برنامهٔ حاوی متغیر پشته

- رکورد فعالسازی این تابع در زیر نشان داده شده است.



پیوند زیر برنامهٔ حاوی متغیر پشته

- از آنجایی که آخرین تابع فراخوانی شونده اولین تابعی است که باید اجرا شود، طبیعی است که ساختار حافظه پشته باشد. به این پشته، پشته زمان اجرا [1] گفته می‌شود. هر زیر برنامه، رکورد فعالسازی خود را بر روی پشته می‌سازد و هر فراخوانی زیر برنامه رکورد مربوط به خود را دارد.
- اشاره‌گر محیطی [2] به رکورد فعالسازی آخرین زیر برنامه اشاره می‌کند. در هر بار فراخوانی یک زیر برنامه، زیر برنامه فراخوانی شده آدرس رکورد فعال سازی فراخوانی کننده را در لینک پویای خود نگهداری می‌کند و آدرس اشاره‌گر محیطی را به آدرس رکورد فعالسازی خود تغییر می‌دهد.

[1] run-time stack
[2] environment pointer (EP)

پیوند زیر برنامهٔ حاوی متغیر پشته

- بنابراین عملیات زیر از سوی فراخوانی‌کننده باید انجام شوند.
۱. یک نمونه رکورد فعالسازی ساخته می‌شود.
۲. وضعیت اجرای برنامه فعلی ذخیره می‌شود.
۳. آرگومان‌ها به پارامترها ارسال می‌شوند.
۴. آدرس بازگشت به فراخوانی شونده ارسال می‌شود.
۵. کنترل اجرای برنامه به فراخوانی شونده داده می‌شود.

پیوند زیر برنامهٔ حاوی متغیر پشته

- زیر برنامه فراخوانی شونده در شروع عملیات زیر را انجام می‌دهد.

۱. آدرس اشاره‌گر محیطی را در لینک پویا ذخیره می کند و آدرس رکورد فعالسازی خود را به اشاره‌گر محیطی می‌دهد.
۲. فضا برای متغیرهای محلی خود بر روی پشته تخصیص می‌دهد.

- زیر برنامه فراخوانی شونده در پایان کار عملیات زیر را انجام می‌دهد.

۱. پارامترها با حالت خروجی را در آرگومان‌ها کپی می‌کند.
۲. اگر زیر برنامه مقدار خروجی باز می‌گرداند، مقادیر را به تابع فراخوانی کننده بازمی‌گرداند.
۳. مقدار اشاره‌گر محیطی را برابر با مقدار لینک پویا قرار می‌دهد.
۴. وضعیت اجرای فراخوانی کننده را بازمی‌گرداند.
۵. کنترل اجرا را به فراخوانی کننده بازمی‌گرداند.

پیوند زیر برنامهٔ حاوی متغیر پشته

- وقتی اجرای یک زیر برنامه به پایان می‌رسد رکورد فعالسازی آن تخریب می‌شود بنابراین متغیرهای محلی آن دیگر در دسترس نیستند.

پیوند زیر برنامهٔ حاوی متغیر پشته

- حال برنامه زیر را در نظر بگیرید.

```
۱  void fun1(float r) {
۲      int s, t;
۳      ... // Point  1
۴      fun2(s);
۵      ...
۶  }
۷  void fun2(int x) {
۸      int y;
۹      ... // Point 2
۱۰     fun3(y);
۱۱     ...
۱۲ }
۱۳ void fun3(int q) {
۱۴     ... // Point 3
۱۵ }
```

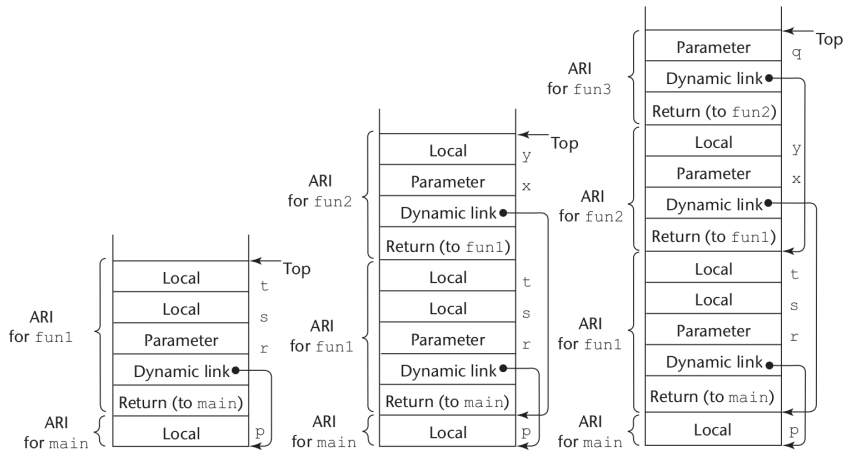
پیوند زیر برنامهٔ حاوی متغیر پشته

- تابع main تابع fun1 را فراخوانی می‌کند. سپس fun1 ، تابع fun2 و fun2 ، تابع fun3 را فراخوانی می‌کند.

```
۱۶ void main() {
۱۷     float p;
۱۸     ...
۱۹     fun1(p);
۲۰     ...
۲۱ }
```

پیوند زیر برنامهٔ حاوی متغیر پشته

- محتوای پشته برای این برنامه در زیر نشان داده شده است.



پیوند زیر برنامهٔ حاوی متغیر پشته

- مجموعه لینک‌های پویا، زنجیرهٔ پویا [1] یا زنجیرهٔ فراخوانی [2] نیز نامیده می‌شود. زنجیرهٔ فراخوانی تاریخچه فراخوانی توابع را نمایش می‌دهد.
- هر کدام از متغیرهای محلی در پشته یک آفست محلی [3] دارند که مکان آنها نسبت به شروع رکورد فعالسازی را مشخص می‌کند.

[1] dynamic chain
[2] call chain
[3] local offset

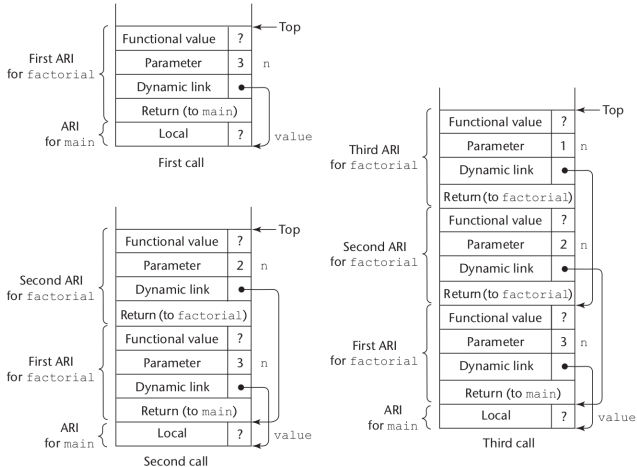
پیوند زیر برنامهٔ حاوی متغیر پشته

- حال برنامه زیر را در نظر بگیرید که در آن فراخوانی بازگشتی وجود دارد.

```
۱  int factorial(int n) {
۲      // Point 1
۳      if (n <= 1)
۴          return 1;
۵      else
۶          return (n * factorial(n - 1));
۷      // Point 2
۸  }
۹  void main() {
۱۰     int value;
۱۱     value = factorial(3);
۱۲     // Point 3
۱۳ }
```

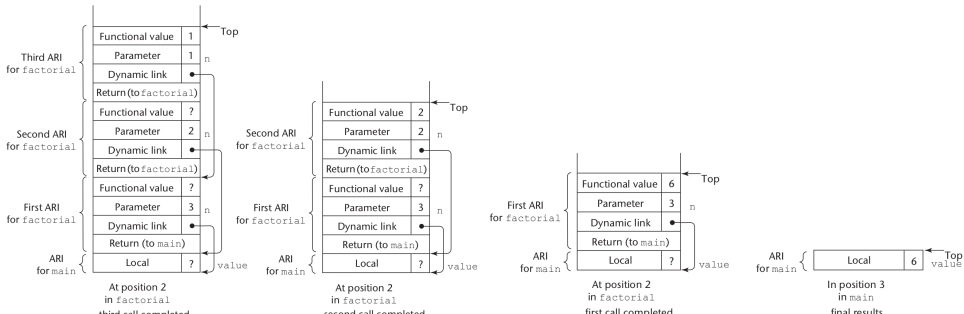
پیوند زیر برنامهٔ حاوی متغیر پشته

- محتوای پشته این برنامه تا رسیدن به نقطه اول در شکل زیر نشان داده شده است.



پیوند زیر برنامهٔ حاوی متغیر پشته

- محتوای پشته برای این برنامه تا رسیدن به نقطه دوم در شکل زیر نشان داده شده است.



زیر برنامه‌های تودرتو

- برخی از زبان‌ها مانند پایتون و روبی اجازه می‌دهند زیر برنامه‌های تودرتو ساخته شوند.
- در پیاده سازی این زبان‌ها علاوه بر لینک پویا که برای دنبال کردن زیر برنامه‌ها به کار می‌رود، یک مقدار دیگر به نام لینک ایستا [1] نیز استفاده می‌شود که برای دنبال کردن دسترسی متغیرها در توابع تودرتو به کار می‌رود. به لینک ایستا، اشاره‌گر حوزه تعریف ایستا [2] نیز گفته می‌شود که در واقع به رکورد فعال سازی زیر برنامه پدر اشاره می‌کند.
- همچنین زنجیرهٔ ایستا [3] به دنبالهٔ لینک‌های ایستا که به یکدیگر متصل شده‌اند گفته می‌شود.
- برای پیدا کردن مقادیر متغیرهای غیر محلی، باید زنجیرهٔ ایستا را دنبال کرد و در زیر برنامهٔ پدر یا اجداد مقدار متغیر غیر محلی را جستجو کرد.

[1] static link
[2] static scope pointer
[3] static chain

زیر برنامه‌های تودرتو

- برنامه زیر را در نظر بگیرید.

```
۱  function main(){
۲      var x;
۳      function bigsub() {
۴          var a, b, c;
۵          function sub1 {
۶              var a, d;
۷              ...
۸              a = b + c; // Point 1
۹              ...
۱۰         } // end of sub1
```

زیر برنامه‌های تودرتو

```
۱۱          function sub2(x) {
۱۲              var b, e;
۱۳              function sub3() {
۱۴                  var c, e;
۱۵                  ...
۱۶                  sub1();
۱۷                  ...
۱۸                  e = b + a; // Point 2
۱۹              } // end of sub3
۲۰              ...
۲۱              sub3();
۲۲              ...
۲۳              a = d + e; // Point 3
۲۴          } // end of sub2
```

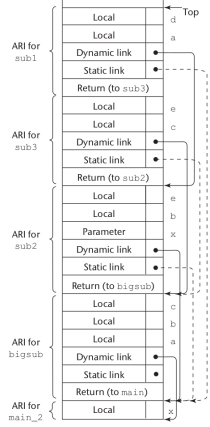
زیر برنامه‌های تودرتو

```
1          ...
2          sub2(7);
3          ...
4      } // end of bigsub
5      ...
6      bigsub();
7      ...
8  } // end of main
```

- در این برنامه تابع main تابع bigsub را فراخوانی می‌کند، سپس bigsub تابع sub2 و sub2 تابع sub3 و sub3 تابع sub1 را فراخوانی می‌کند.
- از طرفی sub1 و sub2 در تابع bigsub تعریف شده‌اند و sub3 در تابع sub2 .

زیر برنامه‌های تودرتو

- پشته فراخوانی برای این برنامه در شکل زیر نشان داده شده است.



بلوک‌ها

- در برخی از زبان‌ها مانند سی می‌توان بلوک‌هایی [1] برای حوزهٔ تعریف متغیرها تعریف کرد.
- برای مثال با بازکردن یک آکولاد بلوک جدیدی ساخته می‌شود که متغیرهای آن بلوک فقط درون بلوک تعریف شده‌اند و خارج از آن قابل دسترسی نیستند.
- بلوک‌ها را نیز می‌توان توسط زنجیره‌های ایستا پیاده سازی کرد. هر بلوک می‌تواند به عنوان یک زیر برنامه بدون پارامتر در نظر گرفته شود.

[1] blocks

برنامه نویسی شیءگرا

انتزاع داده

- برنامه نویسی شیءگرا [1] یک پارادایم (الگوواره) برنامه نویسی بر مبنای مفهوم شیء [2] است. یک شیء در واقع یک نمونه از یک نوع داده است که ویژگی‌ها و رفتارهایی دارد. در برنامه نویسی شیءگرا نوع داده‌ای که دارای ویژگی‌ها و رفتارهای تعریف شده است، کلاس نامیده می‌شود. ویژگی‌های یک کلاس خصوصیاتی هستند که با نوع‌های دیگر داده‌ای قابل توصیف است و رفتارهای یک کلاس توابعی هستند که برای آن کلاس تعریف شده‌اند.
- یک کلاس در واقع یک نوع دادهٔ انتزاعی است و کلاس‌ها می‌توانند در سطوح متفاوت انتزاعی قرار بگیرند.

[1] object-oriented programming
[2] object

انتزاع داده

- انتزاع داده [1] در واقع روشی است برای نمایش یک موجودیت به طوری که تنها ویژگی‌های مهم آن موجودیت نمایش داده شود.
- به عبارت دیگر با استفاده از انتزاع داده می‌توان نمونه‌های متفاوت از موجودات را بر اساس ویژگی‌های مشترک آنها در یک دسته قرار داد.

[1] data abstraction

انتزاع داده

- برای مثال وقتی از کلمهٔ پرنده استفاده می‌کنیم در واقع موجودات بسیار زیادی را بر اساس ویژگی‌هایشان در یک دسته قرار داده‌ایم. همهٔ جانداران که دو بال دارند و دو پا دارند و بدنشان از پر پوشیده شده است را با کلمهٔ پرنده در یک گروه قرار داده‌ایم. حال در یک زبان برنامه نویسی می‌توانیم این کلمه را با ویژگی‌های آن توصیف کنیم که در واقع یک نوع داده‌ای انتزاعی جدید تعریف کرده‌ایم. حال اگر بگوییم باز یک پرنده است، می‌دانیم که باز هم ویژگی‌های همه پرندگان را داراست. حال ممکن است در یک سطح پایین‌تر انتزاع، دسته پرندگانی را تعریف کنیم که پرواز می‌کنند و همچنین دسته پرندگانی که پرواز نمی‌کنند. پرندگانی که پرواز نمی‌کنند همه ویژگی‌های پرندگان را دارا هستند و علاوه بر آن ویژگی‌های دیگری نیز دارند. در یک سطح پایین‌تر انتزاع می‌توانیم کلاسی برای همهٔ پنگوئن‌ها تعریف تعریف کنیم. یک پنگوئن همه ویژگی‌های پرندگانی که پرواز نمی‌کنند و همهٔ ویژگی‌های پرندگان را داراست و علاوه بر آن ویژگی‌های دیگر هم دارد مثلاً اینکه از ماهی تغذیه می‌کند.
- حال یک پنگوئن واقعی را در نظر بگیرید این پنگوئن قد و وزن و ویژگی‌های خاص خود را دارد و از کلاس پنگوئن‌هاست.

انتزاع داده

- در زبان‌های برنامه نویسی دو نوع انتزاع وجود دارد : انتزاع پروسه یا فرایند [1] و انتزاع داده. انتزاع فرایند بدین معنا است که فرایندی با محاسبات مشخص را دسته‌بندی کرده و با یک نام فرایند که زیر برنامه نیز نامیده می‌شود نامگذاری کنیم و پس از آن، آن فرایند را با استفاده از نام و پارامترهای آن استفاده کنیم.
- وقتی می‌خواهیم یک لیست را مرتب کنیم از هر دو نوع انتزاع استفاده می‌کنیم. لیست یک نوع انتزاعی با ویژگی‌های مشخص و تعریف شده است و مرتب سازی، یک عملیات مشخص و تعریف شده است. در زبان‌های طبیعی اسم‌ها داده انتزاعی هستند و فعل‌ها فرایند انتزاعی چرا که یک عملیات مشخص را نامگذاری می‌کنند.

[1] process abstraction

انتزاع داده

- اولین نسخه دادهٔ انتزاعی در زبان کوبول در سال ۱۹۶۰ برای تعریف ساختار داده‌ها به وجود آمد و بعدها مفهوم ساختمان یا استراکت [1] در زبان سی نیز مورد استفاده قرار گرفت.
- یک نوع داده تعریف شده توسط کاربر به وسیلهٔ ساختمان در واقع ویژگی‌هایی را تجمیع می‌کرد که بدین ترتیب متغیرهایی از آن نوع داده انتزاعی می‌توانستند به توابع نیز ارسال شوند.
- یک نمونه از یک نوع انتزاعی در برنامه نویسی شیءگرا یک شیء نامیده می‌شود.

[1] struct

انتزاع داده

- دقت کنید که حتی نوع داده‌های اصلی در یک زبان برنامه نویسی هم انتزاعی هستند. مثلا نوع دادهٔ اعشاری همهٔ اعدادی را که به نحو معینی بر روی حافظه ذخیره می‌شوند و ویژگی‌های مشترکی دارند را با عدد اعشاری یا ممیز شناور [1] تعریف می‌کند.

[1] floating point

انتزاع داده

- یک نوع دادهٔ انتزاعی [1] نوع داده‌ای است که در تعریف آن از تعدادی ویژگی‌ها با نوع داده‌های دیگر استفاده شده و دارای رفتارهایی است. ویژگی‌های خصوصی یک نمونه از آن نوع داده‌ای که شیء نامیده می‌شود از استفاده کنندهٔ شیء پنهان است و استفاده کنندهٔ شیء تنها به رفتارهای عمومی شیء دسترسی دارد.
- یک کاربر شیء انتزاعی نیازی به دانستن جزئیات پیاده سازی آن نوع داده انتزاعی ندارد بلکه کافی است نام نوع داده و توابع (رفتارهای) عمومی که برای آن تعریف شده را بشناسد تا بتواند از آن استفاده کند.
- در زبان سی از ساختمان‌ها برای تعریف داده‌های انتزاعی استفاده می‌شد و توابعی برای استفاده از آن نوع داده‌ها تعریف می‌شد.

[1] abstract data type

انتزاع داده

- در برنامه نویسی شیءگرا نوع داده‌های انتزاعی کلاس نامیده می‌شوند و هر نمونه از یک کلاس یک شیء نامیده می‌شود.
- یک کلاس علاوه بر تعریف ویژگیهای آن کلاس، تعدادی رفتار نیز برای کلاس تعریف می‌کند. یک رفتار در واقع یک تابع است که بر روی یک شیء عملیاتی انجام می‌دهد.
- ویژگی‌های یک کلاس معمولاً به صورت خصوصی تعریف می‌شوند و از استفاده کننده کلاس پنهان هستند و تنها با استفاده از توابع (رفتارهای) تعریف شده بر روی کلاس و رابطهای کاربری تعریف شده می‌توان ویژگی‌های کلاس را تغییر داد.
- بر خلاف زبان‌های غیر شیءگرا که در آنها نوع‌های داده‌ای انتزاعی جدا از توابعی هستند که بر روی آنها اعمال می‌شوند، در زبان‌های شیءگرا ویژگی‌ها و رفتارها در کنار یکدیگر در نوع داده‌ای مورد نظر تعریف می‌شوند.

انتزاع داده

- این مخفی سازی اطلاعات [1] در کلاس‌ها چندین مزیت دارد.
- مزیت اول این مخفی سازی اطلاعات بالا بردن قابلیت اطمینان برنامه است. اگر یک کاربر بتواند اطلاعاتی را در یک نوع داده تغییر دهد، این تغییر ممکن است باعث ایجاد اختلال در منطق داده مورد نظر شود. فرض کنید یک نوع دادهٔ جدید برای تعریف یک موجودیت ایجاد کرده‌ایم. فرض کنید این موجودیت یک صف باشد. نوع دادهٔ مورد نظر باید تعدادی ویژگی به عنوان اطلاعات مورد نیاز این صف نگهداری کند. برای مثال اندیس ابتدای صف در یک آرایه می‌تواند یکی از این اطلاعات باشد. حال اگر کاربر بتواند این اطلاعات را تغییر دهد، عملیات درج و برداشت از صف مورد نظر به درستی عمل نخواهند کرد. بنابراین در برنامه نویسی شیءگرا معمولا ویژگی‌ها پنهان می‌شوند و اطلاعاتی که باید در دسترس قرار بگیرند توسط تعدادی تابع به عنوان واسط کلاس در دسترس کاربران قرار می‌گیرند.
- مزیت دوم ساده شدن برنامه نویسی با کاهش جزئیات برای کاربر است.

[1] information hiding

انتزاع داده

- مزیت سوم بهبود قابلیت تغییر برنامه است به نحوی که برنامه کاربران تحت تاثیر قرار نگیرد. به عبارت دیگر جزئیات پیاده سازی یک نوع داده‌ای از کاربر پنهان می‌ماند و کاربر تنها توسط واسط‌هایی از کلاس‌های مورد نیاز خود استفاده می‌کند. حال اگر تغییر در جزئیات پیاده سازی یک کلاس (برای بهبود عملکرد آن) به وجود بیاید، کاربر کلاس نیازی به تغییر برنامهٔ خود ندارد و با استفاده از همان واسط‌های قبلی از کلاس مورد نیاز خود استفاده می‌کند. برای مثال فرض کنید در پیاده سازی یک پشته به جای آرایه از لیست پیوندی استفاده شود. کاربر نیازی به اطلاع از آن ندارد و می‌تواند به صورتی که قبلا از کلاس استفاده می‌کرد، از کلاس استفاده کند.

انتزاع داده

- در مواقعی که یک کاربر نیاز به تغییر ویژگی‌های یک شیء دارد، معمولاً در کلاس مربوطه توابعی به نام دریافت کننده [1] و تنظیم کننده [2] برای دریافت و تنظیم ویژگی‌های یک شیء از یک کلاس تعریف می‌شوند.
- دریافت کننده‌ها و تنظیم کننده چندین مزیت به همراه دارند. اول آنکه دسترسی به ویژگی‌ها غیر مستقیم می‌شود و کاربر با دستکاری مکان حافظه نمی‌توانند مقدار ویژگی‌های یک شیء را تغییر دهند. دوم آنکه ممکن است تنظیم کننده‌ها نیاز داشته باشند محدودیت‌هایی را برای تنظیم مقادیر اعمال کنند، برای مثال تنظیم‌کننده‌ها می‌توانند محدودهٔ مقدار را بررسی کنند. سوم آنکه ممکن است پیاده سازی ویژگی‌ها تغییر کند اما همچنان تنظیم کننده‌ها و دریافت کننده‌ها به عنوان واسط کاربر یکسان باقی خواهند ماند.

[1] getter
[2] setter

انتزاع داده

- برای ایجاد چنین قابلیت‌هایی در زبان شیءگرا برای ویژگی‌ها و رفتارها سطح دسترسی تعریف می‌شود. تنها ویژگی‌ها و رفتارها با سطح دسترسی عمومی [1] توسط کاربر قابل استفاده هستند و ویژگی‌ها و رفتارها با دسترسی خصوصی [2] از کاربران پنهان می‌شوند.
- همچنین در تعریف یک کلاس می‌توان تعیین کرد در هنگام ساخته شدن یک شیء یا تخریب یک شیء یا انتساب یک شیء به شیء دیگر چه اتفاقی بیافتد. در زبان سی++ می‌توان عملگرها را برای کلاس‌ها سربارگذاری کرد و بدین ترتیب با اعمال عملگرها بر روی اشیاء عملیات مورد نظر اعمال خواهد شد.

[1] public
[2] private

انتزاع داده

- مفهوم انتزاع داده برای اولین بار در زبان سیمولا به وجود آمد.
- زبان سی++ در سال ۱۹۸۵ برای افزودن شیءگرایی به زبان سی به وجود آمد. نوع‌های داده‌ای انتزاعی در سی++ کلاس نامیده می‌شوند. یک کلاس حاوی تعدادی ویژگی است که اعضای داده‌ای [1] نامیده می‌شوند و تعدادی رفتار به نام توابع عضو [2] دارد.
- اعضای یک کلاس مقدار نمی‌گیرند بلکه این اعضای اشیاء کلاس‌ها هستند که دارای مقدار هستند. یک نمونه از یک کلاس شیء نامیده می‌شود.
- اشیاء همچون متغیرهای دیگر می‌توانند ثابت یا غیر ثابت، ایستا، پویا بر روی پشته یا پویا بر روی هیپ باشند.

[1] data members
[2] member function

انتزاع داده

- اشیائی که به صورت پویا بر روی هیپ هستند توسط عملگر `new` ساخته و تخصیص می‌شوند و توسط عملگر `delete` تخریب شده و فضای حافظه آنها آزاد می‌شود. همچنین عملگر `new` تابع سازنده کلاس را و عملگر `delete` تابع مخرب کلای را برای شیء ساخته شده فراخوانی می‌کنند.
- اعضای داده‌ای کلاس نیز می‌توانند متغیرهای ثابت یا غیر ثابت، ایستا یا اشاره‌گر باشند.
- هر یک از اعضای کلاس‌ها می‌توانند خصوصی یا عمومی باشند. اعضای خصوصی با کلمهٔ private مشخص می‌شود و دسترسی کاربر شیء را به آن عضو غیر ممکن می‌سازند. اعضای عمومی یک کلاس با کلمه public مشخص می‌شوند و دسترسی کاربران شیء به آن عضو امکان پذیر است. معمولاً ویژگی‌ها با سطح دسترسی خصوصی و توابعی که کاربران نیاز دارند با سطح دسترسی عمومی تعریف می‌شوند. همچنین یک سطح دسترسی حفاظت شده نیز وجود دارد که در ارث بری از آنها استفاده می‌شود. کلاس‌هایی که از یک کلاس به ارث می‌برند به اعضای حفاظت شده دسترسی دارند، ولی کاربران به اعضای حفاظت شده دسترسی ندارند. اعضای حفاظت شده با کلمهٔ protected مشخص می‌شوند.

انتزاع داده

- در سی++ همچنین می‌توان یک کلاس را از یک نوع پارامتری تعریف کرد. بدین معنی که یک کلاس می‌تواند یک پارامتر به عنوان نوع برای نوع داده‌ای ویژگی‌های خود دریافت کند.
- برای مثال یک پشته که بتواند انواع داده‌ای متنوع را ذخیره کند به صورت زیر تعریف می‌شود.

```
۱ #include <iostream.h>
۲ template <typename Type> // Type is the template parameter
۳ class Stack {
۴   private:
۵     Type *stackPtr;
۶     int maxLen;
۷     int topSub;
```

```
 8   public:
 9     // A constructor for 100 element stacks
10     Stack() {
11         stackPtr = new Type [100];
12         maxLen = 99;
13         topSub = -1;
14     }
15     // A constructor for a given number of elements
16     Stack(int size) {
17         stackPtr = new Type [size];
18         maxLen = size - 1;
19         topSub = -1;
20     }
21     ~Stack() {
22         delete stackPtr;
23     } // A destructor
```

انتزاع داده

```
۲۴      void push(Type number) {
۲۵          if (topSub == maxLen)
۲۶              cout << "Error in push - stack is full\n";
۲۷          else stackPtr[++topSub] = number;
۲۸      }
۲۹      void pop() {
۳۰          if (empty())
۳۱              cout << "Error in pop - stack is empty\n";
۳۲          else topSub--;
۳۳      }
```

انتزاع داده

```
۳۴      Type top() {
۳۵          if (empty())
۳۶              cerr << "Error in top - stack is empty\n";
۳۷          else
۳۸              return (stackPtr[topSub]);
۳۹      }
۴۰      int empty() {
۴۱          return (topSub == -1);
۴۲      }
۴۳  }
```

انتزاع داده

- سپس برای تعریف یک پشته حاوی مقادیر صحیح از پارامتر int در تعریف شیء استفاده می‌کنیم :

```
۱  Stack <int> istack;
۲  Stack <float> fstack;
۳  Stack <Student> stustack;
۴
۵  istack.push(1);
۶  istack.push(2);
۷  while (!istack.empty()) {
۸      int i = istack.pop();
۹      cout << i << endl;
۱۰ }
۱۱ istack.maxLen++; // error
```

انتزاع داده

- در زبان پایتون اولین پارامتر توابع یک کلاس متغیر self است که مشخص می‌کند تابع متعلق به کلاس مربوطه است و می‌تواند به توابع و اعضای داده‌ای دیگر از طریق self دسترسی پیدا کند.
- تابع سازنده نیز با نام __init__ تعریف می‌شود.
- برای مثال :

```
۱ class Bag :
۲       def  __init__ (self) :
۳            self.data = [ ]
۴       def add (self,x) :
۵            self.data.append (x)
```

- اعضای داده‌ای که با (__) شروع می‌شوند اعضای خصوصی هستند.
- برای پیاده سازی مخرب یک کلاس باید از تابع __del__ استفاده کرد.

وراثت

- مفهوم شیءگرایی اولین بار در زبان سیمولا به وجود آمد، اما برای اولین بار در سال ۱۹۸۰ به طور کامل در زبان اسمالتاک پیاده شد. اسمالتاک یک زبان شیءگرای خالص است بدین معنی که همهٔ متغیرها در واقع شیء هستند. در یک زبان شیءگرا علاوه بر انتزاع داده، دو مفهوم اساسی دیگر به نام وراثت و چند ریختی نیز وجود دارند که در اینجا به مطالعهٔ آن می‌پردازیم.

- یکی از معیارهای مهم سنجش یک زبان، قابلیت آن زبان برای تسهیل استفاده مجدد کدها و نرم‌افزارها [1] است.

- انتزاع داده و تعریف کلاس‌ها امکان استفاده مجدد را فراهم می‌سازد، اما در بسیاری مواقع یک کلاس برای استفاده مجدد نیاز به تغییراتی دارد. علاوه بر این در بسیاری مواقع در یک برنامه نیاز به تعریف دو کلاس متفاوت است که با وجود تفاوت ماهیتی، اشتراکات زیادی نیز با یکدیگر دارند و برای صرفه جویی در کد و همچنین زمان برنامه نویسی بهتر است این جنبه‌های مشترک تنها یک بار پیاده سازی شوند.

[1] software reuse

وراثت

- وراثت [1] در برنامه نویسی شیءگرا راه حلی برای مشکلات مطرح شده است. وقتی یک نوع دادهٔ جدید شبیه یک نوع داده قبلی با اندکی تفاوت است، نوع جدید می‌تواند تمام ویژگی‌ها و رفتارهای نوع قبلی را به ارث ببرد. همچنین وقتی دو کلاس با یکدیگر اشتراکات زیادی دارند می‌توان حدس زد که این ویژگی‌ها و رفتارهای مشترک می‌توانند یک موجودیت جدید را در مدل نرم‌افزار تشکیل دهند به طوری که موجودیت جدید پدر و کلاس‌های موجود فرزند آن کلاس هستند. به طور مثال در یک سیستم نرم‌افزاری ممکن است دو کلاس به نام‌های وکتور و صف برای دو ساختار داده داشته باشیم که این دو ساختار در عین تفاوت، اشتراکات زیادی دارند. وجود این اشتراکات نشان دهندهٔ این است که احتمالاً یک ساختار انتزاعی دیگر وجود دارد. می‌توانیم این ساختار انتزاعی را ظرف [2] بنامیم و برای آن یک کلاس تعریف کنیم. سپس دو کلاس وکتور و صف می‌توانند از کلاس ظرف به ارث ببرند. علاوه بر این که ویژگی‌ها و رفتارهای مشترک تنها یک بار تعریف می‌شوند، برنامه نیز ساختار منسجم‌تری خواهد یافت.

[1] inheritance
[2] container

وراثت

- یک کلاس که از کلاس دیگر به ارث می‌برد زیر کلاس [1] یا کلاس فرزند [2] یا کلاس مشتق شده [3] نامیده می‌شود و کلاسی که از آن به ارث برده شده است، کلاس پایه [4] ، کلاس مافوق [5] ، یا کلاس پدر [6] نامیده می‌شود.
- در برنامه نویسی شیءگرا توابع کلاس گاهی متود [7] نیز نامیده می‌شوند.

[1] subclass
[2] child class
[3] derived class
[4] base class
[5] super class
[6] parent class
[7] method

وراثت

- با استفاده از وراثت کلاس فرزند هم می‌تواند ویژگی‌ها و رفتارهایی را اضافه کند و بدین ترتیب علاوه بر ویژگی‌ها و رفتارهای به ارث برده از پدر، تعدادی را نیز خود تعریف می‌کند و هم اینکه می‌تواند تعدادی از رفتارهای پدر را به گونه‌ای دیگر بازتعریف (دوباره تعریف) کند. همچنین کلاس پدر می‌تواند ویژگی‌ها و رفتارهایی را از کلاس‌های فرزند پنهان کند.
- تابعی که بازتعریف یک تابع پدر است، در واقع تابع پدر را لغو [1] می‌کند.
- برای مثال فرض کنید چندین پرنده از کلاس پرنده به ارث می‌برند. برای کلاس پرنده تابعی به نام `draw` برای رسم پرنده در کلاس پدر تعریف شده است که شمایل پرنده را به صورت کلی بدون جزئیات رسم می‌کند. با وجود این که هر پرنده‌ای ویژگی‌های کلاس پرنده را به ارث می‌برد، اما هر کلاس فرزند پرنده تابع رسم را به گونه‌ای متفاوت پیاده سازی می‌کند. پس تابع `draw` در کلاس پدر توسط کلاس‌های فرزند لغو می‌شود.

[1] override

وراثت

- سطح دسترسی حفاظت شده [1] در وراثت استفاده می‌شود. وقتی یک عضو با این سطح دسترسی تعریف شود، کلاس‌های فرزند به آن عضو دسترسی دارند، اما کاربران کلاس به آن عضو دسترسی ندارند.
- ویژگی‌ها و رفتارها معمولاً متعلق به اشیا یا نمونه‌های کلاس هستند، اما گاهی ممکن است نیاز به ویژگی یا رفتاری داشته باشیم که متعلق به کلاس داشته باشد. برای مثال وقتی می‌خواهیم تعداد اشیاء ساخته شده از یک کلاس را بشماریم، متغیر شمارند باید متعلق به کلاس باشد نه اشیاء. اگر ویژگی‌ها و رفتارهای یک کلاس به صورت ایستا تعریف شوند، آن ویژگی‌ها متعلق به کلاس هستند و با تغییر آنها ویژگی همه اشیای کلاس تغییر می‌کند.
- یک کلاس می‌تواند از چندین کلاس به ارث ببرد که به آن وراثت چندگانه [1] گفته می‌شود.

[1] protected
[1] multiple inheritance

چندریختی

- یکی دیگر از قابلیت‌ها در برنامه نویسی شیءگرا چندریختی [1] است که انقیاد پویای توابع [2] نیز نامیده می‌شود. چندریختی در واقع به معنای قابلیت استفاده از یک نام واحد برای توابع متفاوت است.
- توضیح خواهیم داد که چرا به چندریختی انقیاد پویای توابع کلاس نیز گفته می‌شود.
- فرض کنید کلاسی به نام `shape` داشته باشیم که نمایندهٔ همه اشکال هندسی است. حال کلاس‌های متفاوتی از جمله کلاس دایره و مستطیل از این کلاس به ارث می‌برند. همه این کلاس‌ها تابعی به نام `draw` برای رسم شکل دارند که این تابع را از کلاس شکل به ارث می‌برند.

[1] polymorphism
[2] dynamic function binding

چندریختی

- حال فرض کنید می‌خواهیم لیستی از تعداد اشکال متفاوت تشکیل دهیم که همهٔ آنها توسط اشاره‌گری از نوع شکل مشخص شده‌اند. اگر تابع رسم را برای همه اعضا این لیست فراخوانی کنیم در واقع نیاز داریم تابع draw را از کلاس‌های متفاوت فراخوانی کنیم. پس در زمان کامپایل مشخص نیست چه تابعی فراخوانی می‌شود، اما در زمان اجرا با توجه به این که یک عضو لیست به چه شکلی اشاره می‌کند باید تابع رسم برای شکل مربوطه فراخوانی شود. بنابراین زمان انقیاد تابع چند ریخت در زمان اجرا است یعنی به طور پویا صورت می‌گیرد و به همین دلیل چند ریختی را انقیاد پویای توابع کلاس نیز می‌گویند.

- وقتی یک کلاس پدر یک رفتار را تعریف می‌کند اما آن را پیاده سازی نمی‌کند به آن رفتار یک رفتار انتزاعی [1] گفته می‌شود. در سی++ به این رفتارها، توابع مجازی خالص [2] گفته می‌شود. کلاسی که حداقل یک رفتار انتزاعی داشته باشد، کلاس انتزاعی گفته می‌شود. از یک کلاس انتزاعی نمی‌توان نمونه ساخت. کلاسی که از یک کلاس انتزاعی به ارث می‌برد باید رفتارهای انتزاعی آن کلاس را پیاده سازی کند، در غیر اینصورت انتزاعی باقی می‌ماند.

[1] abstract
[2] pure virtual function

پیاده سازی زبان شیءگرا

- در زبان سی++، برای تخصیص حافظه به اشیاء در زمان اجرا، برای هر شی یک رکورد نمونهٔ کلاس [1] در نظر گرفته می‌شود که اندازهٔ آن به اندازهٔ مجموع اندازهٔ اعضای داده‌ای کلاس است و دسترسی به اعضای داده‌ای توسط یک آفست یا فاصله از ابتدای رکورد است.
- وقتی یک کلاس از یک کلاس دیگر به ارث می‌برد، در واقع یک کپی از رکورد نمونهٔ کلاس پدر گرفته می‌شود و اندازه‌ای برای اعضای داده‌ای کلاس فرزند به آن اضافه می‌شود.

[1] class instance record

پیاده سازی ساختار شیءگرا

- توابعی که چند ریخت نیستند با رکورد نمونه در ارتباط نیستند و تنها آدرس آن تابع برای دسترسی به آنها کافی است.
- توابع چندریخت اما با رکورد نمونه در ارتباط اند. در رکورد نمونه باید اشارهگری وجود داشته باشد که به تابع چند ریخت مورد نیاز اشاره کند.
- در رکورد نمونه ساختاری به نام جدول تابع مجازی [1] یا vtable وجود دارد که به ازای هر تابع مجازی در یک رکورد نمونه، آدرس تابع مشخص میکند.

[1] virtuel function table (vtable)

پیاده سازی زبان شیءگرا

- انتخاب توابع چندریخت به طور پویا در زمان اجرا صورت می‌گیرد. به عبارت دیگر تنها در زمان اجرا [1] مشخص می‌شود که یک اشاره‌گر به چه شیئی از اشیای چندریخت اشاره می‌کند.

- به طور مثال برنامه‌ای را در نظر بگیرید که در آن کاربر می‌تواند اشکالی را رسم کرده و از بین اشکال رسم شده، یک شکل را انتخاب می‌کند. حال بسته به این که چه شکلی توسط کاربر انتخاب شده است، مساحت توسط یک تابع چندریخت باید محاسبه شود. پس در جایی از برنامه داریم:

```
۱ shape * shp;
۲ if (/*user chooses circle*/) shp = new circle;
۳ else if (/*user chooses rectangle*/) shp = new rectangle;
۴ double area = shp->calcArea();
```

- در زمان کامپایل [2] مشخص نیست کاربر چه شکلی را انتخاب خواهد کرد و shp به چه شیئی اشاره می‌کند، پس کد ماشین تولید شده نمی‌تواند آدرس تابع calcArea مورد نظر را تعیین کند.

[1] run-time
[2] compile-time

پیاده سازی زبان شیءگرا

- به دلیل این که آدرس تابع مورد نظر برای اجرا در توابع چندریخت به طور پویا در زمان اجرا تعیین می‌شود، چندریختی از سازوکاری به نام انقیاد پویای توابع [1] استفاده می‌کند.
- در مقابل سازوکار انقیاد پویا، انقیاد ایستای توابع [2] وجود دارد. در سربارگذاری توابع از انقیاد ایستا استفاده می‌کنیم.
- به عبارت دیگر، در سربارگذاری توابع، در زمان کامپایل، کامپایلر همهٔ اطلاعات مورد نیاز برای قرار دادن آدرس توابع سربارگذاری شده در کد ماشین را دارد.

[1] dynamic binding
[2] static binding

پیاده سازی زبان شیءگرا

- فرض کنید در یک برنامه، بسته به این که کاربر می‌خواهد با استفاده از نام دانشجو یا شمارهٔ دانشجویی، اطلاعات دانشجو را بیابد، تابع سربارگذاری شدهٔ `getinfo` فراخوانی می‌شود.

```
۱ if (/*user chooses selection by name*/) {
۲     cin >> name; getinfo(name);
۳ } else if (/*user chooses selection by id*/) {
۴     cin >> id; getinfo(id);
۵ }
```

- در زمان کامپایل، کامپایلر می‌تواند دقیقا کد ماشین معادل کد بالا را تولید کرده و آدرس توابع سربارگذاری شده را جایگزین نام توابع کند. پس در زمان اجرا هیچ تصمیم‌گیری صورت نمی‌گیرد.

پیاده سازی زبان شیءگرا

- از آنجایی که انتخاب تابع چند ریخت در زمان اجرا صورت می‌گیرد، چندریختی با استفاده از یک جدول توابع مجازی به نام vtable و یک اشاره‌گر به جدول توابع مجازی به نام vptr پیاده‌سازی می‌شود.
- نحوهٔ پیاده‌سازی چندریختی توسط کامپایلر بدین صورت است که کامپایلر به هر کلاس چندریخت که توابع مجازی را تعریف می‌کند، یک اشاره‌گر vptr اضافه می‌کند که این اشاره‌گر به به یک جدول vtable اشاره می‌کند. در این جدول آدرس توابع مجازی که پیاده‌سازی شده‌اند قرار می‌گیرد.
- حال هر تابعی که از یک کلاس چندریخت ارث‌بری کند، طبق قوانین وراثت اشاره‌گر vptr را نیز به ارث می‌برد. اشاره‌گر در کلاس فرزند به جدولی اشاره می‌کند که در آن جدول آدرس توابع کلاس پیاده‌سازی شده در کلاس فرزند ذکر شده است. اگر تابعی چندریخت در کلاس فرزند تعریف نشده باشد، برای آن تابع آدرس تابعی قرار می‌گیرد که نزدیک‌ترین پدر آن را پیاده‌سازی کرده باشد.
- حال در زمان اجرا با استفاده از vptr کامپایلر می‌تواند تصمیم بگیرد چه توابعی را اجرا کند.

پیاده سازی زبان شیءگرا

- برای مثال فرض کنید کلاس A توابع چندریخت f و g را تعریف کرده است. پس کلاس A یک اشارهگر مجازی vptr دارد که به جدول توابع مجازی vtable از کلاس A اشاره میکند. در این جدول آدرس پیادهسازی توابع f و g ذکر شده است.
- حال اگر کلاس B از کلاس A به ارث ببرد، اشارهگر را نیز به ارث میبرد و اشارهگر vptr در کلاس B به جدولی مجازی مربوط به کلاس B اشاره میکند. حال اگر B هیچ کدام از توابع f و g را تعریف نکند، در جدول vtable کلاس B آدرس توابع f و g در کلاس پدر ذکر میشود. اما اگر B هر یک از این توابع را پیادهسازی کند، در جدول توابع مجازی آن، آدرس توابع پیادهسازی شده توسط خود کلاس B ذکر میشود.

پیاده سازی زبان شیءگرا

- حال برنامه زیر را در نظر بگیرید.

```
A aobj; B bobj; A * aptr;
if (/*user chooses A*/) aptr = &aobj;
else if (/*user chooses B*/) aptr = &bobj;
aptr->f(); aptr->g();
```

- فرض کنید کلاس B تنها تابع f را پیاده‌سازی کند. کامپایلر این برنامه را به شکل زیر کامپایل خواهد کرد.

```
A aobj; B bobj; A * aptr;
// aobj.vptr and bobj.vptr are added.
// aobj.vtable contains addresses of f and g implemented in A.
// bobj.vtable contains the address of f implemented in B
// and the address of g implemented in A.
if (/*user chooses A*/) aptr = &aobj;
else if (/*user chooses B*/) aptr = &bobj;
aptr->f(); // => aptr->vptr->vtable[f]();
aptr->g(); //=> aptr->vptr->vtable[g]();
```

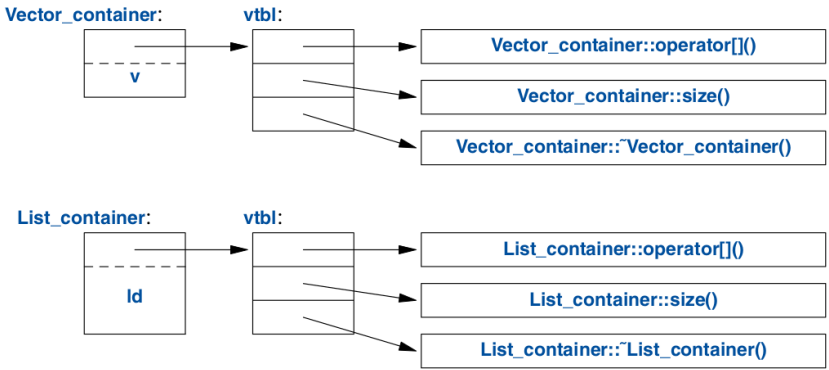
پیاده سازی زبان شیءگرا

```
A aobj; B bobj; A * aptr;
// aobj.vptr and bobj.vptr are added.
// aobj.vtable contains addresses of f and g implemented in A.
// bobj.vtable contains the address of f implemented in B
// and the address of g implemented in A.
if (/*user chooses A*/) aptr = &aobj;
else if (/*user chooses B*/) aptr = &bobj;
aptr->f(); // => aptr->vptr->vtable[f]();
aptr->g(); //=> aptr->vptr->vtable[g]();
```

- پس در زمان اجرا بسته به اینکه vptr به چه جدولی اشاره کند و در جدول مربوطه چه آدرسی ذکر شده است، تصمیم‌گیری مبنی بر اجرای تابع چندریخت مورد نظر صورت می‌گیرد.

پیاده سازی زبان شیءگرا

- پس اشیای چندریخت که از کلاس‌های انتزاعی به ارث برده‌اند، جدولی به نام جدول تابع مجازی [1] یا vtable در حافظه نگهداری می‌کنند که در آن جدول آدرس توابعی که باید فراخوانی شوند، یادداشت شده است.



[1] virtual function table

پیاده سازی زبان شیءگرا

- در دیباگر gdb می‌توان جدول vtable برای اشاره‌گر ptr را با دستور `-exec info vtbl ptr` مشاهده کرد.

```
-exec info vtbl aptr
vtable for 'A' @ 0x55555556b8f8 (subobject @ 0x7fffffffd0f0):
[0]: 0x55555556370c <A::f()>
[1]: 0x55555556371c <A::g()>
-exec info vtbl aobj
vtable for 'A' @ 0x55555556b8f8 (subobject @ 0x7fffffffd0f0):
[0]: 0x55555556370c <A::f()>
[1]: 0x55555556371c <A::g()>
-exec info vtbl bobj
vtable for 'B' @ 0x55555556b8d8 (subobject @ 0x7fffffffd100):
[0]: 0x55555556372c <B::f()>
[1]: 0x55555556371c <A::g()>
```

مسائل طراحی زبان شیءگرا

- تعدادی از مسائل در زمان طراحی یک زبان شیءگرا باید مورد بررسی قرار بگیرند.
- در طراحی یک زبان شیءگرا می‌توان همهٔ نوع‌های داده‌ای را به صورت کلاس تعریف کرد. بدین ترتیب از اعداد ساده گرفته تا ساختارهای پیچیده همگی کلاس هستند. این طراحی گرچه یکنواخت و یکپارچه است اما یک نقص نیز دارد. وقتی نوع‌های اساسی مانند عدد صحیح و اعشاری به صورت کلاس طراحی شوند تمام سربارهایی که تعریف یک کلاس دارد برای این نوع‌ها نیز وجود خواهد داشت. این سربار باعث کاهش راندمان و افزایش زمان اجرا می‌شود.
- راه حلی که برای این مسئله در نظر گرفته شده است این است که در بیشتر زبان‌های شیءگرا نوع‌های اساسی و ابتدایی [1] بدون شیءگرایی همانند برنامه نویسی رویه‌ای طراحی شده‌اند و نوع‌های دیگر به صورت کلاس.

[1] primitive type

مسائل طراحی زبان شیءگرا

- یکی دیگر از مسائل در طراحی زبان‌های شیءگرا این است که آیا یک زبان باید وراثت چندگانه را پشتیبانی کند یا خیر.
- برای روشن شدن این مسئله مثال زیر را در نظر بگیرید. فرض کنید کلاس A و B هر دو تابع f را پیاده سازی کنند. حال کلاس C از کلاس A و B به صورت وراثت چندگانه به ارث می‌برد. حال فرض کنید تابع f از کلاس C فراخوانی شود، در این حالت فراخوانی مبهم است، زیرا مشخص نیست کامپایلر باید تابع f از کلاس A را فراخوانی کند یا تابع f از کلاس B .
- زبان سی++ وراثت چندگانه را پشتیبانی می‌کند و در چنین مواقعی پیام خطا با محتوای ابهام در فراخوانی صادر می‌کند ولی زبان جاوا وراثت چندگانه را پشتیبانی نمی‌کند.

مسائل طراحی زبان شیءگرا

- زبان جاوا وراثت چندگانه را در کلاس‌های رابط [1] پشتیبانی می‌کند. یک کلاس رابط در واقع کلاسی است که در آن پیاده سازی وجود ندارد، بنابراین وراثت چندگانه ابهامی ایجاد نمی‌کند، زیرا هیچ پیاده سازی وجود ندارد که ابهام در فراخوانی به وجود آورد.

[1] interface

مسائل طراحی زبان شیءگرا

- وراثت چندگانه در زبان سی++ یک مشکل دیگر نیز ایجاد می‌کند. فرض کنید کلاس A متغیر x را تعریف می‌کند. سپس کلاس‌های B و C هر دو از A به ارث می‌برند. کلاس D از کلاس B و C به صورت وراثت چندگانه به ارث می‌برد. حال اگر D بخواهد به متغیر x دسترسی پیدا کند مشخص نیست آیا به متغیر x که از طریق A به ارث رسیده است دسترسی پیدا می‌کند یا به متغیر x که از B به ارث رسیده است. در این حالت نیز ابهام به وجود می‌آید زیرا هر کدام از کلاس‌های B و C ممکن است مقداری متفاوت برای متغیر x تعیین کرده باشند. این مشکل، مشکل لوزی [1] نامیده می‌شود، زیرا A و B و C و D یک لوزی می‌سازند. برای حل این مشکل لوزی، زبان سی++ وراثت مجازی را تعریف کرده است. وقتی B و C به صورت مجازی از A به ارث می‌برند، هر دو برای متغیر x یک خانه حافظه تخصیص می‌دهند، بنابراین در دسترسی D به متغیر x ابهامی به وجود نمی‌آید.
- طراحان زبان جاوا تصمیم گرفته‌اند وراثت مجازی را ممنوع کنند، زیرا عقیده داشته‌اند مشکلاتی که وراثت چندگانه به وجود می‌آورد بیشتر است از مشکلاتی که می‌تواند در طراحی حل کند.

[1] diamond problem

مسائل طراحی زبان شیءگرا

- در زبان پایتون نیز وراثت چندگانه پشتیبانی می‌شود ولی ابهامی به وجود نمی‌آورد چرا که اگر دو پدر یک تابع یکسان پیاده سازی کرده باشند، تابع فرزند در فراخوانی تابع مذکور اولویت را به پدر اول میدهد.

```
۱  class B :
۲      def f() :
۳          print ("B")
۴  class C :
۵      def f() :
۶          print ("C")
۷  class D(B,C) :
۸      pass
۹  d = D()
۱۰ d.f() # B
```

مسائل طراحی زبان شیءگرا

- مسئلهٔ دیگری که در طراحی شیءگرا وجود دارد، مسئلهٔ تخصیص حافظه و آزادسازی حافظه است. هر کلاس ممکن است تعداد زیادی ویژگی داشته باشد که هر کدام از این ویژگی‌ها ممکن است آرایه یا اشیایی از کلاس‌های دیگر باشد و بنابراین ممکن است تعداد زیادی از ویژگی‌ها بر روی هیپ ساخته شوند.
- وقتی یک شیء تخریب می‌شود، برنامه نویس باید مراقب باشد که همهٔ حافظه‌های تخصیص داده شده در هیپ را آزاد کند در غیر اینصورت نشست حافظه رخ می‌دهد. طراح زبان دو راه پیش رو دارد. اگر آزادسازی حافظه را به برنامه نویس محول کند قابلیت اطمینان برنامه پایین می‌آید ولی در عوض برنامه می‌تواند با سرعت بالاتری اجرا شود. اگر آزادسازی حافظه به طور خودکار انجام شود، قابلیت اطمینان برنامه بالا می‌رود ولی سربار اضافی تحمیل شده بابت آزادسازی حافظه به صورت خودکار سرعت اجرای برنامه را پایین می‌آورد.
- زبان سی++ روش اول و جاوا روش دوم را در پیش گرفته است. پایتون نیز دارای مکانیزم بازیافت حافظه خودکار است. زبان راست راه حل سومی در پیش گرفته است. برنامه‌نویس بر روی حافظه کنترل دارد ولی باید از قوانینی که زبان تعیین کرده است پیروی کند و در صورتی که آزادسازی حافظه به درستی صورت نگرفته باشد، کامپایلر پیام خطا صادر می‌کند.

جاوا

- در اینجا به پاره‌ای از تفاوت‌ها بین سی++ و جاوا اشاره می‌کنیم.
- کلاس‌ها در سی++ می‌توانند بدون کلاس پدر باشند، اما در جاوا همه کلاس باید کلاس مافوق داشته باشند.
- کلاس `Object` در بالاترین رده در سلسله مراتب کلاس‌هاست.
- کلاس `Object` تعدادی متود از جمله `toString` برای تبدیل شیء به رشته و `equals` برای مقایسه برابری دو شیء دارد.
- همهٔ شیءها در جاوا بر روی هیپ ساخته می‌شوند برخلاف سی++ که اشیاء می‌توانند بر روی پشته یا بر روی هیپ ساخته شوند.

- عملگر delete برای آزادسازی حافظه در جاوا وجود ندارد چرا که بازیافت حافظه به صورت خودکار انجام می‌شود. بازیافت کنندهٔ حافظه تنها بر روی حافظه کنترل دارد و بر روی منابع دیگر هیچ کنترلی ندارد بنابراین اگر یک شیء در حال خواندن یک فایل باشد و شیء مورد نظر توسط بازیافت کنندهٔ حافظه از بین برود، بازیافت کننده هیچ عملیاتی بر روی فایل انجام نمی‌دهد. برای اینکه برنامه نویس بتواند در هنگام تخریب شیء عملیاتی را پیش‌بینی کند، متود `finalize` می‌تواند پیاده سازی شود. این متود در هنگام تخریب شیء فراخوانی می‌شود، اما مشخص نیست دقیقا چه زمانی یک شیء تخریب می‌شود.
- در جاوا با استفاده از کلمه `final` می‌توان کلاسی تعریف کرد که هیچ کلاسی نتواند از آن ارث ببرد. چنین کلاسی که نمی‌تواند فرزند داشته باشد یک کلاس نهایی نامیده می‌شود.

- یک متود چند ریخت که متود پدر را لغو می‌کند می‌تواند با کلمه `@override` نشانه‌گذاری شود. در صورتی که متود مورد نظر در کلاس پدر پیاده سازی شده باشد، متود پدر لغو می‌شود و با متود فرزند جایگزین می‌شود و البته کلمه `@override` تأثیری در این فرایند ندارد، اما اگر متودی با کلمهٔ `@override` نشانه گذاری شود و کلاس پدر آن متود را پیاده سازی نکرده باشد، آنگاه کامپایلر پیام خطا صادر می‌کند.
- برای ارسال آرگومان از کلاس فرزند به کلاس پدر از تابع `super` استفاده می‌شود.

جاوا

- اگر یکی از سازنده‌های کلاس پدر به طور صریح با کلمه `super` فراخوانی نشوند، به طور ضمنی سازندهٔ پیش‌فرض کلاس پدر یا سازنده با صفر پارامتر فراخوانی می‌شود.
- جاوا برخلاف سی++، وراثت خصوصی را پشتیبانی نمی‌کند.
- همچنین بر خلاف سی++، جاوا وراثت چندگانه را پشتیبانی نمی‌کند، اما می‌توان کلاس‌های واسط یا `interface` تعریف کرد که هیچ پیاده سازی ندارند اما وراثت چندگانه را پشتیبانی می‌کنند. یک کلاس واسط هیچ ویژگی ندارد و همچنین هیچ سازنده‌ای برای آن تعریف نمی‌شود. در یک کلاس واسط تنها متودها اعلام می‌شوند اما تعریف نمی‌شوند.
- یک کلاس واسط می‌تواند از چند کلاس واسط به ارث ببرد و همچنین یک کلاس می‌تواند چندین کلاس واسط را پیاده سازی کند، اما یک کلاس تنها از یک کلاس به ارث می‌برد.
- وقتی یک کلاس، یک کلاس واسط را پیاده سازی می‌کند، باید همهٔ متودهای آن را تعریف کند.

- قابلیت چند ریختی توسط کلاس‌های واسط می‌تواند مورد استفاده قرار بگیرد. برای مثال یک متود می‌تواند به عنوان پارامتر یک کلاس واسط تعریف کند. سپس به عنوان آرگومان به این کلاس باید کلاسی ارسال شود که کلاس واسط مذکور را پیاده سازی کرده باشد.
- یک کلاس در جاوا می‌تواند با استفاده از کلمهٔ abstract به صورت انتزاعی تعریف شود. در یک کلاس انتزاعی حداقل یکی از متودها انتزاعی است یعنی هیچ تعریفی ندارد. از یک کلاس انتزاعی نمی‌توان شیء ساخت. کلاس انتزاعی جاوا معادل کلاس انتزاعی سی++ است که حداقل یکی از توابع آن مجازی خالص است.
- در سی++ تنها توابعی که به صورت مجازی با کلمهٔ `virtual` تعریف شوند چند ریخت هستند اما در جاوا همه متودهایی که به صورت عمومی تعریف شوند می‌توانند چندریخت باشند.

مدیریت استثنا

- یکی از ساختارهای کنترلی در زبان‌های شیءگرا مدیریت استثنا [1] است.
- یک استثنا [1] رویدادی غیر طبیعی است که در زمان اجرا تشخیص داده می‌شود. پردازش ویژه‌ای که برای استثنا مهیا می‌شود، مدیریت استثنا [2] نامیده می‌شود. در هنگام خطا یک استثنا ارسال [3] یا پرتاب [4] می‌شود. در قسمت دیگری از برنامه این استثنا دریافت [5] و عملیات لازم انجام می‌شود.

---

[1] exception handing
[1] exception
[2] exception handing
[3] raise
[4] throw
[5] catch

مدیریت استثنا

- دو مزیت برای ساختار مدیریت استثنا می‌توان برشمرد. اول آنکه بدون مدیریت استثنا برنامه نویس باید از همه استثناها آگاه باشد و آنها را در برنامهٔ خود به طور صریح بیان کند. به طور مثال دسترسی به آرایه را توسط ساختار if بررسی کند. این امر باعث پیچیده شدن برنامه می‌شود. دوم آنکه توسط مدیریت استثنا می‌توان استثنایی را در یک تابع دریافت کرد و در توابع دیگر آن را مدیریت نمود. بدون ساختار مدیریت استثنا چنین کاری نیازمند پیچیدگی برنامه جهت بازگرداندن مقادیر خطا می‌شود.

نوع عمومی : راست

- در زبان راست یک ساختمان را می‌توانیم از نوع دادهٔ عمومی تعریف کنیم و بدین ترتیب نوع اعضای ساختمان را به عنوان پارامتر به ساختمان ارسال کنیم.

```
۱ struct Point<T> {
۲     x: T,
۳     y: T,
۴ }
۵ fn main() {
۶     let integer = Point { x: 5, y: 10 };
۷     let float = Point { x: 1.0, y: 4.0 };
۸ }
```

راست : نوع عمومی

- یک ساختمان عمومی می‌تواند بیش از یک پارامتر نوع نیز دریافت کند.

```
۱ struct Point<T, U> {
۲     x: T,
۳     y: U,
۴ }
۵ fn main() {
۶     let both_integer = Point { x: 5, y: 10 };
۷     let both_float = Point { x: 1.0, y: 4.0 };
۸     let integer_and_float = Point { x: 5, y: 4.0 };
۹ }
```

راست : نوع عمومی

- نوع دادهٔ شمارشی نیز می‌تواند به صورت عمومی تعریف شود.

```
۱ enum Option<T> {
۲     Some(T),
۳     None,
۴ }
۵ enum Result<T, E> {
۶     Ok(T),
۷     Err(E),
۸ }
```

راست : نوع عمومی

- در پیاده سازی متودهای یک ساختمان نیز می‌توانیم از نوع عمومی استفاده کنیم. قرار دادن نوع T بعد از impl به کامپایلر می‌گوید T یک نوع عمومی است نه یک نوع خاص.

```
struct Point<T> {
    x: T,
    y: T,
}
impl<T> Point<T> {
    fn x(&self) -> &T {
        &self.x
    }
}
fn main() {
    let p = Point { x: 5, y: 10 };

    println!("p.x = {}", p.x());
}
```

راست : نوع عمومی

- ممکن است بخواهیم برای یک نوع خاص یک تابع جداگانه تعریف کنیم. در اینصورت می‌توانیم پیاده سازی ساختمان را به صورت زیر انجام دهیم.

```
۱ impl Point<f32> {
۲     fn distance_from_origin(&self) -> f32 {
۳         (self.x.powi(2) + self.y.powi(2)).sqrt()
۴     }
۵ }
```

راست : نوع عمومی

- پارامترهای نوع یک ساختمان می‌توانند با پارامترها متودها تفاوت داشته باشند.

```
struct Point<X1, Y1> {
    x: X1,
    y: Y1,
}
impl<X1, Y1> Point<X1, Y1> {
    fn mixup<X2, Y2>(self, other: Point<X2, Y2>) -> Point<X1, Y2> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}
```

راست : نوع عمومی

```
۱ fn main() {
۲     let p1 = Point { x: 5, y: 10.4 };
۳     let p2 = Point { x: "Hello", y: 'c' };
۴     let p3 = p1.mixup(p2);
۵     println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
۶ }
```

راست : نوع عمومی

- استفاده از نوع‌های عمومی هیچ سرباری بر سرعت اجرای برنامه ندارد. در زمان کامپایل همهٔ کدهای عمومی توسط کامپایلر به نوع‌های اصلی و ساخته شده توسط کاربر تبدیل می‌شوند.

راست : رفتار مشترک

- با استفاده از رابط‌ها [1] می‌توانیم رفتار مشترک برای چند نوع متفاوت تعریف کنیم.
- برای این کار ابتدا باید آن رابط را تعریف کنیم و سپس رابط را برای نوع‌های متفاوت تعریف کنیم.

```
۱ pub trait Summary {
۲     fn summarize(&self) -> String;
۳ }
```

[1] trait

راست : رفتار مشترک

- حال برای ساختمان‌های متفاوت می‌توانیم آن رابط را پیاده سازی کنیم.

```
۱  pub struct NewsArticle {
۲      pub headline: String,
۳      pub location: String,
۴      pub author: String,
۵      pub content: String,
۶  }
۷  impl Summary for NewsArticle {
۸      fn summarize(&self) -> String {
۹       format!("{}, by {} ({})", self.headline, self.author,
۱۰                                              self.location)
۱۱     }
۱۲ }
```

راست : رفتار مشترک

```rust
pub struct Tweet {
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}
impl Summary for Tweet {
    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}
```

راست : رفتار مشترک

- در زبان راست وراثت وجود ندارد. بدین معنی که یک ساختمان نمی‌تواند از یک ساختمان دیگر به ارث ببرد. اما چند ساختمان می‌توانند یک رفتار مشترک داشته باشند.
- همچنین چند ریختی به شکلی که در زبان‌های شیءگرا وجود دارد، در راست وجود ندارد اما ساختمان‌هایی که یک رفتار مشترک داشته باشند می‌توانند به عنوان پارامتر به توابع ارسال شوند.
- راست بدین دلیل از وراثت استفاده نمی‌کند که در وراثت معمولاً کدهایی بین کلاس‌ها که به اشتراک گذاشته می‌شوند بیش از حد نیاز هستند. برای مثال کلاس‌های فرزند به همهٔ توابع پدر و اجداد نیاز ندارند، اما آنها را به ارث می‌برند. شیءگرایی اساساً برای پیاده سازی مفهوم رفتار مشترک تعریف شده است که این رفتارهای مشترک در راست، توسط رابط‌ها پیاده سازی می‌شوند.

راست : رفتار مشترک

- حال فرض کنید می‌خواهیم یک برنامه واسط کاربر بنویسیم. یک صفحه شامل چندین عنصر است که هر کدام از آنها باید در پنجره یا صفحه شامل چندین عنصر است که هر کدام از آنها باید در پنجره یا صفحه برنامه رسم شوند. یک عنصر می‌تواند یک دکمه یا یک جعبهٔ انتخاب یا اشیای دیگر باشد.
- بنابراین باید ابتدا این رفتار مشترک را تعریف کنیم.
- رفتار مشترک عناصر قابلیت رسم کردن آنها است که آن را به عنوان یک رابط به صورت زیر تعریف می‌کنیم.

```
۱ pub trait Draw {
۲     fn draw(&self);
۳ }
```

راست : رفتار مشترک

- یک صفحه یا Screen تشکیل شده است از تعدادی عناصر که هر کدام از آنها باید ساختمانی باشد که رابط رسم با Draw را پیاده سازی کرده باشد.

```
pub struct Screen {
    pub components: Vec<Box<dyn Draw>>,
}
```

- در اینجا Box یک اشارهگر هوشمند تعریف میکند که برای وقتی به کار میرود که اندازه یک نوع در زمان کامپایل نامشخص است. همچنین dyn مشخص میکند که Draw یک رابط است

راست : رفتار مشترک

- حال برای صفحه تابعی پیاده سازی می‌کنیم که همه عناصر درون خود را چاپ کند.

```
۱ impl Screen {
۲     pub fn run(&self) {
۳         for component in self.components.iter() {
۴             component.draw();
۵         }
۶     }
۷ }
```

راست : رفتار مشترک

- حال یک دکمه یا Button را تعریف می‌کنیم که رابط رسم یا Draw را پیاده سازی می‌کند.

```
۱  pub struct Button {
۲      pub width: u32,
۳      pub height: u32,
۴      pub label: String,
۵  }
۶  impl Draw for Button {
۷      fn draw(&self) {
۸          // code to actually draw a button
۹      }
۱۰ }
```

راست : رفتار مشترک

- می‌توانیم یک جعبهٔ انتخاب یا SelectBox و هر شیء دیگری را به همین شکل تعریف کنیم و رابط رسم را برای آنها پیاده سازی کنیم.

```
struct SelectBox {
    width: u32,
    height: u32,
    options: Vec<String>,
}
impl Draw for SelectBox {
    fn draw(&self) {
        // code to actually draw a select box
    }
}
```

راست : رفتار مشترک

- حال برنامه‌ای می‌نویسیم که شامل دو متغیر از نوع‌های دکمه و جعبهٔ انتخاب است و این برنامه دو تابع رسم متفاوت را برای این متغیرها فراخوانی می‌کند.

```
fn main() {
    let screen = Screen {
        components: vec![
            Box::new(SelectBox {
                width: 75,
                height: 10,
                options: vec![
                    String::from("Yes"),
                    String::from("Maybe"),
                    String::from("No"),
                ],
            }),
```

راست : رفتار مشترک

```
            Box::new(Button {
                width: 50,
                height: 10,
                label: String::from("OK"),
            }),
        ],
    };
    screen.run();
}
```

راست : رفتار مشترک

- رفتارهای یک رابط می‌توانند پیاده سازی پیش فرض نیاز داشته باشد. بدین ترتیب یک ساختمان می‌تواند برخی رفتارهای یک رابط را پیاده سازی کند و برای برخی دیگر از پیاده سازی پیش فرض استفاده کند.
- برای مثال :

```
۱ pub trait Summary {
۲     fn summarize(&self) -> String {
۳         String::from("(Read more...)")
۴     }
۵ }
```

راست : رفتار مشترک

- همچنین رفتارهای پیش‌فرض یک رابط می‌توانند دیگر رفتارها را فراخوانی کنند.

```
pub trait Summary {
    fn summarize_author(&self) -> String;
    fn summarize(&self) -> String {
        format!("(Read more from {}...)", self.summarize_author())
    }
}
```

راست : رفتار مشترک

- رابط‌ها را می‌توان به عنوان پارامتر نیز به توابع دیگر ارسال کرد. بدین ترتیب قابلیت چندریختی زبان‌های شیءگرا در زبان راست قابل استفاده است.

```
۱ pub fn notify(item: &impl Summary) {
۲     println!("Breaking news! {}", item.summarize());
۳ }
```

راست : رفتار مشترک

- روش قبلی برای ارسال یک رابط به یک تابع درواقع معادل کد زیر است که یک نوع عمومی به تابع ارسال می‌کند.

```
pub fn notify<T: Summary>(item: &T) {
    println!("Breaking news! {}", item.summarize());
}
```

راست : رفتار مشترک

- ممکن است بخواهیم یک پارامتر چند رابط را پیاده سازی کرده باشد در این صورت می‌توانیم از عملگر + به صورت زیر استفاده کنیم.

```
pub fn notify(item: &(impl Summary + Display)) {
```

راست : رفتار مشترک

- کد قبل در واقع یک میانبر برای کد زیر است.

```
pub fn notify<T: Summary + Display>(item: &T) {
```

راست : رفتار مشترک

- وقتی تعداد زیادی رابط در تعریف توابع داشته باشیم، ممکن است کد پیچیده شود. در چنین مواقعی می‌توانیم از یک روش دیگر برای ساده‌سازی کد استفاده کنیم.
- برای مثال :

```
fn some_function<T: Display + Clone, U:
                    Clone + Debug>(t: &T, u: &U) -> i32 {
// it is equivalent to :
fn some_function<T, U>(t: &T, u: &U) -> i32
where
    T: Display + Clone,
    U: Clone + Debug,
}
```

برنامه نویسی همروند

همروندی

- همروندی یا همزمانی [1] در یک برنامهٔ کامپیوتری به معنای توانایی واحدهای مختلف یک برنامه برای اجرای همزمان یا به عبارت دیگر اجرای موازی است به طوری که نتیجه نهایی برنامه نسبت به اجرای غیر همزمان تفاوتی نداشته باشد. با اجرای یک برنامه به صورت همروند بر روی چند پردازنده، سرعت اجرا بهبود می‌یابد.
- توسط برنامه نویسی همروند [2] که توسط آن قسمت‌هایی از برنامه به صورت همزمان اجرا می‌شوند، زمان پاسخ (تأخیر) [3] و توان عملیاتی [4] برنامه بهبود می‌یابد.

[1] concurrency
[2] concurrent programming
[3] response time (delay)
[4] throughput

همروندی

- به طور مثال ضرب دو ماتریس را در نظر بگیرید. از آنجایی که درایه‌های ماتریس حاصلضرب مستقل از یکدیگر قابل محاسبه هستند، بنابراین هر یک از درایه‌ها در صورتی که تعداد پردازنده‌ها کافی باشد می‌تواند به طور مستقل و موازی با درایه‌های دیگر محاسبه شود. در این صورت ماتریس با تأخیر کمتر محاسبه خواهد شد.
- حال یک سیستم پردازش تصویر را در نظر بگیرید که در آن تصاویر به ترتیب از ورودی خوانده می‌شوند و پس از چند مرحله پردازش در خروجی نمایش داده می‌شوند. پس از این که اولین مرحلهٔ پردازش توسط یک پردازنده انجام شد، در صورتی که تعداد پردازنده‌ها کافی باشد، پردازنده اول می‌تواند تصویر دوم را پردازش کند و تصویر اول برای پردازش به پردازندهٔ دوم برود. بدین ترتیب در یک فاصله زمانی معین تعداد بیشتری تصویر با استفاده از برنامه نویسی همروند می‌تواند پردازش شود. در این حالت می‌گوییم توان عملیاتی سیستم افزایش یافته است.

همروندی

- همروندی می‌تواند در سطوح متفاوتی باشد : در سطح دستورات ماشین، در سطح دستورات زبان برنامه نویسی، در سطح زیر برنامه و یا در سطح برنامه.
- همروندی در سطح دستورات ماشین مربوط به طراحی سخت‌افزار و همروندی در سطح برنامه مربوط به طراحی سیستم عامل است. طراحان سخت افزار سازوکارهایی را برای اجرای موازی دستورات زبان ماشین فراهم می‌کنند و طراحان سیستم‌عامل روش‌هایی را برای اجرای موازی برنامه‌ها و زمانبندی برنامه‌ها پیاده‌سازی می‌کنند. همروندی در سطح دستورات زبان برنامه‌نویسی و زیربرنامه‌ها موضوع بحث زبان‌های برنامه نویسی است.
- در بسیاری از برنامه‌ها همروندی اهمیت زیادی پیدا می‌کند. برای مثال یک سرور وب به طور همزمان داده‌ها را دریافت و ارسال می‌کند و اطلاعات را به کاربر نمایش می‌دهد و به درخواست‌های کاربر واکنش نشان می‌دهد.
- یک برنامه همروند مقیاس پذیر [1] است زیرا اگر تعداد پردازنده‌ها افزایش یابد می‌تواند تعداد بیشتری داده را پردازش کند و با سرعت بیشتری اجرا شود.

[1] scalable

همروندی

- اولین کامپیوترهایی که پردازش موازی را پشتیبانی می‌کردند، در دهۀ ۱۹۵۰ به وجود آمدند که شامل یک پردازنده اصلی برای انجام محاسبات و یک یا چند پردازنده برای انجام عملیات ورودی و خروجی می‌شدند.
- در دهۀ ۱۹۶۰ سیستم عامل‌ها، برنامه‌های مختلف را به طور همزمان بر روی چندین پردازنده اجرا و زمانبندی می‌کردند.
- در اواسط دهۀ ۱۹۶۰ کامپیوترهایی به وجود آمدند که دستورات ماشین را به طور همزمان می‌توانستند اجرا کنند. کامپایلرها بر روی این ماشین‌ها می‌توانستند تعیین کنند چه دستوراتی به طور همزمان اجرا شوند.

همروندی

- در آن زمان معماری سخت‌افزارهای کامپیوتری به دو دسته تقسیم شد : معماری یک عملیات چند داده و معماری چند عملیات چند داده.
- در معماری اول چندین پردازنده به طور همزمان قسمت‌های مختلف داده را پردازش می‌کنند. بنابراین در این معماری که به آن معماری یک عملیات چند داده [1] گفته می‌شود، پردازش موازی در سطح داده [2] صورت می‌گیرد. به عبارت دیگر در این معماری داده به چند قسمت تقسیم شده و یک عملیات واحد بر روی قسمت‌های مختلف داده اعمال می‌شود. پردازنده‌های گرافیکی غالباً از این نوع معماری هستند.

[1] Single Instructions، Multiple Data (SIMD)
[2] data level parallelism

همروندی

- برای مثال فرض کنید می‌خواهیم عناصر یک آرایه را با یکدیگر جمع کنیم. می‌توانیم آرایه را به چند قسمت تقسیم کرده، و عملیات جمع را بر روی قسمت‌های مختلف آرایه اعمال کنیم. در این معماری از موازی‌سازی داده بهره گرفته می‌شود. در بسیاری از کاربردهای پردازش صدا و تصویر از موازی‌سازی داده بهره گرفته می‌شود بدین صورت که تصویر یا صدا به چند قسمت تقسیم شده، سپس پردازنده‌های مختلف یک عملیات واحد را بر روی قسمت‌های مختلف صدا یا تصویر انجام می‌دهند.

همروندی

- در معماری دوم چندین پردازنده به طور همزمان عملیات متفاوت را بر روی چندین داده متفاوت اعمال می‌کنند و به آن معماری چند عملیات، چند داده [1] گفته می‌شود. به عبارت دیگر چندین عملیات در یک برنامه می‌توانند به طور همزمان اجرا شوند و هر یک از عملیات بر روی یک پردازندهٔ متفاوت اجرا می‌شود. در این معماری از موازی‌سازی در سطح عملیات [2] بهره گرفته می‌شود.

[1] Multiple Instruction، Multiple Data (MIMD)
[2] task level parallelism

همروندی

- برای مثال یک سیستم‌عامل یا یک مرورگر وب به طور همزمان عملیات متفاوتی را انجام می‌دهد که این عملیات بر روی پردازنده‌ها توزیع می‌شوند. به عنوان مثال دیگر در یک برنامه پردازش تصویر، یک پردازنده عملیاتی را بر روی یک تصویر انجام داده و پس از اعمال عملیات، تصویر را به پردازندهٔ بعدی برای اعمال عملیات دیگر انتقال می‌دهد و خود پردازش را با تصاویر بعدی ادامه می‌دهد. در این معماری حافظه می تواند مشترک باشد که در آن صورت نیاز به همگام‌سازی [1] یا هماهنگ سازی واحدهای پردازش وجود دارد تا داده‌ها به درستی خوانده و نوشته شوند. حافظه همچنین می‌تواند توزیع شده [2] باشد که در این صورت نیز نیاز که برقراری ارتباط بین پردازنده‌ها وجود دارد. به این معماری معمولاً چندپردازنده‌ای [3] نیز گفته می‌شود.

[1] synchronization
[2] distributed
[3] multiprocessor

همروندی

- از آنجایی که معماری سخت افزار دائماً در حال به‌روز رسانی است کامپیوترهای جدید روش‌های همزمانی متنوعی را در سطح سخت‌افزار پشتیبانی می‌کنند. برای مثال وقتی دستورات جاری در حال اجرا هستند، دستورات آینده کد گشایی و آماده اجرا می‌شوند یا دو مسیر متفاوت برای بارگیری دستورات و داده‌ها در پردازنده‌ها وجود دارد و یا بخش‌های مختلف دستورات محاسباتی ریاضی تا حد امکان به طور موازی اجرا شوند.
- همروندی در سخت‌افزار تا حدودی می‌تواند نیازهای راندمان برنامه را برآورده کند و قسمتی از مسئولیت بهبود زمان اجرا بر عهده نرم‌افزار است.

همروندی

- دو دسته از واحدهای همروند وجود دارند. در دستهٔ اول با فرض بر این که بیشتر از یک پردازنده وجود دارد، چندین واحد از برنامه به طور موازی بر روی پردازنده‌های مختلف اجرا می‌شوند. به این دسته همروندی فیزیکی [1] گفته می‌شود. در دستهٔ دوم یک پردازنده وجود دارد و واحدهای مختلف برنامه به طور همزمان به طور قطعه قطعه شده و در هم آمیخته [2] بر روی پردازنده اجرا می‌شوند. به این دسته همروندی منطقی [3] گفته می‌شود.
- در بیشتر مواقع تعداد واحدهای همروند از تعداد پردازنده‌ها بیشتر است و بنابراین همروندی به صورت فیزیکی و منطقی اتفاق می‌افتد.

[1] physical concurrency
[2] interleaved
[3] logical concurrency

همروندی

- یک ریسمان کنترلی [1] یا ریسهٔ کنترلی به دنباله‌ای از دستورات گفته می‌شود که به طور پیوسته یکی پس از دیگری در یک واحد از برنامه اجرا می‌شوند.
- در همروندی فیزیکی هر پردازنده تنها می‌تواند یک ریسهٔ کنترلی را اجرا کند، اما در همروندی منطقی بیش از یک ریسهٔ کنترلی نیز می‌توانند بر روی یک پردازنده اجرا شوند.
- برنامه‌ای که در آن چند ریسهٔ کنترلی به طور همزمان اجرا می‌شوند، یک برنامه چند ریسه‌ای [2] گفته می‌شود.

[1] thread of control
[2] multithreaded program

همروندی

- همروندی در سطح دستورات معمولا به این صورت است که دستوراتی که می‌توانند به طور موازی اجرا شوند، به صورت خودکار توسط کامپایلر تشخیص داده شده و به صورت موازی بر روی چندین پردازنده انجام شوند. برای مثال دستورات زیر را در نظر بگیرید.

```
۱ e = a + b
۲ f = c + d
۳ m = e * f
```

- دستور سوم وابسته به دستورات اول و دوم است، اما دستورات اول و دوم از یکدیگر مستقل هستند و بنابراین می‌توانند به صورت موازی اجرا شوند. در اینصورت با تحلیل وابستگی دستورات، و اعمال موازی سازی برنامه می‌تواند با سرعت بیشتری اجرا شود.
- همروندی در سطح زیربرنامه معمولاً به این صورت است که هر زیر برنامه به یک ریسه کنترل سپرده می‌شود.

همروندی

- یک واحدکار یا یک وظیفه [1] واحدی است از برنامه که می‌تواند به صورت همروند با واحدهای دیگر اجرا شود.
- توجه کنید که اجرای موازی [2] حالت خاصی از اجرای همروند [3] است. در اجرای همروند چندین واحد کاری به گونه‌ای اجرا می‌شوند که در بازه‌های زمانی متفاوت اجرای آنها همپوشانی داشته باشد. در اجرای موازی چندین واحد کاری در یک بازهٔ معین همزمان اجرا می‌شوند.
- معمولاً یک ریسه یک واحد کار را به عهده می‌گیرد. وقتی ریسه شروع به انجام عملیات می‌کند، برنامه‌ای که ریسه را راه اندازی کرده است نیاز ندارد منتظر اتمام عملیات ریسه بماند و می‌تواند عملیات خود را ادامه دهد یا ریسه‌های دیگر را راه‌اندازی کند.
- واحدهای کاری می‌توانند حافظه را با یکدیگر به اشتراک بگذارند و یا هر کدام حافظه مختص به خود داشته باشند.

[1] task
[2] parallel
[3] concurrent

همروندی

- معمولا به واحدهای کنترلی که واحدهای کاری را بدون به اشتراک گذاری حافظه انجام می‌دهند، پردازه یا پروسه [1] گفته می‌شود و ریسه‌ها [2] مسئول پردازش واحدهای‌کار با به اشتراک گزاری حافظه هستند.
- ریسه‌ها نسبت به پروسه‌ها بهینه‌تر و کارآمدتر هستند، اما از طرفی چون حافظه آنها اشتراکی است نیاز به همگام‌سازی دارند.
- یک ریسه می‌تواند با ریسه‌های دیگر از طریق به اشتراک گذاری متغیرها یا از طریق کانال‌های ارتباطی و مکانیزم ارسال پیام ارتباط برقرار کند.

[1] process
[2] thread

همروندی

- همگام‌سازی [1] سازوکاری (مکانیزمی) است که توسط آن دسترسی به داده‌های مشترک کنترل می‌شود.
- به وضعیتی که در آن دو یا چند ریسه برای به دست آوردن یک منبع مشترک رقابت می‌کنند، یک وضعیت رقابتی [2] گفته می‌شود.
- معمولاً یک برنامهٔ سیستم عامل به نام زمانبند [3] ریسه‌ها و پروسه‌ها را برای اجرا بر روی پردازنده‌های مختلف زمانبندی می‌کند.

[1] synchronization
[2] race condition
[3] scheduler

همروندی

- یک واحدکار می‌تواند در چند حالت مختلف باشد :

۱. جدید [1] : واحدکار به تازگی ساخته شده و هنوز آماده انجام عملیات نیست.

۲. آماده [2] : واحدکار آماده اجرا است، اما در حال اجرا نیست. زمانبند باید این واحدکار را زمانبندی کند تا به حالت اجرا درآید. همهٔ واحدهای کار که آماده به شروع عملیات هستند در یک صف واحدهای کاری آماده [3] قرار می‌گیرند.

۳. در حال اجرا [4] : یک پردازنده به واحد کار داده شده و می‌تواند اجرا شود.

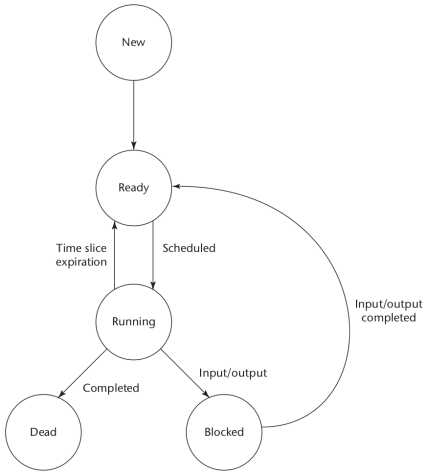
[1] new
[2] ready
[3] task-ready queue
[4] running

- ۴. مسدود [1] : واحد کار مسدود شده است و اجرای آن متوقف شده است. دلیل توقف می‌تواند این باشد که واحد کار نیاز به عملیات ورودی خروجی داشته و یا اینکه زمانبند برای اجرای واحدهای کاری دیگر آن متوقف کرده است.
۵. پایان‌یافته [2] : اجرای واحدکار به اتمام رسیده است. واحد کار یا با موفقیت به کار خود پایان داده است و یا به دلیل خطا یا به طور دستی اجرای آن متوقف شده و تخریب شده است.

[1] blocked
[2] dead

همروندی

- در شکل زیر حالت‌های مختلف یک واحدکار نشان داده شده‌اند.



همروندی

- یک ویژگی برنامه‌های همروند ویژگی زنده بودن [1] است. یک برنامه زنده برنامه‌ای است که اجرای آن تا خاتمه برنامه ادامه پیدا کند و اجرای آن متوقف نشود. یک برنامه ممکن است با بن‌بست [2] مواجه شود بدین معنی که دو یا چند ریسه برای ادامه کار به آزادسازی منابع توسط دیگران نیاز پیدا می‌کنند و بنابراین هیچ‌کدام نمی‌توانند کار خود را ادامه دهند که در نتیجه برنامه به بن‌بست برخورد می‌کند.

[1] liveness
[2] deadlock

همروندی

- دو نوع همگام‌سازی برای داده‌های مشترک وجود دارد : همگام‌سازی مشارکتی [1] و همگام‌سازی رقابتی [2].
- در همگام‌سازی مشارکتی دو واحدکاری به طور همزمان کاری را انجام می‌دهند و یکی از واحدها در نقطه‌ای متوقف می‌شود تا واحد دیگر کار خود را به اتمام برساند و با ارسال پیام به واحد کار اول اطلاع دهد که می‌تواند عملیات خود را ادامه دهد.
- در همگام‌سازی رقابتی چند واحدکاری نیاز به یک منبع مشترک دارند و آن منبع مشترک نمی‌تواند به طور همزمان مورد استفاده قرار بگیرد پس باید بر سر به دست آوردن منبع با یکدیگر رقابت کنند و اگر یک واحد کاری منبع را به دست آورده واحدکاری دیگر باید صبر کند تا منبع آزاد شود.

[1] cooperation synchronization
[2] competition synchronizarion

همروندی

- در ادامه به سه روش مختلف برای اجرای همزمان واحدهای کار به صورت همروند ارائه می‌شود : سمافور [1] ، مانیتور [2] و ارسال پیام [3] .

[1] semaphore
[2] monitor
[3] message passing

سمافور

- یک سمافور سازوکاری ساده است که برای همگام‌سازی واحدهای کاری استفاده می‌شود.
- سمافور راهکاری قدیمی است که همچنان در زبان‌های برنامه نویسی همروند و کتابخانه‌هایی که برای پشتیبانی از همروندی طراحی شده‌اند استفاده می‌شود.
- ادسخر دایکسترا [1] در سال ۱۹۶۵ سمافور را به عنوان راه‌حلی برای همگام‌سازی داده‌های مشترک بین واحدهای پردازش طراحی کرد.

[1] Edsger Dijkstra

سمافور

- برای حفاظت از منابع مشترک باید یک محافظ [1] در اطراف منبع مشترک قرار بگیرد.
- یک محافظ در واقع وسیله‌ای است که به یک واحدکار اجازه می‌دهد که به منبعی دسترسی پیدا کند اگر شرایط آن برقرار باشد.
- سمافور در واقع پیاده سازی چنین محافظی است.
- یک سمافور ساختار داده‌ای است که از یک عدد صحیح و یک صف تشکیل شده است. این صف که صف توصیف وظایف [2] نامیده می‌شود، اطلاعات مربوط به اجرای وظایف (واحدهای کار) را در خود نگه می‌دارد.

[1] guard
[2] task descriptor queue

سمافور

- یک روش ساده برای پیاده سازی سمافور به اینگونه است که اطلاعات هر واحدکاری که نیاز به دسترسی به منبع مشترک دارد، در یک صف نگهداری شده و سپس دسترسی به آنها به طور ترتیبی داده می‌شود.
- دو نوع عملیات مهم بر روی سمافور عملیات انتظار [1] و آزادسازی [2] نامیده می‌شوند.

[1] wait
[2] release

سمافور

- از شمارنده سمافور می‌توان به عنوان شمارنده برای تعداد سلول‌های یک آرایه و یا شمردن تعداد وظیفه‌هایی که می‌توانند به طور همزمان به یک منبع دسترسی داشته باشند استفاده کرد.
- یک سمافور دودویی [1] سمافوری است که شمارندهٔ آن تنها بتواند صفر و یا یک باشد.

[1] binary semaphore

سمافور

- توابع انتظار و آزادسازی سمافور می‌توان به صورت زیر پیاده سازی کرد.

```
wait(semaphore) :
    if semaphore.counter > 0 :
        semaphore.counter --
    else
        # put the caller in semaphore's queue
        # attempt to transfer control to some ready task
        # (if the task-ready queue is empty, deadlock occurs)

release(semaphore) :
    if semaphore.queue.empty() :
        # no task is waiting
        semaphore.counter ++
    else
        # put the calling task in the task-ready queue
        # transfer control to a task from semaphore's queue
```

سمافور

- در زمان کامپایل نمی‌توان بررسی کرد که سمافورها درست استفاده شده‌اند، بنابراین ممکن است برنامه‌ای که از سمافور استفاده می‌کند با بن‌بست مواجه شود.
- در زبان سی++ سمافور دودویی توسط ساختار داده mutex پیاده سازی شده است.

سمافور

- در یک برنامه تولید–مصرف داده بر روی یک بافر، می‌توان از سمافور به صورت زیر استفاده کرد.

```
sem = Semaphore(1)
fullspots = Semaphore(0)
emptyspots = Semaphore(BUFLEN)

def producer() :
    while True :
        # -- produce VALUE --
        wait(emptyspots)
        wait(sem)
        # DEPOSIT(VALUE)
        release(sem)
        release(fullspots)
```

سمافور

```
۱ def consumer() :
۲     while True :
۳         wait(fullspots)
۴         wait(sem)
۵         # FETCH(VALUE)
۶         release(sem)
۷         release(emptyspots)
۸         # -- consume VALUE --
```

مانیتور

- یک روش دیگر برای همگام‌سازی داده‌های مشترک استفاده از مانیتور [1] است.
- مانیتور ساختار داده‌ای است که علاوه بر دربرگرفتن مکانیزم‌های لازم برای همگام‌سازی، داده را نیز در بر می‌گیرد، بنابراین داده مشترک در درون خود مانیتور است.
- از آنجایی که داده در درون مانیتور جای می‌گیرد، تنها یک دسترسی در هر لحظه به استفاده کننده می‌توان داد.
- اگر در یک مانیتور یک بافر قرار بگیرد همچنان وظیفه برنامه نویس است که اطمینان حاصل کند در هنگام خالی بودن به خواننده و در هنگام پر بودن بافر به نویسنده دسترسی داده نشود، وگرنه برنامه با بن‌بست مواجه می‌شود.
- اگر در زبان جاوا یک کلاس تعریف کنیم که همهٔ توابع آن همزمان باشند، و از آن کلاس یک شیء بسازیم در واقع یک مانیتور ساخته‌ایم، زیرا توابع کلاس همگام سازی بر روی داده‌هایی را انجام می‌دهند که در شیء قرار دارند.

[1] monitor

ارسال پیام

- گاهی لازم است دو واحدکاری با یکدیگر ارتباط برقرار کنند. در چنین مواقعی فعالیت ها به پیام‌های دریافتی بستگی پیدا می‌کند.
- برای مثال فرض کنید یک واحد کاری در حال انجام محاسبات است و در همان هنگام واحدکاری دوم نیاز دارد که واحدکاری اول محاسباتی را انجام دهد. واحد کاری اول نمی‌تواند کار خود را متوقف کند، بنابراین واحدکاری دوم پیامی را در صندوق پیام‌های واحد اول ارسال می‌کند و به محض اینکه واحدکاری اول فرصت پیدا کرد، پیام را دریافت و محاسبات مورد نظر را انجام می‌دهد.
- بنابراین در این مکانیزم واحدهای کاری با یکدیگر ارتباط برقرار می‌کنند و پیام‌های خود را در صف پیام‌های یکدیگر ارسال می‌کنند.
- در زبان سی++ مکانیزم ارسال پیام توسط متغیرهای وضعیت `condition variables` پیاده سازی شده است.

راست : همروندی

- یکی از اهداف مهم طراحی زبان راست، افزایش قابلیت اطمینان در برنامه نویسی همروند است. توسط راست می‌توان به طور امن و کارامد برنامه‌های همروند ایجاد کرد.
- در برنامه نویسی همروند اجزای مختلف برنامه به طور مستقل اجرا می‌شوند. معمولا دنبال کردن این برنامه‌ها توسط برنامه نویس کار مشکلی است بدین دلیل که روند برنامه در اجرا معمولا متفاوت از روند برنامه در هنگام دیباگ است و بنابراین پیدا کردن خطاهای برنامه نویسی و دسترسی‌های غیر مجاز در این برنامه‌ها به طور ذاتی مشکل است.
- با استفاده از قوانین مالکیت و قرض دادن بسیاری از خطاهای برنامه نویسی همروند در زمان کامپایل قابل شناسایی هستند.

راست : همروندی

- از آنجایی که ریسه‌ها همزمان اجرا می‌شوند، هیچ تضمینی برای ترتیب اجرای آن‌ها وجود ندارد. بنابراین مشکلات متعددی در این همروندی ممکن است به وجود بیاید.

۱. وضعیت رقابتی [1] وقتی اتفاق می‌افتد که چندین ریسه به یک داده یا یک منبع به طور همزمان دسترسی پیدا می‌کنند، در حالی که ترتیب دسترسی آنها مشخص نیست.

۲. بن بست [2] وقتی اتفاق می‌افتد که چند ریسه منتظر یکدیگر بمانند و بنابراین سیستم با بن‌بست روبرو می‌شود.

۳. نتیجه نادرست یا دسترسی غیر مجاز وقتی اتفاق می‌افتد که به ازای یک ترتیب خاص از اجرای ریسه‌ها نتیجه مورد نظر به دست بیاید و در عین حال تکرار آن سناریو به سادگی امکان پذیر نیست.

[1] race condition
[2] deadlock

راست : همروندی

- برای ساختن یک ریسه از تابع spawn :: thread استفاده می‌شود. این تابع یک تابع یا یک بستار به عنوان ورودی دریافت می‌کند.

```
use std::thread;
use std::time::Duration;
fn main() {
    thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });
    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
}
```

راست : همروندی

- می‌توانستیم در مثال قبل به جای بستار یک تابع را به spawn ارسال کنیم.

```
۱ fn print_th() {
۲     for i in 1..10 {
۳         println!("hi number {} fron spawned print_th", i);
۴         thread::sleep(Duration::from_millis(1));
۵     }
۶ }
۷ fn main() {
۸     thread::spawn(print_th);
۹ }
```

راست : همروندی

- وقتی یک ریسه در یک برنامه به همراه برنامه اصلی اجرا شود، ممکن است برنامه اصلی قبل از ریسه به اتمام برسد، و بنابراین ممکن است ریسه کار خود را به اتمام نرساند.
- بدین منظور گاهی نیاز داریم یک ریسه برای یک ریسه دیگر صبر کند. این کار با متود `join` امکان پذیر است.

راست : همروندی

```rust
fn main() {
    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });
    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
    }
    handle.join().unwrap();
}
```

راست : همروندی

- در مثال قبل از متود `unwrap` استفاده کردیم. با استفاده از این متود، اگر خطایی در هنگام پیوستن ریسه اتفاق بیافتد، آن خطا به کاربر نمایش داده می‌شود.

- در یک ریسه می‌توان متغیرهای محلی را نیز تسخیر [1] کرد. برای مثال در برنامه زیر ریسه از وکتوری که در برنامه اصلی تعریف شده استفاده می‌کند.

```
use std::thread;
fn main() {
    let v = vec![1, 2, 3];
    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    }); // compile error
    handle.join().unwrap();
}
```

[1] capture

راست : همروندی

- اما کامپایلر پیام خطا صادر می‌کند. در اینجا ریسه در واقع متغیر v را قرض گرفته است، اما کامپایلر نمی‌تواند اطمینان حاصل کند که قبل از اتمام اجرای ریشه متغیر v معتبر می‌ماند. برای مثال فرض کنید قبل از اتمام ریسه، متغیر v را توسط drop به صورت زیر تخریب کنیم.

```
use std::thread;
fn main() {
    let v = vec![1, 2, 3];
    let handle = thread::spawn(|| {
        println!("Here's a vector: {:?}", v);
    }); // compile error (what if v is dropped while executing?)
    drop(v); // oh no!
    handle.join().unwrap();
}
```

راست : همروندی

- بنابراین مالکیت متغیر v باید به ریسه انتقال پیدا کند. برای این کار از کلمهٔ move استفاده می‌کنیم.

```
۱ fn main() {
۲     let v = vec![1, 2, 3];
۳     let handle = thread::spawn(move || {
۴         println!("Here's a vector: {:?}", v);
۵     });
۶     handle.join().unwrap();
۷ }
```

راست : ارسال پیام

- یکی از راه‌های ارتباط بین ریسه‌ها ارسال پیام است.
- طراحان زبان راست همانند طراحان زبان گو [1] پیشنهاد می‌کنند برنامه نویسان به جای ایجاد ارتباط توسط حافظهٔ مشترک از مکانیزم ارسال پیام استفاده کنند.
- برای ایجاد سازوکار ارسال پیام، زبان راست کتابخانه‌ای را برای استفاده از کانال‌ها [2] ی ارتباطی پیاده سازی کرده است. داده‌ها بر روی کانال‌ها بین ریسه‌ها مبادله می‌شوند.
- یک کانال از دو بخش تشکیل شده است : یک فرستنده و یک گیرنده. فرستنده پیام را بر روی کانال ارسال می‌کند و گیرنده آن را دریافت می‌کند.
- وقتی فرستنده و گیرنده از بین بروند کانال بسته می‌شود.

[1] Go programming language
[2] channels

راست : ارسال پیام

- یک کانال توسط تابع `mpsc::channel()` از کتابخانهٔ استاندارد `std::sync` ساخته می‌شود. `mpsc` به معنی چند تولید کننده، یک مصرف کننده [1] است. در واقع در پیاده سازی کانال در کتابخانه استاندارد، در یک کانال چندین تولید کننده می‌توانند پیام ارسال کنند و تنها یک مصرف کننده می‌تواند پیام‌ها را دریافت کند.
- تابع `mpsc::channel()` یک دوتایی تولید می‌کند که عنصر اول آن فرستندهٔ کانال است و عنصر دوم آن گیرندهٔ کانال.

[1] multiple producer، single consumer

راست : ارسال پیام

- یک ریسه می‌تواند به صورت زیر بر روی کانال پیام ارسال کند.

```
use std::sync::mpsc;
use std::thread;
fn main() {
    let (tx, rx) = mpsc::channel();
    thread::spawn(move || {
        let val = String::from("hi");
        tx.send(val).unwrap();
    });
}
```

راست : ارسال پیام

- ریسه نیاز دارد مالکیت فرستندهٔ کانال را در اختیار بگیرد تا بتواند پیام ارسال کند. متود `send` نوع `Result<T,F>` را بازمی‌گرداند و در صورتی که گیرندهٔ پیام از بین رفته باشد پیام خطا باز می‌گرداند. متود `unwrap` در صورت عدم موفقیت پیام خطا تولید می‌کند.

راست : ارسال پیام

- یک ریسه یا ریسهٔ اصلی به صورت زیر می‌تواند از روی کانال پیام دریافت کند.

```
۱  use std::sync::mpsc;
۲  use std::thread;
۳  fn main() {
۴      let (tx, rx) = mpsc::channel();
۵      thread::spawn(move || {
۶          let val = String::from("hi");
۷          tx.send(val).unwrap();
۸      });
۹      let received = rx.recv().unwrap();
۱۰     println!("Got: {}", received);
۱۱ }
```

راست : ارسال پیام

- گیرنده دو متود برای دریافت پیام دارد. متود `recv` ریسه را متوقف می‌کند و منتظر می‌ماند تا پیامی از روی کانال دریافت کند.

راست : ارسال پیام

- مقدار بازگشتی متود از نوع `Result<T,F>` است. که در صورت موفقیت پیام را دریافت و در صورت عدم موفقیت (چنانچه فرستنده متوقف شده باشد) پیام خطا ارسال می‌کند.
- متود `try-recv` ریسه را متوقف نمی‌کند. در صورتی که کانال باز باشد ولی پیامی ارسال شده باشد مقدار `Ok` ودر غیر اینصورت مقدار `Err` را توسط نوع داده شمارشی باز می‌گرداند. سپس ریسه می‌تواند کارهای دیگر خود را انجام می‌دهد و در یک حلقه کانال را دوباره بررسی کند.

راست : ارسال پیام

- وقتی از متود ()Send استفاده می‌کنیم، مالکیت مقدار فرستاده شده منتقل می‌شود. این بدسن دلیل است که اطمینان حاصل می‌شود که مقدار فرستاده شده توسط دو ریسه به طور همزمان تغییر نمی‌کند.
- بنابراین برنامه زیر در زمان کامپایل پیام خطا صادر می‌کند.

```
۱  use std::sync::mpsc;
۲  use std::thread;
۳  fn main() {
۴      let (tx, rx) = mpsc::channel();
۵      thread::spawn(move || {
۶          let val = String::from("hi");
۷          tx.send(val).unwrap();
۸          println!("val is {}", val); // compile error
۹      });
۱۰     let received = rx.recv().unwrap();
۱۱     println!("Got: {}", received);
۱۲ }
```

راست : ارسال پیام

- بر روی یک کانال می‌توان تعداد متعددی پیام به صورت زیر ارسال کرد.

```
use std::sync::mpsc;
use std::thread;
use std::time::Duration;
fn main() {
    let (tx, rx) = mpsc::channel();
    thread::spawn(move || {
        let vals = vec![
            String::from("hi"),
            String::from("from"),
            String::from("the"),
            String::from("thread"),
        ];
```

راست : ارسال پیام

```
        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });
    for received in rx {
        println!("Got: {}", received);
    }
}
```

راست : ارسال پیام

- در صورتی که بخواهیم بر روی یک کانال بیش از یک فرستنده داشته باشیم، باید فرستنده کانال را توسط clone() کپی عمیق کنیم، زیرا مالکیت فرستنده کانال در ازسال پیام باید به ریسه انتقال پیدا کند.

```
let (tx, rx) = mpsc::channel();

let tx1 = tx.clone();
thread::spawn(move || {
    let vals = vec![
        String::from("hi"),
        String::from("there"),
    ];
    for val in vals {
        tx1.send(val).unwrap();
        thread::sleep(Duration::from_secs(1));
    }
});
```

راست : ارسال پیام

```
    thread::spawn(move || {
        let vals = vec![
            String::from("more"),
            String::from("messages"),
        ];
        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });
    for received in rx {
        println!("Got: {}", received);
    }
```

راست : داده اشتراکی

- علاوه بر سازوکار ارسال پیام، چند ریسه می‌توانند توسط حافظه مشترک یا داده مشترک نیز با یکدیگر ارتباط برقرار کنند.
- در مکانیزم ارسال پیام وقتی پیام بر روی کانال ارسال شد، فرستنده مالک داده نیست مشکل حافظه اشتراکی این است که چند ریسه می‌خواهند همزمان مالکیت حافظه را به دست بیاورند که ممکن است باعث ایجاد خطا شود. راست در زمان کامپایل اطمینان حاصل می‌کند که این اتفاق نمی‌افتد.

راست : داده اشتراکی

- میوتکس `mutex` یا قفل [1] مخفف کلمه ممانعت متقابل [2] است، بدین معنی که با استفاده از مکانیزم قفل تنها یک ریسه می‌تواند در یک زمان به داده دسترسی داشته باشد
- برای دسترسی به یک حافظهٔ اشتراکی ریسه باید ابتدا کلید قفل را به دست آورد. قفب در واقع یک ساختار داده است که ریسه‌هایی که نیاز به دسترسی به حافظه مشترک را دارند را در یک صف انتظار قرار می‌دهد. سپس به ترتیب دسترسی به ریسه‌ها داده می‌شودو هر ریسه‌ای که دسترسی را به دست آورد ورودی را قفل می‌کند و در هنگام اتمام کار قفل را آزاد می‌کند.

[1] lock
[2] mutual exclusion

راست : داده اشتراکی

- نوع `Mutex<T>` یک قفل ایجاد می‌کند که دادهٔ اشتراکی آن از نوع T است. دقت کنید که داده در درون قفل قرار دارد و می‌توان گفت که قفل‌ها در راست به صورت مانیتور پیاده سازی شده‌اند.
- برای مثال :

```
use std::sync::Mutex;
fn main() {
    let m = Mutex::new(5);
    {
        let mut num = m.lock().unwrap();
        *num = 6;
    }
    println!("m = {:?}", m);
}
```

راست : داده اشتراکی

- برای ارتباط چند ریسه توسط سازوکار قفل باید همهٔ ریسه‌ها بتوانند مالکیت قفل را در اختیار بگیرند. برای دادن مالکیت اشتراکی از اشاره‌گرهای هوشمند استفاده می‌کنیم. نوع `Arc<T>` تعداد دسترسی‌ها به یک متغیر را شمارش می‌کند، بنابراین می‌تواند هنگامی که هیچ دسترسی به متغیر وجود ندارد فضای حافظه آن را آزاد کند.

راست : داده اشتراکی

- بنابراین برای دسترسی چند ریسه به یک قفل به طور همزمان به صورت زیر عمل می‌کنیم.

```
۱  use std::sync::{Arc, Mutex};
۲  use std::thread;
۳  fn main() {
۴      let counter = Arc::new(Mutex::new(0));
۵      let mut handles = vec![];
۶      for _ in 0..10 {
۷          let counter = Arc::clone(&counter);
۸          let handle = thread::spawn(move || {
۹              let mut num = counter.lock().unwrap();
۱۰             *num += 1;
۱۱         });
```

راست : داده اشتراکی

```
            handles.push(handle);
        }
        for handle in handles {
            handle.join().unwrap();
        }
        println!("Result: {}", *counter.lock().unwrap());
}
```

راست : داده اشتراکی

- دقت کنید که قفل را به صورت غیر قابل تغییر تعریف کردیم ولی دادهٔ درون آن قابل تغییر است.
- قفل‌ها ممکن است به نحوی استفاده شوند که باعث ایجاد بن بست شود. راست نمی‌تواند از این خطای احتمالی در زمان کامپایل جلوگیری کند. بن بست وقتی رخ می‌دهد که دو ریسه در انتظار یکدیگر برای آزادسازی قفل توسط طرف مقابل بمانند.

جاوا : همروندی

- در زبان جاوا برای ایجاد یک ریسه باید کلاسی که عملیات ریسه را پیاده سازی می‌کند یا از کلاس `Thread` به ارث ببرد د یا رابط `Runnable` را پیاده سازی کند. سپس عملیات ریسه در متود `run` قرار داده می‌شود. ریسع با فراخوانی متود `()start` بر روی شیء آغاز به کار می‌کند و برای انتظار برای اتمام ریسه از متود `()join` استفاده می‌کند.
- سمافورها توسط کلاس `Semaphore` پیاده سازی شده‌اند که دو متود `acquire` و `release` برای آن تعریف شده است.
- یک متود می‌تواند با استفاده از کلمهٔ `synchronized` تعریف شود. متودی که به صورت همگام شده [1] تعریف شده است باید عملیت خود را به اتمام برساند تا بتواند دوباره توسط یک ریسه فراخوانی شود.
- پیاده سازی این متودها به این‌صورت است که هر شیء در جاوا یک قفل دارد و برای فراخوانی یک تابع همزمان، ابتدا باید شیء قفل شود، وقتی اجزای متود به پایان رسید، قفل آزاد می‌شود.

[1] synchronized

جاوا : همروندی

- کلاسی که همهٔ توابع آن به صورت همگام شده تعریف شده باشند در واقع یک مانیتور است.
- اگر بخواهیم به جای همگام‌سازی کل یک متود فقط قسمتی از آن را همگام‌سازی کنیم، می‌توانیم از کلمه synchronized قبل از یک بلوک این کار را انجام دهیم.

```
synchronized (expression) {
    statements
}
```

- در اینجا expression در واقع یک شیء است که بر روی آن یک قفل گرفته می‌شود.
- هر شیء در جاوا یک صف در اختیار دارد که این صف اطلاعات ریسه‌هایی که نیاز به آن شیء دارند را در خود نگه می‌دارد.

جاوا : همروندی

- برای ارتباط بین ریسه‌ها، همهٔ اشیای جاوا که از کلاس جد* به نام کلاس `Object` به ارث می‌برند، متودهایی به نام `wait` ، `notify` و `notifyAll` دارند.
- متود انتظار `wait` موجب می‌شود یک ریسه در صف انتظار یک شی وارد شود و منتظر می‌ماند تا وقتی که یک ریسهٔ دیگر متود اعلام `notify` را فراخوانی کند. وقتی متود اعلام فراخوانی می‌شود، در واقع تغییری در سیستم رخ داده است و ریسه‌ای که در صف انتظار بوده است باید بررسی کند که تغییر صورت گرفته نیازش را برآورده می‌سازد یا خیر.

جاوا : همروندی

- در برنامه زیر برای همگام سازی و خواندن و نوشتن بر روی یک صف از مکانیزم انتظار و اعلام استفاده شده است.

```
// Queue
// This class implements a circular queue for storing int
// values. It includes a constructor for allocating and
// initializing the queue to a specified size. It has
// synchronized methods for inserting values into and
// removing values from the queue.
class Queue {
    private int [] que;
    private int nextIn, nextOut, filled, queSize;
    public Queue(int size) {
        que = new int [size];
        filled = 0;
        nextIn = 1;
        nextOut = 1;
```

جاوا : همروندی

```
        queSize = size;
    } //** end of Queue constructor
    public synchronized void deposit (int item)
           throws InterruptedException {
        try {
            while (filled == queSize)
            wait();
            que [nextIn] = item;
            nextIn = (nextIn % queSize) + 1;
            filled++;
            notifyAll();
        } //** end of try clause
        catch(InterruptedException e) {}
    } //** end of deposit method
```

جاوا : همروندی

```
۱      public synchronized int fetch()
۲             throws InterruptedException {
۳          int item = 0;
۴          try {
۵              while (filled == 0)
۶              wait();
۷              item = que [nextOut];
۸              nextOut = (nextOut % queSize) + 1;
۹              filled--;
۱۰             notifyAll();
۱۱         } //** end of try clause
۱۲         catch(InterruptedException e) {}
۱۳         return item;
۱۴     } //** end of fetch method
۱۵ } //** end of Queue class
```

جاوا : همروندی

- حال تولیدکننده و مصرف‌کنندهٔ صف به صورت دو ریسه به صورت زیر تعریف می‌شوند.

```
۱  class Producer extends Thread {
۲      private Queue buffer;
۳      public Producer(Queue que) {
۴          buffer = que;
۵      }
۶      public void run() {
۷          int new_item;
۸          while (true) {
۹              //-- Create a new_item
۱۰             buffer.deposit(new_item);
۱۱         }
۱۲     }
۱۳ }
```

جاوا : همروندی

```
۱  class Consumer extends Thread {
۲      private Queue buffer;
۳      public Consumer(Queue que) {
۴          buffer = que;
۵      }
۶      public void run() {
۷          int stored_item;
۸          while (true) {
۹              stored_item = buffer.fetch();
۱۰             //-- Consume the stored_item
۱۱         }
۱۲     }
۱۳ }
```

جاوا : همروندی

```
۱ // in main :
۲ Queue buff1 = new Queue(100);
۳ Producer producer1 = new Producer(buff1);
۴ Consumer consumer1 = new Consumer(buff1);
۵ producer1.start();
۶ consumer1.start();
۷ producer1.join();
۸ consumer1.join()
```

جاوا : همروندی

- در جاوا همچنین کلاس‌هایی برای دسترسی تجزیه ناپذیر [1] به داده‌ها فراهم شده است. برای مثال با استفاده از کلاس `AtomicInteger` می‌توان یک عدد صحیح تعریف کرد که ریسه‌های مختلف به آن دسترسی پیدا می‌کنند. در واقع این کلاس یک مکانیزم قفب دارد که این مکانیزم در سطح سخت افزار پیاده سازی شده است.

[1] atomic

جاوا : همروندی

- علاوه بر مکانیزم همگام‌سازی به صورت همگام‌شده یا `synchronized` ، در جاوا مکانیزم قفل نیز وجود دارد که در کلاس `ReentrantLock` پیاده سازی شده است توسط این قفل که در واقع همان مکانیزم ممانعت متقابل است می‌توان دسترسی به حافظه اشتراکی را به تنها یک ریسه محدود کرد.

جاوا : همروندی

- این قفل‌ها به صورت زیر استفاده می‌شوند.

```
۱ Lock lock = new ReentrantLock();
۲ . . .
۳ Lock.lock();
۴ try {
۵     // The code that accesses the shared data
۶ } finally {
۷     Lock.unlock();
۸ }
```