به نام خداوند بخشنده‌ی مهربان

شماره‌ی ۳ ۲۹ مهر ۱۴۰۰

کد نامه

ویژه‌ی دانش‌جویان مبانی برنامه‌سازی نیم‌سال اول ۱۴۰۱-۱۴۰۰ دانشکده‌ی مهندسی کامپیوتر دانشگاه صنعتی شریف

تاریخچه‌ی برنامه‌نویسی کامپیوتر

زبان‌های برنامه‌نویسی از هفتاد سال پیش تا امروز، راه دور و درازی را پشت سر گذاشته‌اند. امروز که می‌توانیم به راحتی با کلماتی معنی‌دار و ساده حرف دلمان را به لپ‌تاپمان بگوییم، شاید بیش‌تر از هر وقتی باید قدردان انسان‌های فداکاری باشیم که با کامپیوترهای غول‌پیکر و زبان‌نفهم دوران خودشان سروکله می‌زدند و راه را برای ما هموار می‌کردند. در این شماره از کدنامه قصد داریم کمی از مسیری که طی‌شده تا زبان‌هایی مثل سی پا به این دنیا بگذارند را با یک‌دیگر بررسی کنیم. با کدنامه همراه باشید.

تقویم درس در هفته‌ی پیش‌رو

۲ آبان

تعطیل رسمی

۴ آبان

دستورات شرط و حلقه

۸-۹ آبان

مهلت تمرین ۲ و انتشار تمرین ۳ – شرط و حلقه

؟ چرا بهتر است کدی که می‌نویسیم، «تمیز» باشد؟

ممکن است تصور کنید که به کدی که می‌نویسید، مسلط هستید و نیازی نیست حتما کد را به شکلی تمیز و منظم نگارش کنید، اما در واقعیت، بسیاری از پروژه‌های برنامه‌نویسی به شکل تیمی و به همراه اعضایی که لزوما با آن‌ها دوست یا آشنا نیستید، انجام می‌شوند. هم‌تیمی‌های شما، لزوما از نحوه‌ی برنامه‌نویسی شما آگاهی کامل ندارند و در نتیجه هم‌کاری در نوشتن کد به سخت‌ترین شکل ممکن صورت می‌پذیرد. هم‌چنین، ممکن است فردی پس از شما، ادامه‌ی ساخت برنامه را به عهده بگیرد و در صورت تمیز نبودن کد شما، رفع ایرادات کد برای وی، بسیار سخت خواهد شد. افزون بر آن، ممکن است خودتان چند سال بعد، به سبک متفاوتی کد زده و به راحتی متوجه کارکرد برنامه‌های سال‌های پیش خود، نشوید.

در شماره‌ی بعدی کدنامه، به طور مفصل، در خصوص کدنویسی تمیز و اهمیت آن، صحبت خواهد شد.

تاریخچه‌ی برنامه‌نویسی کامپیوتر

زهرا آذر

اصلا چه شد که انسان‌ها به‌سراغ برنامه‌نویسی رفتند؟

سوال فوق، اولین سوالی است که باید در رابطه با تاریخچه‌ی برنامه‌نویسی از خود بپرسیم. برای پاسخ دادن به آن، نگاهی به زندگی یک دانشمند پیش از شروع برنامه‌نویسی می‌اندازیم. برای مثال یک فیزیک‌دان برای آن‌که چند عدد بزرگ را با هم جمع یا تفریق کند و روی آن‌ها محاسباتی انجام دهد، مجبور به ساعت‌ها محاسبه بود؛ بماند که بعد از این همه محاسبه، هیچ تضمینی نبود که خطایی در این میان رخ نداده باشد. چه بسا زمانی که یک دانشمند باید صرف محاسبات تکراری می‌کرد، بیش از زمانی بود که به کشفیات جدید می‌پرداخت. در نتیجه، انسان‌ها به این فکر افتادند که وقت ارزشمند خودشان را با کارهای تکراری یا به عبارتی دیگر کارهایی که دارای **الگوی ثابتی** بودند، هدر ندهند و همان‌طور که در طول تاریخ، ابر و باد و مه و خورشید و فلک را به جای خود به‌کارگرفته‌بودند، محاسبات را نیز به موجود دیگری بسپارند. پیدا کردن آن موجود، خودش قصه‌ی مفصلی دارد؛ ولی ما فعلا به این‌که چه‌طور با آن موجود **صحبت کردند** می‌پردازیم، زیرا این‌جا بود که برنامه‌نویسی به وجود آمد. پس فعلا قبول کنید که موجودی به اسم کامپیوتر یا رایانه را با استفاده از چند موجود دیگر به نام gate های منطقی ساختند و می‌خواستند با آن صحبت کنند، ولی او حرف‌های کمی را می‌فهمید. بنابراین در برنامه‌نویسی – همان‌طور که در فلوچارت کشیدن به وضوح مشاهده کردید – ما یک عملیات که الگوی مشخصی دارد را به شکل یک الگوریتم قابل فهم برای رایانه در می‌آوریم. اولین عملیاتی که به این شکل درآمده و به کامپیوترها سپرده شد، جمع زدن ساده بود، ولی در حال حاضر با پیشرفت‌هایی که در سخت‌افزار و ریاضیات به وجود آمده است، حتی توانسته‌ایم عملیاتی مثل «صحبت کردن» را – که به نظر بسیار پیچیده و غیرقابل الگویابی است – با پیدا کردن قواعدش، به شکل یک الگوریتم درآورده و به رایانه‌ها بیاموزیم.

از «صفر و یک» تا «هوش مصنوعی»

این سوال که «اولین زبان برنامه نویسی دقیقا چه بود»، سوالی است که نمی‌توانیم به راحتی پاسخی برای آن پیدا کنیم؛ چه بسا کسی در خلوت خودش زبانی ابداع کرده و به گوش کسی هم نرسیده باشد. اما، اولین زبان برنامه‌نویسی شناخته شده را شخصی به اسم Ada Lovelace برای کمک به یک ریاضی‌دان در محاسبه‌ی اعداد برنولی ابداع کرد. این اتفاق حدود ۱۰۰ سال پیش از ساخته شدن اولین کامپیوترهای مدرن رخ داد و به نوعی از اولین قدم‌ها در این مسیر بود.

می‌توان گفت پیشرفت زبان‌های برنامه‌نویسی – به طور سامان‌یافته – وقتی شروع شد که فردی به اسم John von Neumann، دو اصل اساسی زیر را برای توسعه‌ی برنامه‌نویسی ارائه کرد:

۱. اولین اصل که «تکنیک برنامه‌ی مشترک» نام دارد، به طور کلی تاکید می‌کند که سخت‌افزار باید هرچه ساده‌تر باشد و پیچیدگی‌ها به نرم‌افزار واگذار شود.

۲. اصل دوم با نام «انتقال کنترل به‌صورت شرطی»، به برنامه‌نویس اجازه می‌دهد که چندین بلوک به اسم «زیرروال» در برنامه‌ی خود داشته باشد و به عبارت دیگر برای بخش‌های خاصی از برنامه‌اش، اسامی‌ای انتخاب کند تا هروقت لازم شد آنها را صدا بزند. در ادامه‌ی درس، با این مفهوم بیش‌تر آشنا خواهید شد.

می‌توان زبان‌های برنامه‌نویسی پس از آن را به پنج نسل تقسیم کرد. در ادامه، با سبک و سیاق زبان‌های هر نسل آشنا می‌شویم:

نسل اول

اگر به یک قطعه کد از زبان‌های برنامه‌نویسی نسل اول (که احتمالا روی نوعی کاغذ خاص یا کارت، پانچ شده است) نگاه کنید، چیزی جز صفر و یک نخواهید دید. این زبان‌ها در واقع بدون هیچ واسطه‌ای با سخت‌افزار صحبت می‌کنند و به همین دلیل، به آن‌ها زبان ماشین نیز گفته می‌شود. پاسخ به این سوال که کامپیوتر همین صفر و یک‌ها را چه‌طور متوجه می‌شود، موضوع درس «مدارهای منطقی» و دروس پس از آن است که در ترم‌های آینده با آن‌ها آشنا می‌شوید؛ هر چند اگر عجله دارید، می‌توانید دوباره سری به شماره‌ی دوم کدنامه بزنید و با این دید، آن را مطالعه کنید.

نسل دوم

احتمالا اسم زبان Assembly را در گوشه و کنار دنیای برنامه‌نویسی زیاد شنیده‌اید. این زبان که مربوط به نسل دوم از زبان‌های برنامه‌نویسی می‌شود، وظیفه‌ی مهم نجات بشر از صفر و یک را بر عهده داشته است. البته، اگر پس از یادگیری یک زبان سطح بالاتر به این زبان مراجعه کنید، نه تنها اسمبلی ناجی بزرگی به نظر نخواهد رسید، بلکه کار شما را سخت‌تر هم خواهد کرد؛ ولی اگر از دید برنامه‌نویسانی که فقط با اعداد باینری سروکار داشتند به این زبان نگاه کنید، همین که چند کلمه‌ی معنی‌دار در آن می‌بینید، یعنی معجزه!

هر دستور اسمبلی، دقیقا معادل یک دستور زبان ماشین است؛ به همین دلیل می‌گوییم دستورات زبان اسمبلی **رابطه یا تناظر یک به یکی** با زبان ماشین دارند و این یعنی برخلاف زبان‌های نسل اول، به برنامه‌ای احتیاج داریم که این زبان را به زبان ماشین، معادل‌سازی کند که به آن **Assembler** گفته می‌شود (خود اسمبلر هم یک برنامه است که می‌تواند به زبان ماشین نوشته شده باشد). به کمک این زبان، می‌توانیم جواب سوال معروف «خود اولین کامپایلر، به چه زبانی نوشته شده است؟»

را بدهیم، زیرا اسمبلی نیازی به کامپایلر نداشته و ما را از حلقه‌ی باطل «خودش را چه کسی کامپایل می‌کند؟»، خارج می‌کند. (دقت کنید که فرآیند اسمبل کردن، بسیار ساده‌تر از کامپایل کردن است. اسمبلر به طور خط به خط، کد اسمبلی را به کد ماشین تبدیل کرده و اجرا می‌کند؛ ضمنا، هر دستور اسمبلی را دقیقا به یک دستور باینری برمی‌گرداند. این در حالی است که کامپایلر برنامه را به طور کامل به زبان ماشین تبدیل کرده و سپس آن را اجرا می‌کند؛ هم‌چنین، ممکن است کامپایلر هر دستور را به چندین دستور ماشین تبدیل کند.)

نسل سوم

نوبتی هم که باشد، نوبت زبان C خودمان است. این زبان و بسیاری از مشتقات آن ازجمله C++ و Java، از این نسل هستند. این نسل از زبان‌ها، ایراد بزرگی که بر زبان‌های نسل قبلی وارد بود را تا حدودی برطرف می‌کنند؛ زبان ماشین و همچنین اسمبلی، کاملا وابسته به معماری کامپیوتر هستند، در حالی که کد نوشته‌شده به زبان C را می‌توان در رایانه‌هایی با معماری‌های متفاوت استفاده کرد. از آن جایی که در این ترم با زبان C بسیار سر و کار خواهید داشت، در این‌جا تاریخچه‌ی این زبان را کمی مفصل‌تر بررسی می‌کنیم.

تاریخچه‌ی زبان C

حدود ۵۵ سال قبل، زبان BCPL برای نوشتن نرم‌افزارهای سیستم‌عامل و همین‌طور کامپایلرها ابداع شد. چند سال پس از آن، زبان B بر مبنای آن زبان ساخته‌شده و از آن برای ساخت اولین نسخه‌های سیستم‌عامل Unix استفاده شد. سپس، فردی به نام Dennis Ritchie با برطرف کردن نقص‌های این دو زبان، زبان C را براساس آن‌ها ابداع کرد. ریچی از این زبان برای نوشتن نسخه‌های بعدی سیستم‌عامل Unix بهره گرفت، اما زبان C ماندگارتر از والدینش بود و بعدها از آن برای نوشتن انواع سیستم‌عامل‌ها استفاده شد. جالب است بدانید اکثر سیستم‌عامل‌های فعلی نیز با همین زبان یا یکی از مشتقات آن نوشته شده‌اند. اما، همین پراستفاده بودن C، سبب دردسر شد؛ زیرا پس از مدتی، نسخه‌های متعددی از آن برای سخت‌افزارهای مختلف به وجود آمد. در همین زمان بود که سازمان استانداردهای ملی آمریکا و سپس سازمان بین‌المللی استاندارد به میدان آمدند و نسخه‌ی استاندارد و مستقل از سخت‌افزاری را برای C تصویب کردند. این نسخه «ANSI C» نام دارد. پس از آن، این استاندارد به مرور به‌روزرسانی شد و نسخه‌های دیگری مانند C99 و C11 به وجود آمدند. آخرین نسخه‌ی استاندارد C نیز حدود سه سال پیش با نام C17 منتشر شد.

نسل چهارم

زبان‌های نسل چهارم، در اصل در ادامه‌ی زبان‌های نسل سوم و برای طراحی برنامه‌هایی با کارایی خاص طراحی شده‌اند. برای مثال، متلب (MATLAB) زبانی است که به طور خاص برای انجام محاسبات عددی ساخته شده است. این زبان‌ها به برنامه‌نویس امکان آن را می‌دهند که کم‌تر با جزئیات درگیر شده و با یک خط کد، کار ده‌ها خط از یک زبان نسل سوم را انجام دهند؛ در نتیجه، سرعت توسعه‌ی برنامه با زبان‌های این نسل، بسیار بالاست.

نسل پنجم

این نسل از زبان‌ها که کم‌تر از سی سال است که پا به عرصه‌ی زبان‌های برنامه‌نویسی گذاشته‌اند، برخلاف چهار نسل قبلی، نیازی به برنامه‌نویسی که الگوریتم حل مسئله را بنویسد، ندارند! در واقع، این زبان‌ها فقط با دانستن مسئله آن را برای ما حل می‌کنند. همان‌طور که از این تعریف می‌توان حدس زد، زبان‌های نسل پنجم بیش‌تر برای ساخت ابزارهای مربوط به هوش مصنوعی استفاده می‌شوند. ویژگی دیگری که می‌توان برای این زبان‌ها در نظر گرفت، ابزارهای بصری آن‌ها برای توسعه‌ی برنامه است؛ البته، هنوز نمی‌توانیم حد و مرز دقیقی برای این نسل از زبان‌ها تعیین کنیم.

دیدیم که در طول این هفتاد سال، زبان‌های برنامه‌نویسی روز به روز بیشتر به زبان انسان شبیه شده، کار را برای برنامه‌نویسان راحت‌تر کرده و از طرفی، امکان توسعه‌ی برنامه‌های پیشرفته‌تری را نیز به ما داده‌اند. حال وقت آن است که با عامل دیگری که برنامه‌نویسی را در سال‌های اخیر آسان کرده است آشنا شویم:

تاریخچه‌ی IDEها

تصور کنید برای کد زدن باید ابتدا برنامه‌ی خود را در یک ویرایشگر متن بنویسید، آن‌را save کنید و با استفاده از یک کامپایلر که برنامه‌ی مجزایی است، اجرا کنید و چک کنید که کار می‌کند یا خیر؛ اگر کار کرد که هیچ و اگر درست کار نکرد، خطاهایی که کامپایلر به شما می‌دهد را روی کاغذ بنویسید، دوباره سراغ ویرایشگر متن بروید و تا زمانی که تمام خطاهای برنامه رفع شود، این کارها را تکرار کنید.

حالا به یک برنامه‌نویس بیست یا سی سال بزرگ‌تر از خودتان مراجعه کنید و شیوه‌ی کد زدن را در آن زمان، از او بپرسید. (اگر چنین کسی در دسترستان نیست، یک بار دیگر داستان کوتاه ترسناک بالا را بخوانید.)

حالا که به عمق فاجعه پی بردید، درک می‌کنید که چرا برخی برنامه‌نویس‌ها دست به‌کار شده‌اند و برای راحتی همکارهای خودشان، برنامه‌هایی به اسم IDE یا همان «محیط توسعه‌ی یکپارچه» را طراحی کرده‌اند.

اولین IDEها تنها دو وظیفه‌ی اصلی ویرایش کد و کامپایل کردن آن را به عهده داشتند. با افزایش توان سخت‌افزاری رایانه‌ها، زمینه برای طراحی IDEهایی که به روش‌های گوناگون به برنامه‌نویس کمک کنند (مانند CodeBlocks)، فراهم شد. در حال حاضر، این برنامه‌های دوست‌داشتنی کلمات اصلی زبان را برایمان رنگی می‌کنند، دنباله‌ی کلماتی که تایپ می‌کنیم را حدس می‌زنند، در خطایابی و debug کردن کد به ما کمک می‌کنند و کم مانده است به جایمان کد بزنند.

پیشرفت و تنوع زبان‌های برنامه‌نویسی و IDEها و هم‌چنین پیشرفت‌های سخت‌افزاری و افزایش حافظه‌ی کامپیوترها، دنیای کامپیوتری‌ای که امروز در اطرافمان می‌بینیم را شکل داده‌اند و کار را برای ما بسیار ساده‌تر کرده‌اند. پس دفعه‌ی بعدی که با زبانی که بی‌شباهت به زبان آدمیزاد نیست، کدی می‌نویسید و با یک دکمه آن‌را اجرا می‌کنید، به یاد تمام خون‌دل‌هایی که در این مسیر خورده شده‌اند، باشید!

آشنایی با خطاهای متداول در کوئرا

سید پارسا نشایی

شما که در حال خواندن این مقاله هستید، به احتمال بسیار زیاد، تمرین صفر خود را در Quera بارگذاری یا همان آپلود کرده‌اید. بارگذاری در کوئرا به جای روش‌های سنتی مانند ایمیل، سبب می‌شود تا بتوانید آرشیوی از ارسال‌های خود را مشاهده کرده و هر یک از آن‌ها که بیش‌تر می‌پسندید را به عنوان ارسال نهایی برای مشاهده توسط دستیاران آموزشی، تیک بزنید. (توجه کنید که تمام ارسال‌های شما – حتی ارسال‌های غیر نهایی – توسط دستیاران آموزشی، قابل مشاهده‌اند، اما تنها ارسال‌های نهایی شما، ملاک نمره‌دهی و ارزیابی درس خواهد بود و سایر ارسال‌های شما، توسط دستیاران نادیده گرفته خواهند شد.) آپلود تمرین صفر توسط شما در کوئرا، مزایایی برای دستیاران آموزشی نیز دارد، از جمله تصحیح راحت‌تر پاسخ‌های شما به کمک قابلیت دانلود دسته‌بندی‌شده‌ی سوال‌ها در کوئرا و نیز مشاهده‌ی دقیق‌تر و راحت‌تر میزان تاخیر هر یک از دانش‌جویان در تحویل تمرین. اما، موارد فوق تنها موارد استفاده از Quera نیستند و در تمرین‌های یک به بعد درس، دلیل اصلی استفاده از Quera، وجود «داور خودکار آنلاین» برای کدهای ارسالی توسط شما است.

داور (judge) خودکار آنلاین

از تمرین یک به بعد، تمرین‌ها پس از ارسال توسط شما، بلافاصله به صورت خودکار توسط کوئرا داوری می‌شوند و نمره‌ی سوال به شما اعلام می‌شود تا بتوانید متناسب با نمره‌ی کسب شده، پاسخ خود را بهبود بخشید و مجددا آن را ثبت (submit) کنید. پس از آپلود هر فایل پاسخ، وضعیت پاسخ به «در حال داوری» تغییر می‌کند و اگر پس از چند ثانیه، صفحه را refresh کنید، نمره‌ی سوال (غالبا از ۱۰۰) در جلوی اسم سوال در لیست «ارسال‌ها» به شما نمایش داده خواهد شد. توجه کنید که در صورتی که نمره‌ی کسب شده در آخرین ارسال (submission) شما، بیش از نمره‌ی کسب شده در submissionای باشد که هم‌اکنون به عنوان «ارسال نهایی» در نظر گرفته شده است، ارسال جدید شما، به صورت خودکار، به عنوان ارسال نهایی در نظر گرفته خواهد شد.

با کلیک روی نمره، می‌توانید لیستی از test caseها را به همراه نمره‌ی شما در هریک، مشاهده کنید. هر test case، یک ورودی نمونه است که به کد شما داده شده است و خروجی کد شما با خروجی مورد انتظار، تطبیق داده شده و در صورت عینا یکسان بودن، نمره‌ی آن را دریافت خواهید کرد. اخذ نکردن نمره‌ی هر تست‌کیس، می‌تواند به یکی از دلایل زیر باشد:

- خطای Wrong Answer – کد شما خروجی درست را تولید نکرده است و احتمالا یکی از نکات سوال تمرین را در نظر نگرفته‌اید.
- خطای Compilation Error – حتما از زبان C به جای زبان C++ استفاده کنید، وگرنه ممکن است به این خطا برخورید.
- خطای Syntax Error – دستوری را در زبان C اشتباه نوشته‌اید. با کامپایل کردن دوباره روی دستگاه خود، اشکال را بیابید.
- خطای Time Limit Exceeded – اجرای هر کد، باید در زمانی که بالای سوال مشخص شده، به اتمام برسد؛ در غیر این صورت، لازم است کد خود را بهینه کنید تا در زمان کوتاه‌تری اجرا شود. در شماره‌های آینده‌ی کدنامه، با تکنیک‌هایی برای رفع این خطا آشنا خواهید شد.
- خطای Memory Limit Exceeded – اجرای هر کد، باید حافظه‌ای کم‌تر از محدودیت اعلام شده در بالای سوال را اشغال کند. برای حل این مشکل، لازم است اندازه‌ی «آرایه»های کد – که بعدا با آن‌ها آشنا می‌شوید – را کم‌تر کنید.
- خطای Runtime Error – سایر خطاهای ممکن، هم‌چون تقسیم عدد بر صفر و یا – همان‌گونه که بعدا خواهید خواند – دسترسی به عناصری از آرایه که موجود نیستند، که در حالت عادی باعث crash کردن برنامه‌ی C شما و خروج ناگهانی از آن می‌شوند، در این دسته جای می‌گیرند.