برنامه‌نویسی با هَسکِل

از مبانی اولیه

کریستوفر اَلِن، جولی مورونوکی
ترجمه‌ی کیان مسکوت

برنامه‌نویسی تابعی خالص، برای تازه‌کارها و باتجربه‌ها

نسخه‌ی نمونه

برنامه‌نویسی با هسکل

از مبانی اولیه

کریستوفر اَلِن، جولی مورونوکی

ترجمه‌ی کیان مسکوت

(نسخه‌ی نمونه)

برنامه‌نویسی با هسکل

از مبانی اولیه

www.haskellbook.ir

سرشناسه	:	اَلِن، کریستوفر Allen, Christopher
عنوان و نام پدیدآور	:	برنامه‌نویسی با هسکل، از مبانی اولیه / کریستوفر الن، جولی مورونوکی
مشخصات نشر	:	تهران: کیان مسکوت، ۱۳۹۷ -
مشخصات ظاهری	:	۲ ج.: جدول.
شابک (دوره)	:	۹۷۸-۶۲۲-۰۰-۱۳۱۸-۱
شابک (ج. ۱)	:	۹۷۸-۶۲۲-۰۰-۱۳۱۶-۷
شابک (ج. ۲)	:	۹۷۸-۶۲۲-۰۰-۱۳۱۷-۴
وضعیت فهرست‌نویسی	:	فیپا
یادداشت (عنوان اصلی)	:	Haskell Programming from First Principles: Pure Functional Programming without Fear or Frustration [۲۰].
موضوع	:	هسکل (زبان برنامه‌نویسی کامپیوتر)
موضوع	:	Haskell (Computer Program Language)
موضوع	:	برنامه‌نویسی تابعی (کامپیوتر)
موضوع	:	Functional Programming (Computer Science)
شناسه‌ی افزوده	:	مورونوکی، جولی
شناسه‌ی افزوده	:	Moronuki, Julie
شناسه‌ی افزوده	:	مسکوت، کیان، ۱۳۶۸ - ، مترجم
رده‌بندی کنگره	:	QA ۷۳/۷۶/۷آ۵ه ۱۳۹۷
رده‌بندی دیوئی	:	۰۰۵/۱۳۳
شماره‌ی کتابشناسی ملی	:	۵۳۶۸۱۷۹

فهرست،
پیش گفتار
و معرفی

فهرست مطالب

پیش‌گفتار مترجم

دلیل اینکه پیش‌گفتارِ مترجم رو قبل از نویسنده آوردم، اینه که فکر می‌کنم بهتر باشه اول از همه راجع به نگارشِ عامیانه‌ی این کتاب صحبت کنم. البته احتمالاً این نوشتار خیلی براتون عجیب و نامتعارف نباشه، با این حال چنین سبکی هنوز بین کتاب‌های علمی/فنی رایج نیست.

همین نوشتارِ عامیانه باعث شده بعضی جاها در نگاهِ اول گنگ و مبهم به نظر برسن. یه قاعده‌ی کلی که در این مواقع کمک‌تون می‌کنه، اینه که یادتون باشه تا جایی که میشه عامیانه بخونین! مثلاً توی همین جمله‌ی قبل، منظور از "کمک‌تون می‌کنه" همون "به شما کمک می‌کند" میشه.[1]

---

[1] اینجا میشد "کمک‌تون" رو سَرِهم هم نوشت، یعنی: کمکتون. ولی این نگارش، تشخیصِ لغتِ "کمک" رو یه کم سخت می‌کنه و به همین خاطر جدا از هم نوشته شدن. بطورِ مشابه، در طولِ کتاب بجای عبارت‌هایی مثلِ "هسکل هست،" از "هسکله" استفاده شده ("هـ" جدا در انتها، بجای "هست" – اینطوری "جریان" یا "flow" خوندن حفظ میشه و خواننده مجبور نیست دائماً بین نگارش محاوره‌ای و کتابی رفت‌وبرگشت کنه). کلاً آدمها نوشته‌ها رو اکثراً لغت به لغت

موردِ دیگه‌ای که هست، اجتناب‌ناپذیریِ تلفیقِ لغات انگلیسی در متنِ این کتابه (و یاد گرفتنِ اون لغات به شدت توصیه میشه!). برای مثال این جمله رو در نظر بگیرین:

اسمِ رایج‌ترین محیطِ تعاملی یا interactive ِهسکل، GHCi ِه.

اینجا ارتباط لغتِ interactive با لغت هسکل، به واسطه‌ی یه کسره مشخص شده. آخرِ جمله هم بجای "می‌باشد" در حالتِ معادلِش (یعنی "GHCi می‌باشد")، از ترکیبِ کسره و "هـ" استفاده شده.

یکی دیگه از عواقبِ ترکیب لغات انگلیسی (یا هر زبان دیگه‌ای که از چپ به راست نوشته میشه) در متون فارسی زمانی پیش میاد که یه عبارتی که از چند کلمه درست شده آخرِ خط میاد، طوری که لغاتِ اون عبارت در دو خط نوشته میشن. خیلی سعی شده که از چنین چیزی توی این کتاب پرهیز بشه، اما اگر موردی دیدین (توی این کتاب یا هرجای دیگه)، یادتون باشه اول هرچیز توی خط اول نوشته شده رو از

می‌خونن، فقط لغات نا آشنا رو حرف به حرف می‌خونیم. جدا کردنِ این طور پسوندها کمک می‌کنه ظاهرِ لغاتِ آشنا حفظ بشه.

چپ به راست بخونین، بعد برین خط بعد و دوباره از چپ به راست بخونین. مثلاً این رو فرض کنین:

لغت کلـمه لـغت کلمه لغت کلمه لغت کلمه لغت Here’s a sample sentence کلمه لغت کلمه لغت کلمه.

اون جمله‌ی انگلیسی اینه: Here’s a sample sentence.

نکته‌ی دیگه در موردِ فونتِ[1] نوشته‌هاست. برایِ کدها، اسمِ فایل‌ها، کتابخونه‌ها، تابع‌ها، لینک‌ها و غیره از فونتِ عرض‌ثابت[2] استفاده شده. گاهی پیش اومده که یه لغت با دو فونت نوشته شده باشه؛ مثلاً عملِ "map کردن" (به معنای "نگاشت کردن") با فونت معمولی، ولی *تابعِ* `map` با فونتِ عرض‌ثابت نوشته شده.

سعی شده خطایی توی کتاب نباشه... ولی اگه ایرادی دیدین، می‌تونین از طریق سایت کتاب (www.haskellbook.ir)، یا آدرس ایمیلِ

---

[1] قلم یا font. تلفظِ صحیح‌ترِش "فانت" ِه.

[2] Monospaced Font

<**آدرسِ سایت کتاب**>**@errata** گزارش بدین تا در نسخه‌های بعدی اصلاح بشه.

و نکته‌ی آخر اینکه هر جا لازم بوده، یه سری توضیحاتی اضافه کردم تا به انتقالِ مطلب کمک کنن. تقریباً همه‌ی این توضیحات با یه "م." (خلاصه‌ی "مترجم") شروع شدن.

در کل، مطالبِ این کتاب سختی‌های خودش رو داره، و واقعاً امیدوارم که این سبکِ نگارشی به انتقالِ بهتر و بهینه‌ترِ مطالب به شما خواننده‌ی محترم کمک کنه!

پیش‌گفتار نویسنده‌ها

داستان کریس

من بیشتر از ۱۵ ساله که برنامه‌نویسی می‌کنم، که ۸ سالِش بصورتِ حرفه‌ای بوده. بیشتر با لیسپِ معمولی[۱]، کلوژِر[۲]، و پایتون[۳] کار کردم. تقریباً ۶ سالِ پیش هسکل[۴] نظرم رو جلب کرد. هسکل زبانی بود که بهم ثابت کرد تحقیقات در زبان‌های برنامه‌نویسی داره نتیجه میده و پیشرفت می‌کنه، و اینکه استفاده از زبانی که بر پایه‌ی اون پیشرفت‌ها طراحی شده واقعاً پرفایده‌ست.

من خطاهایی در کلوژر داشتم که خودم و چندتا کلوژِر نویسِ حرفه‌ای زودتر از ۲ ساعت نمی‌تونستیم حل‌شون کنیم؛ اون هم به خاطرِ تایپ سیستمِ دینامیک[۵]ش بود. یه عالَم تست می‌کردیم. هر جا میشد

---

[۱] Common Lisp

[۲] Clojure

[۳] Python

[۴] Haskell

[۵] Dynamic Type System

printlns اضافه می‌کردیم. از توی REPL تک‌به‌تک تابع‌ها رو تست می‌کردیم. باز هم خیلی طول می‌کشید. تازه فقط ۲۵۰ خط کدِ کلوژر بود. بالاخره بعد از اینکه درست‌ش کردم، فهمیدم مشکل از وجودِ IFn توی بردارهای کلوژر بود. همین باعث شده بود داده‌های بدشکلی از سرمنشاء مشکل به جاهای دیگه‌ای از برنامه پخش بشن. با پایتون و لیسپِ معمولی هم مشکلاتِ مشابهِ این رو داشتم. چنین چیزی با هسکل در طولِ یک دقیقه یا کمتر حل میشه، چون تایپ سیستمِ‌ش سریع و دقیق میگه کجا رو ناهماهنگ با بقیه‌ی کد نوشتین.

دلیلی که از هسکل استفاده می‌کنم اینه که می‌خوام بدونِ ترس کُدَم رو بازسازی[۱] کنم، چون نمی‌خوام از نگهداریِ کد[۲] بَدَم بیاد، تا با خیالِ راحت از کدی که نوشتم باز هم استفاده کنم. چنین مزایایی رو نمیشه بدونِ یاد گرفتنِ یه سری چیزهای جدید بدست آورد. فرقِ بین کسانی که "ریاضی‌شون خوبه" و "ذهنی" حل می‌کنن با ریاضیدان‌های حرفه‌ای اینه که گروهِ دوم کارِشون رو نشون میدن و از ابزارهایی استفاده

---

[۱] Refactor

[۲] Code Maintenance

می‌کنن که در حلِ مسئله کمک‌شون می‌کنه. وقتی با زبانی که تایپ سیستمِ دینامیک داره کار می‌کنین، خودتون رو مجبور می‌کنین که همه کار رو "ذهنی" انجام بدین، که اصلاً لزومی نداره. آدم‌ها حافظه‌ی کاریِ محدودی دارن، من هم ترجیح میدم از هر ابزاری که کمک می‌کنه سریع‌تر استدلال کنم و کدِ بهتری بنویسم استفاده کنم.

هسکل زبانِ سختی برای استفاده نیست، بلکه کاملاً برعکس‌ش. من الان مسائلی رو می‌تونم حل کنم که با کلوژر، لیسپِ معمولی، یا پایتون نمی‌تونستم. تدریسِ خوبِ هسکل کارِ سختیه، و تدریسِ بد هم یاد گرفتن‌ش رو برای خیلی‌ها سخت کرده.

ولی نباید اینطوری باشه.

من دو سالِ گذشته رو به تدریسِ هسکل بصورتِ آنلاین و حضوری گذروندم. در طولِ راه به تمرین‌ها و روش‌های تدریسی که مفاهیم مختلف رو بهتر منتقل می‌کردن دقت کردم و نکته‌برداری کردم. نهایتاً اون نکته‌ها راهنمای من برای یادگیریِ هسکل شده بودن. من هنوز دارم

از طریق تعاملِ حضوری با آدم‌ها در شهرِ آستینِ تکزاس[۱]، آنلاین، و کانالِ IRC که برای تازه‌کارها درست کردم تا در یادگیریِ هسکل کمک‌شون کنه، تدریسِ بهترِ هسکل رو یاد می‌گیرم.

این کتاب رو نوشتم چون نمی‌خوام بقیه هم مثل من تقلا کنن.

داستان جولی

من بهارِ ۲۰۱۴ روی توئیتر[۲] با کریس آشنا شدم. هرکسی که کریس رو می‌شناسه، میدونه که خیلی طول نمی‌کشه آدم رو ترغیب به یاد گرفتنِ هسکل کنه.

بهش گفتم علاقه‌ای به برنامه‌نویسی ندارم. گفتم هیچ چیز و هیچ کسی تا حالا نتونسته من رو مجذوب به برنامه‌نویسی کنه. وقتی فهمید پیش‌زمینه‌ی زبان‌شناسی دارم، فکر کرد شاید به خاطرِ پردازشِ زبانِ طبیعی[۳] هم که شده هسکل یاد بگیرم. باز هم قانع نشده بودم.

---

[۱] Austin, Texas

[۲] Twitter

[۳] Natural Language Processing

از یه راهِ دیگه وارد شد. حسابی داشت روی مطالبی که برای تدریسِ هسکل داشت وقت میذاشت و می‌خواست اونها رو ارزیابی، و نوشته‌هاش رو بهتر کنه. من رو قانع کرد که سعی کنم هسکل رو یاد بگیرم تا تدریس به کسی که چیزی از کدنویسی نمی‌دونه رو تجربه کنه. من هم با نگرشِ "هر چیزی واسه علم" قبول کردم.

کریس خودش می‌دونست که آموزش‌های هسکلی که موجود بودن، هر کدوم یه مشکلاتی داشتن، ولی فکر نکنم واقعاً درک کرده بود که چقدر برای من غیرقابل تحمل بودن. هر کدوم رو که می‌دیدم به یه پیش‌زمینه‌ای از بقیه‌ی زبان‌های برنامه‌نویسی اتکا داشتن و خیلی از لغات رو تعریف نمی‌کردن، یا خیلی از امکاناتِ هسکل رو با قیاس و آنالوژی (اون هم نه به خوبی) با امکاناتِ بقیه‌ی زبان‌ها توضیح می‌دادن. من هم اصلاً اون امکانات رو نمی‌شناختم.

وقتی گفتم هیچ تجربه‌ی برنامه‌نویسی ندارم، واقعاً جدی گفتم. باید با یادگیریِ اینکه کامپایلر[1] کارش چیه، معنیِ کنترلِ نسخه[2] چیه، چه

---

[1] Compiler

[2] Version Control

چیزهایی شاملِ عوارض[۱] میشن، کتابخونه[۲] چیه، ماژول[۳] چیه، یا اصلاً سرریزِ پشته[۴] چیه. این سؤال‌ها رو یک سال پیش (از الان که دارم این رو می‌نویسم) داشتم؛ این کتاب رو که تموم کنیم و منتشر شه، یه کم بیشتر از دو سال میشه.

نهایتاً وقتی کریس متوجه شد که یه نوع کتابِ جدیدی برای یادگیریِ هسکل لازمه، تصمیم گرفت خودش یکی بنویسه. اون موقع قبول کردم بشم موشِ آزمایشگاهی‌ش. اون فصل‌ها رو برام می‌فرستاد و من هم از روشون هسکل یاد می‌گرفتم و بهش بازخورد میدادم. در طولِ پاییز همینطوری "گَهی تند و گهی خسته" پیش رفتیم. بالاخره به این نتیجه رسیدیم که اگه من هم یه کم از مسئولیتِ نویسندگی رو به عهده بگیرم خیلی بهتر پیش میریم. یه فرایندِ نوشتاری درست کردیم که کریس یه فصل رو می‌نوشت و مشخص می‌کرد چه مطالبی باید بیان بشن. بعد من بخشهایی که می‌فهمیدم رو مشخص می‌کردم و برای

---

[۱] Side-Effects

[۲] Library

[۳] Module

[۴] Stack Overflow

بخش‌هایی که از قبل بلد نبودم یا برام واضح نبودن سؤال‌هایی رو مطرح می‌کردم. انقدر به سؤال‌هام جواب میداد تا می‌فهمیدم، و من هم بعضی جاها رو اصلاح و مطالبی اضافه می‌کردم. هردومون تمرین‌ها رو طراحی می‌کردیم – تمرین‌هایی که من می‌نوشتم خیلی ساده‌تر از تمرین‌های کریس بودن، ولی همین تنوع چیزِ خوبی بود.

در طولِ این فرایند، همه‌ش سعی کردم از دیدِ کسی که کاملاً تازه‌کاره نگاه کنم. یکی از دلایلش این بود که می‌خواستم درکی که از هسکل داشتم رو عمیق‌تر کنم، به همین خاطر دائماً مطالبی که فکر می‌کردم بلدم رو به چالش می‌کشیدم. دلیلِ دیگه‌ش هم این بود که می‌خواستم این کتاب برای همه مناسب باشه.

از صحبت‌هایی که با بقیه‌ی کسانی که در حالِ یاد گرفتنِ هسکل بودن متوجه شدم، اونها هم به خاطر بقیه‌ی آموزش‌های هسکل که خونده بودن، قانع شده بودن که هسکل سخت و جادویی‌ه، و بهتره فقط جادوگرها باهاش کار کنن!

اصلاً نباید اینطور باشه.

معرفی

به یه راهِ جدید برای یادگیریِ هسکل خوش اومدین. ممکنه خسته از تلاش‌های قبلی برای یادگیریِ هسکل به این کتاب اومده باشین. یا شاید هم کلاً چیز زیادی از هسکل نمی‌دونین. شاید هم قانع نشدین که ممکنه چیزی بهتر از لیسپِ معمولی/اسکالا[۱]/روبی[۲]/یا هر زبان دیگه‌ای که دوست دارین وجود داشته باشه، و می‌خواین با ما بحث کنین! یا شاید هم دنبالِ ۱۸ میلیاردُمین (نکته: حدودی میگیم) آموزشِ مونَد[۳] می‌گشتین به امید اینکه این دفعه دیگه بالاخره موندها رو یاد بگیرین. از هر جایی که اومدین، خوش اومدین! هدف ما اینه که هسکل رو تا جای ممکن واضح، بی‌دردسر، و کاربردی معرفی کنیم، اون هم برای هر مخاطبی.

---

[۱] Scala

[۲] Ruby

[۳] Monad

چرا این کتاب

اگه تازه می‌خواین برنامه‌نویسی رو شروع کنین، هسکل گزینه‌ی خیلی خوبیه. هسکل یه زبانِ برنامه‌نویسیِ تابعی[۱] ِه، و کاربردِ عمومی داره. هرجایی که بشه با نوشتنِ برنامه‌ای مسئله‌ای رو حل کرد، میشه از هسکل هم استفاده کرد (به غیر از بعضی برنامه‌های جاسازی‌شده‌ی[۲] بخصوص).

اگر هم برنامه‌نویس هستین، ممکنه دلایلِ متنوعی برای تقویتِ مهارت‌هاتون از طریقِ یادگیریِ هسکل داشته باشین: از عشق به برنامه‌نویسیِ تابعیِ خالص گرفته تا انگیزه برای کدنویسیِ تابعی در اِسکالا، تا پیدا کردنِ راهی به سمتِ پیوراسکریپت[۳] یا اِدریس[۴]. زبان‌هایی

---

[۱] برنامه‌نویسیِ تابعی یه سبکی از برنامه‌نویسیه که ساختارِ اصلیِ برنامه‌ها متشکل از تابع‌ها و فراخوانِ اونهاست (بجای یه سری از دستورات که کامپیوتر اجرا کنه). الان ماهیت‌ش خیلی مهم نیست؛ هسکل کاملاً سبکِ تابعی رو در بر داره، و در طولِ کتاب واضح و روشن میشه.

[۲] Embedded Applications

[۳] PureScript

[۴] Idris

مثلِ جاوا[۱] دارن خورده‌خورده بعضی مفاهیمِ تابعی رو به کار می‌گیرن، اما اصلاً به عنوانِ یه زبانِ تابعی طراحی نشدن. از اونجا که هسکل یه زبانِ تابعیِ خالصه، محیطِ "حاصلخیزی" برای یادگیری و تسلط به برنامه‌نویسیِ تابعی به حساب میاد. مهم نیست توی چه زبانی کد می‌نویسین، نحوه‌ی فکر کردن و حلِ مسئله که با برنامه‌نویسیِ تابعی یاد می‌گیرین خیلی مفیده. ما از بعضی خواننده‌ها شنیدیم که این کتاب برای کارهاشون در زبان‌های متنوعی مثلِ اسکالا، اف‌شارپ[۲]، فرِگه[۳]، سوئیفت[۴]، پیوراسکریپت، اِدریس، و اِلم[۵] مفید بوده.

اولِ کار، ممکنه نوشتنِ هسکل به نظر سخت‌تر بیاد، نه فقط به خاطرِ گرامر و مفاهیمی که با اکثرِ زبان‌ها فرق داره، بلکه به خاطرِ امکاناتی مثل تایپ‌دار بودنِ قوی‌ش که یک سری دیسیپلین در نحوه‌ی کدنویسی رو اجبار می‌کنه. اما این عیب نیست، مزیتِ ه. هر چقدر هم

---

[۱] Java

[۲] F#

[۳] Frege

[۴] Swift

[۵] Elm

که دوست نداشته باشیم اقرار کنیم، ما آدم‌ها تمرکز و حافظه‌ی کوتاه-مدتِ محدودی داریم. نمیشه همه‌ی فراداده[1]های مرتبط با برنامه‌هامون رو توی ذهن‌مون نگه داریم. اگه حافظه‌ی محدودِمون رو روی کارهایی که کامپیوتر می‌تونه برامون انجام بده صرف کنیم، اصلاً سازنده عمل نکردیم. کامپیوترها در حفظِ داده‌ها خیلی خوب عمل می‌کنن، بخصوص در حفظِ فراداده‌هایی مثلِ تایپ‌ها.

دلیلِ استفاده‌ی ما از هسکل، نخبه بودن نیست – دلیلِ استفاده از ابزاری مثل هسکل اینه که کمک‌مون می‌کنن. چنین ابزارهای خوبی مثل هسکل، باعث میشن سریع‌تر کار کنیم، اشتباهاتِ کمتری داشته باشیم، و با خوندنِ کُدِمون، هدفِ کد رو بهتر و کامل‌تر درک کنیم.

هسکل (در بلند مدت) آسون‌تره، و باعث میشه کارِ بهتری انجام بدیم. این فایده‌شه. شروع با هسکل یه کم زحمت داره، اما با کمی حوصله، و انگیزه برای حلِ تمرین‌ها، راحت میگذره.

---

[1] Metadata

قبوله، ولی من فقط یه آموزشِ موند می‌خواستم...

خبر بد اینکه همین دنبالِ راهِ آسون گشتن برای یاد گرفتنِ هسکل (و برنامه‌نویسیِ تابعی) باعث میشه به نظر سخت برسه. خبر خوب اینکه ما خیلی تجربه‌ی تدریس داریم و نمی‌خوایم چنین بلایی سرِ هیچ کسی بیاد، بخصوص شما خواننده‌ی محترم.

پیشنهاد می‌کنیم اگه چیزی از برنامه‌نویسی می‌دونین فراموش کنین و این کتاب رو با ذهنی مبتدی بخونین. خودتون رو بسپرین به دریای تایپ‌ها.

اگه برنامه‌نویسِ باتجربه‌ای هستین، یاد گرفتنِ هسکل مثلِ این می‌مونه که بخواین از اول برنامه‌نویسی یاد بگیرین. یادگیریِ هسکل، راه‌های جدیدی برای فکر کردن در موردِ ساختاردهیِ برنامه‌ها تحمیل می‌کنه، که این راه‌ها نسبت به روش‌هایی که اکثر مردم در زبان‌های دستوری[۱] یا بی‌تایپ[۲] ازشون استفاده می‌کنن و باهاشون راحتن خیلی

---

[۱] Imperative

[۲] Untyped

متفاوت‌اند. اکثرِ کسانی که حداقل یکی دوتا زبان برنامه‌نویسی یاد گرفتن، به پیش‌وپاافتاده بودنِ این فرایند عادت کردن، و همین انتظارات‌شون باعث میشه از هسکل زده بشن.

اگه هسکل اولین زبانِ‌تونه (یا حتی اگه اولی‌ش نیست)، شاید متوجهِ مشکلی که خیلی از آموزش‌های هسکل دارن شده باشین: فرض می‌کنن که یه پیشینه‌ای در برنامه‌نویسی دارین و متعاقباً اکثرِ مفاهیمِ هسکل رو با مقایسه‌ی مفاهیمِ برنامه‌نویسی در زبان‌های دیگه توضیح میدن؛ ولی ما ادعا می‌کنیم این روش به برنامه‌نویس‌های با تجربه هم هیچ کمکی نمی‌کنه. بیشتر وقت‌ها، مقایسه‌ی هسکل با یه زبانِ دیگه فقط منجر به یه درکِ سطحی از هسکل میشه، و قیاس و آنالوژی با loop ها و ساختارهای مشابه، باعثِ درکِ اشتباه از طرزِ کارِ کدِ هسکل میشه. به خاطرِ همه‌ی این دلایل، ما سعی کردیم اصلاً به دانشِ پیشین از زبان‌های برنامه‌نویسیِ دیگه اتکا نکنیم. دقیقاً مثلِ یاد گرفتنِ یه زبانِ دیگه غیر از زبانِ مادری می‌مونه، تا وقتی همه‌ش معنیِ لغت‌به‌لغت کلمات رو یاد بگیرین، خیلی روان و با درکِ بالا نمی‌تونین از اون زبان

استفاده کنین. بهترین راهِ یاد گرفتنِ هسکل هم اینه که بر مبنای خودِش یاد بگیرین.

هسکل سخته؟

توی اینترنت شایعه شده که آدم باید یه دکترای ریاضی، و دَرکِ کاملی از مونَدها داشته باشه تا بتونه تو هسکل "hello, world"[1] بنویسه.

ما "hello, world" رو تو فصلِ ۳ می‌نویسیم. قبل‌ش یه کم حساب کار می‌کنیم تا گرامرِ تابع‌ها و اعمالِ اونها براتون عادی بشه. ولی لازم نیست دکترای مونَدولوژی (!) داشته باشین.

در حقیقت در ذاتِ برنامه‌ی "hello, world" که می‌نویسیم، موند به کار رفته، و وقتی به آخرِ کتاب برسین، حتماً موندها رو درک می‌کنین؛ اما خیلی قبل‌تر از درکِ طرزِ کارِشون با موندها کار می‌کنین.

---

[1] نوشتنِ "hello, world" (به معنای "سلام، دنیا") در یه زبانِ برنامه‌نویسیِ جدید، مثلِ "اولین برنامه‌ی بچه" می‌مونه. حالا منظور از این شایعه اینه که وقتی نوشتنِ برنامه‌ای به این سادگی انقدر سخته، پس حتماً کار با خودِ زبان محاله. زبان‌هایی هستن که عمداً نوشتنِ چنین برنامه‌هایی رو سخت کردن، ولی هسکل جزئشون نیست.

بعضی اوقات می‌بینین که این کتاب بیشتر از حدِ لازم برای هسکل‌نویسیِ موفق واردِ جزئیات میشه. هیچ اشکالی نداره. لازم نیست بارِ اول همه چیز رو تمام و کمال درک کنین. در کل، برگشتن به مطالبی که یک بار خوندین تا بیشتر بررسی‌شون کنین و یه بار دیگه بخونین، رَوِش بهینه‌ای برای یادگیریِ ه.

کلامی چند به برنامه‌نویس‌های جدید

ما واقعاً تلاش کردیم این کتاب رو تا جای ممکن برای هرکسی (با هر علمِ پیشینی) در دسترس قرار بدیم. مقایسه و اشاره به زبان‌های دیگه رو به حداقل رسوندیم، و قول میدیم اگه جایی چیزی از هسکل رو با یه چیزی از یه زبانِ دیگه مقایسه کردیم، خودِ اون مقایسه برای درکِ هسکل مهم نیست – فقط یه کم مخلفات برای کَسانیه که اون یکی زبان رو می‌شناسن.

با همه‌ی اینها، هر چقدر که در کتاب پیش میریم و تمرین‌ها و پروژه‌ها "واقعی‌تر" میشن، به ناچار لغات و مفاهیمی به کار میرن که توی این کتاب جایی برای توضیح‌شون نیست، ولی برنامه‌نویس‌ها

باهاشون آشنا اند. مثلاً شاید لازم باشه JSON رو خودتون از روی اینترنت یاد بگیرین (م. برای اکثر اینها مطالب فارسی هم پیدا میشه). بخشِ بعدیِ این معرفی به لغاتی اشاره داره که شاید نشناسین، ولی برنامه‌نویس‌ها بلدن – نگران نشین. ما فکر می‌کنیم شما هم از خوندن‌شون چیزهایی یاد می‌گیرین، ولی حتی اگر هم چیزی متوجه نشدین، هیچ اشکالی نداره. این که از قبل همه‌ی لغت‌های این کتاب رو بلد نیستین، نشون از این نیست که نمی‌تونین هسکل یاد بگیرین: فقط نشون از اینه که *هنوز همه چیز رو بلد نیستین*، و در واقع همه همینطوراند، پس مشکلی نیست.

در طرفِ مقابل، این کتاب چیز زیادی از نحوه‌ی کار با ترمینال[۱] و برنامه‌های ویرایش متن[۲] نمیگه. فرض بر اینه که تا حدی با ترمینال آشنایی دارین که کارهای ساده‌ای مثلِ ساختِ پوشه یا باز کردنِ یه

[۱] Terminal

[۲] Text Editor

فایل رو بلدین. و با توجه به تنوعِ text editor ها، برای هیچ کدوم آموزشی نذاشتیم.[۱]

اگه کمک لازم داشتین یا دوست داشتین با برنامه‌نویس‌های تابعیِ دیگه آشنا بشین، راه‌های زیادی وجود داره. کانالِ IRC روی Freenode به اسمِ `#haskell-beginners` معلم‌هایی داره که خوشحال میشن کمک‌تون کنن، و آغوشِ بازی برای مسائلِ بخصوصی که می‌خواین حل کنین دارن.[۲] کانال‌های اِسلَک[۳] و ساب‌رِدیت[۴]هایی هم هستن که

---

[۱] اگه خیلی تازه‌کارین و چیزی در موردِ برنامه‌های ویرایشِ متن نمی‌دونین، ما Atom رو پیشنهاد می‌کنیم. رایگان و متن‌باز (open-source) و قابلِ تنظیمِ ه. در طولِ نوشتن این کتاب، جولی از Sublime Text استفاده کرد و راضی بود، ولی مجانی نیست. کریس بیشترِ اوقات از Emacs استفاده می‌کنه؛ بین برنامه‌نویس‌ها برنامه‌ی خیلی محبوبیه، ولی یادگیریش یه کم زمان می‌بره. Vim هم یه برنامه‌ی دیگه‌ست که اون هم باز یادگیریش کمی زمان‌بره. اگه هیچ تجربه‌ای با Emacs یا Vim ندارین، همون Sublime یا Atom گزینه‌های خوبی‌اند.

[۲] Freenode IRC (مخففِ Internet Relay Chat)، شبکه‌ای از کانال‌های مکالمه‌ی متنیِ ه. شبکه‌های IRC دیگه، یا حتی پلتفورم‌های چتِ دیگه‌ای هم وجود دارن، ولی کانال‌های IRC هسکل روی Freenode مکان‌های محبوبی برای ملاقاتِ هسکل‌نویس‌ها شدن. اگه به آشنایی با زیستگاهِ هسکل‌نویس‌ها علاقه دارین، راه‌های زیادی برای دسترسی به Freenode IRC وجود داره، مثلِ Irssi و HexChat.

[۳] Slack

[۴] Sub-reddit

هسکل‌نویس‌ها اونجا جمع میشن. بلاگ‌های هسکلیِ خیلی زیادی هم وجود دارن، که خیلی‌هاشون رو در پانویس‌ها و منابعِ پیشنهادیِ آخر فصل‌ها معرفی کردیم. خیلی از خواننده‌های این کتاب در سوئیفت و اِسکالا برنامه می‌نویسن، شاید بد نباشه به جامعه‌های اونها هم یه سَری بزنین.

تبلیغ هسکل

مابقیِ این معرفی کمی از تاریخِ هسکل صحبت می‌کنه و به بقیه‌ی زبان‌های برنامه‌نویسی و سبک‌های مختلف ارجاع داره. اگه تازه‌کارین، ممکنه همه‌ی اینها براتون مفهوم نباشن، ولی اشکالی نداره. بقیه‌ی کتاب با در نظر داشتنِ تازه‌کارها نوشته شده. با پیشروی در کتاب، این قابلیت‌هایی که اینجا اشاره می‌کنیم رو متوجه میشین.

یه ذره هسکل رو با بقیه‌ی زبان‌ها مقایسه می‌کنیم تا نشون بدیم چرا به نظرِ ما استفاده از هسکل با ارزشه. هسکل پیشرفتی از زبان‌هایی

مثلِ ML ِه (که توسطِ رابین میلنر[۱] و دیگران از دانشگاهِ ادینبورگ[۲] در سالِ ۱۹۷۳ اختراع شد). ML خودش از روی ISWIM درست شده بود، که اون هم از روی ALGOL60 و Lisp. این تاریخچه رو گفتیم تا بدونین هسکل جدید نیست. محبوب‌ترین پیاده‌سازیِ[۳] هسکل، به اسمِ کامپایلرِ گلاسگو برای هسکل[۴] (GHC)، خیلی جا افتاده و خوش‌ساخته. قابلیت‌هایی که در هسکل کنار هم جمع شدن، باعث شده تبدیل به زبانی بشه که خیلی گویاتر از روبی باشه، و امنیتِ تایپی[۵]ش بیشتر از هر زبانِ تجاریِ دیگه‌ای بشه.

در سال ۱۹۶۸، نسخه[۶]ی ALGOL68 قابلیت‌های زیر رو داشت:

۱. رکورد تایپ[۷]هایی که کاربر می‌تونست تعریف کنه.

---

[۱] Robin Milner

[۲] University of Edinburgh

[۳] Implementation

[۴] Glasgow Haskell Compiler

[۵] Type Safety

[۶] Dialect

[۷] Record Type

۲. تایپ‌های جمعی[۱] که کاربر می‌تونست تعریف کنه (اتحادهایی که محدود به شمارش‌های ساده نبودن).

۳. بیانیه‌های سوئیچ[۲] یا case که با تایپ‌های جمع کار می‌کردن.

۴. مقادیرِ ثابتی که در زمانِ کامپایل اجبار، و بجای =: با استفاده از = تعریف می‌شدن.

۵. گرامرِ یکپارچه برای استفاده از مقادیر و تایپ‌های مرجع – اشاره‌ی اشاره‌گر[۳] دستی نبود.

۶. بستارها[۴] با گستره‌ی واژگانی[۵] (بدون این قابلیت، خیلی از الگوهای تابعی از هم می‌پاشن).

۷. اجرای موازیِ دستورالعمل‌ها، مستقل از پیاده‌سازیِ زبان.

---

[۱] Sum Types

[۲] Switch

[۳] Pointer Dereferencing

[۴] Closures

[۵] Lexical Scoping

۸. کامپایلِ چندگذری[۱] – یعنی میشه بعد از اینکه از چیزی استفاده کردین تعریف‌ش کنین.

تا اولایلِ قرنِ ۲۱، خیلی از زبان‌های محبوبی که کاربریِ عمومی و تجاری دارن در خیلی قابلیت‌ها هنوز به پایِ ALGOL68 نمیرسن. دلیل اینکه چنین چیزی میگیم، اینه که ایمان داریم امکانِ پیشرفت در علومِ کامپیوتر، برنامه‌نویسی و زبان‌های برنامه‌نویسی، نه تنها ممکنه، بلکه برای تبدیلِ نرم‌افزار به یه رشته‌ی مهندسیِ واقعی، واجب و مطلوبه. درسته که رشته‌ی مهندسیِ نرم‌افزار وجود داره، ولی معنای واقعیِ مهندسی، به کار گرفتنِ اطلاعاتِ علمی و عملی در کنارِ همدیگه، برای ساخت و نگهداریِ سیستم‌های بهتره. با پیشرفتِ علم و تغییر در موادِ موجود، مهندس‌ها هم باید پیشرفت کنن.

هسکل از پیشرفت‌هایی که از ALGOL68 تا به حال در زبان‌های برنامه‌نویسیِ محبوب اختراع شدن استفاده می‌کنه، در کنارشِ از یه پیاده‌سازیِ جا افتاده و طراحیِ خوب هم بهره می‌بره. هسکل گاهی

---

[۱] Multi-Pass

اوقات به عنوانِ یه زبانِ "آکادمیک" کنار زده میشه، دلیل‌ش هم اینه که نسبتاً هم‌قدم با علمِ روزِ ریاضیات و کامپیوتر پیش رفته. به نظر ما، این پیشرفت خوبه و کمک به حلِ مسائلِ عملی در رایانش و طراحیِ نرم‌افزار می‌کنه.

پیشرفت، هم مطلوب و هم ممکنه، ولی یکنواخت و اجتناب‌ناپذیر نیست. تاریخِ دنیا پر از امثالِ پیشرفت‌های نامنظمِ ه. برای مثال، تخمین زده شده که بین سال‌های ۱۵۰۰ تا ۱۸۰۰ میلادی، بیماریِ اسکوربوت[۱] دو میلیون دریانورد رو از پا درآورده. دنیای غرب، بارها درمانِ اسکوربوت رو فراموش کرده. در سالِ ۱۶۱۴، جراحِ کلِ گروهانِ هندِ شرقی، توصیه کرد که برای جلوگیری از اسکوربوت، همه‌ی سیاحت‌های دریایی با خودشون مرکبات هم ببرن. جونِ خیلی‌ها رو نجات داد، ولی درکی که از *دلیلِ* درمانِ اسکوربوت با مرکبات داشتن اشتباه بود. همین مسئله منجر به استفاده از لیمو شیرازی[۲] که ویتامین C ِکمتری نسبت به

---

[۱] Scurvy

[۲] Lime

لیموترش[1] داره شد، و اسکوربوت برگشت، تا اینکه در سال ۱۹۳۲ اسید اسکوربیک کشف شد. بی‌نظمی و لجبازی (با اینکه دریانوردها همینطور از اسکوربوت می‌مردن، ارتشِ دریاییِ بریتانیا باز هم لیمو شیرازی رو تأیید می‌کرد) می‌تونه جلوی پیشرفت رو بگیره. ما ترجیح میدیم دکتری داشته باشیم که مایل به قبول و درکِ اشتباهات‌ش باشه، با علمِ جدید هم‌پا باشه، و حتی فعالانه به فکرِ ارتقا و گسترشِ دانسته‌هاش باشه، تا اینکه دکتری داشته باشیم که همه‌ی کارهاش رو برپایه‌ی فرضیه‌های سطحی انجام میده.

راه‌های دیگه‌ای هم برای جلوگیری از اسکوربوت وجود داره، همونطور که زبان‌های برنامه‌نویسیِ دیگه‌ای هم برای نوشتن نرم‌افزار وجود دارن. حالا شاید شما یکی از اون جوینده‌هایی هستین که فکر می‌کنه اسکوربوت سَرِش نمیاد. با این حال بردنِ چندتا لیموترش ضرری نداره. هسکل هم همینطور، حتی اگه تنها ابزارتون نیست، توسعه‌ی نرم‌افزاری‌تون رو به کمکِ امنیتِ تایپی ارتقا میده. مشکلات و باگ‌هایی که در یه نرم‌افزار پیش میان، اونقدر پیش‌وپاافتاده نیستن که سریع

[1] Lemon

برطرف بشن. پس وقتی راه‌های بهتری برای حل اونطور مشکلات وجود دارن (نه تمام و کمال، ولی بهتر)، ارزش داره که روشون وقت بذارین.

لیمو شیرازی‌هاتون رو بذارین کنار بیاین پیشِ ما تو صفِ لیمو ترش!

این کتاب چیا داره؟

این کتاب بیشتر شبیهِ یه واحد درسی، یا یه دوره‌ی آموزشی می‌مونه تا یه کتاب. در جاهای مختلف کتاب، خیلی آزادانه تمرین‌های مختلفی قرار دادیم؛ پیشنهاد می‌کنیم حل‌شون کنین، حتی اگه به نظر خیلی ساده میان. همون تمرین‌ها تأثیرِ خیلی زیادی در درکِ یه سری مطالب دارن. هر چقدر هم ما حرف بزنیم، باز هم به خوبیِ اینکه خودتون *انجام* بدین نیست. اگه توی یکی از فصل‌های جلوتر متوجه شدین که مطلبی خوب براتون جا نیوفتاده، می‌تونین به فصل‌های قبلی‌ش برگردین و انقدر تمرین حل کنین تا قشنگ متوجه بشین.

ما عقیده داریم تکرار با فاصله‌ی زمانی، و درکِ پله‌پله، استراتژی‌های خوبی برای یادگیری هستن. اول یه چیزی رو اجمالاً اشاره می‌کنیم، بعداً بارها و بارها بهش برمی‌گردیم. همین که تجربه‌تون با هسکل کمی

عمق پیدا کنه، دیگه یه پایه‌ای دارین که از اونجا سطحِ درک‌تون رو عمیق‌تر کنین. اگه با بارِ اول مفهوم یا مطلبی رو کامل متوجه نشدین سعی کنین خیلی نگران نشین. با انجامِ تمرین‌ها و برگشت به مفاهیم، درکِ خوبی از برنامه‌نویسیِ تابعی پیدا می‌کنین.

تمرین‌های چند فصلِ اول طوری طراحی شدن تا شما رو سریعاً با گرامرِ هسکل و تایپ سیگنچرها[۱] آشنا کنن، ولی انتظارِ سخت‌تر شدنِ تمرین‌ها با هر فصلِ بعدی رو داشته باشین. هر جا که میشد، سعی کنین نمونه کدها و تمرین‌ها رو اول توی ذهن‌تون استدلال کنین، بعد برای تست کردن، تایپ‌شون کنین (بنویسین)، حالا چه توی REPL[۲]، چه توی یه فایلِ منبع[۳]. این کار قابلیتِ درک و استدلالِ برنامه‌ها و همینطور خودِ هسکل رو خیلی بیشتر می‌کنه. تمرین‌های آخری ممکنه

---

[۱] Type Signatures

[۲] این مخففِ read-eval-print loop ه، که به ترتیب میشه: خوندن، محاسبه، چاپ، تکرار (حلقه). یه محیط یا شِل (shell) محاوره‌ای برای برنامه‌نویسی که expression یا بیانیه‌ها رو محاسبه، و نتیجه‌شون رو در همون محیط برمی‌گردونه. اسمِ REPL ی که ما استفاده می‌کنیم، GHCi ه؛ i برای interactive (تعاملی).

[۳] Source File

یه کم سخت باشن. اگه برای مدتِ طولانی روی یه تمرین گیر کردین، رَدِش کنین و یه روز دیگه دوباره برین سراغش.

ما ترکیبی از مسائلِ انتزاعی و عملی رو برای استفاده از هسکل در پروژه‌های متنوعی پوشش دادیم. تجربه‌ی کریس در سیستم‌های بک‌اِند[۱] تولیدی، و فرانت‌اندِ[۲] برنامه‌هایِ تحتِ وب[۳]ه. جولی یه زبان‌شناسه، و تحصیلات‌ش در رشته‌ی معلِمیه؛ یادگرفتنِ هسکل اولین تجربه‌ش با برنامه‌نویسیِ کامپیوتر بوده. و این کتاب به کمکِ همه‌ی این تجربه‌ها اولویت‌بندی شده. هدفِ ما اینه که به شما کمک کنیم تا نه تنها کدهای تابعی با امنیتِ تایپی بنویسین، بلکه طوری خوب و عمیق یاد بگیرین که با توجه به علایق و اولویت‌هاتون، اگه خواستین بتونین پروژه‌های پیشرفته‌ترِ هسکل رو دنبال کنین.

---

[۱] Backend

[۲] Frontend

[۳] Web Applications

هر فصل به جنبه‌های مختلفِ یک موضوعِ بخصوص متمرکز شده. با یه معرفی کوتاه از جبرِ لاندا[1] شروع می‌کنیم. ربطش به برنامه‌نویسی چیه؟ همه‌ی زبان‌های برنامه‌نویسیِ تابعی بر پایه‌ی جبرِ لاندا درست شدن، و یه آشناییِ اجمالی با جبرِ لاندا، در آینده به هسکل‌نویسی‌تون کمک می‌کنه. برای مثال اگه جبر لاندا رو متوجه بشین، دیگه فهمیدنِ یکی از قابلیت‌ها به اسمِ currying اصلاً کاری نداره.

چند فصلِ بعد، بیانیه‌ها و توابعِ پایه‌ی هسکل، چندتا عملیاتِ ساده روی string ها (نوشته‌ها)، و چندتا تایپِ اساسی رو پوشش میدن. ممکنه به شدت وسوسه بشین که از گوشه‌کنارِ چندتا فصلِ اول بزنین یا کلاً رَدِشون کنین (مخصوصاً اگه قبلاً برنامه‌نویسی کرده باشین)، *ولی خواهش می‌کنیم این کار رو نکنین.* حتی اگه اون فصل‌های اول مفاهیمی رو میگن که بلدین، وقت گذاشتن برای آشنایی و راحتی با گرامرِ خلاصه‌ی هسکل، درکِ کاملِ فرق‌های بین کار توی REPL در مقابلِ استفاده از فایل‌های منبع، و آشنایی با پیغام‌های خطای (گاهاً

[1] م. جبرِ لاندا یا Lambda Calculus. این لاندا همون حرفِ یونانیِ ه (λ). درسته که تلفظِ صحیحِش لامبدا، یا درست‌تر بگیم "لَمبدا"ِه، ولی به خاطرِ جا افتادگی‌ش، لاندا عموماً خواناتره.

مبهم) ِکامپایلر خیلی مهمه. قطعاً ایرادی نداره سریع تموم‌شون کنین، فقط ردِشون نکنین.

بعد از اون چندتا فصل، دیگه هم گسترده‌تر پیش میریم، هم عمیق‌تر. وقتی کتاب رو تموم کنین، نه تنها یاد می‌گیرین موندها چی هستن، بلکه می‌تونین در برنامه‌های خودتون خیلی خوب ازشون استفاده کنین و جبرِ زیرِش رو درک کنین. قول میدیم – واقعاً همینطور میشه. تنها خواهشی که داریم ازتون اینه که بعدش نرین روی بلاگ‌تون یه آموزِشِ موند بنویسین و موندها رو به دُلمه‌ی هالوپینو تشبیه کنین!

می‌تونین توی هر فصل انتظار این موارد رو داشته باشین:

- افزایشِ تعداد تابع‌های استانداردی که بلدین؛
- آشنایی با الگوهای گرامریِ پیشرفته‌تری که بر پایه‌ی الگوهای قبلی درست شدن؛
- یاد گرفتنِ پایه‌های نظری برای درکِ طرزِ کارِ هسکل؛
- مثال‌های واضح و عملی از نحوه‌ی خوندنِ کدِ هسکل؛

- نمایشِ قدم‌به‌قدم برای نوشتنِ تابع‌های خودتون؛
- توضیحاتی برای طرزِ خوندنِ پیغام‌های خطای رایج و نحوه‌ی اجتناب از اون خطاها؛
- تمرین‌هایی با سختی‌های متنوع؛
- تعاریفِ لغاتِ مهم.

تعاریف رو در آخرِ اکثرِ فصل‌ها آوردیم. البته هر لغت در طولِ فصل هم تعریف شده، اما یه تعریفِ مستقل هم در آخر فصل‌ها برای دوره نوشتیم. اینطوری اگه بینِ دوتا فصل کمی فاصله بیوفته، با خوندنِ اون تعاریف، براتون یادآوری میشه که تا اونجا چه چیزهایی یاد گرفته بودین. هر موقع هم فکر می‌کنین یه دوره لازم دارین، راحت می‌تونین پیداشون کنین.

در آخرِ اکثرِ فصل‌ها یه سری منابعِ پیشنهادی هم برای مطالعه نوشتیم. مسلماً مطالبِ واجبی نیستن، ولی برای ما خیلی مفید بودن و

فکر کردیم که ممکنه به شما هم در درکِ بهترِ مطالبی که در طولِ فصل خوندین کمک کنن.

بهترین روش‌ها برای کار با مثال‌ها و تمرین‌ها

ما سعی کردیم در هر فصل، مثال‌ها و تمرین‌های متنوعی آورده باشیم. با اینکه همه‌ی تلاش‌مون رو کردیم تا آموزنده‌ترین تمرین‌ها رو بنویسیم، متوجه هستیم که شاید بعضی تمرین‌ها برای بعضی‌ها جذابیت نداشته باشن. و با توجه به اینکه خواننده‌های این کتاب با پیش‌زمینه‌های متنوعی میان، بعضی تمرین‌ها یا به نظر خیلی آسون میان یا به نظر خیلی سخت، و احتمالاً همون تمرین‌ها برای یه سری دیگه از خواننده‌ها کاملاً مناسب باشن. سعی کنین بیشترین تمرین‌هایی که برای شما مناسب هستن رو حل کنین. ولی اگه مثلاً همه‌ی تمرین‌های تایپ و تایپکلاس‌ها رو رَد کرده بودین و سرِ مانوید[1]ها یه کم سردرگم شدین، برگردین و انقدر تمرین کنین تا یاد بگیرین.

اینها رو به خاطر داشته باشین تا از تمرین‌ها بیشترین بهره رو ببرین:

---

[1] Monoid

- مثال‌ها معمولاً طوری طراحی شدن تا با کدِ واقعی، مطلبی که تازه گفته شده، یا قراره با جزئیاتِ بیشتری توضیح داده بشه رو نمایش بدن.

- قراره همه‌ی مثال‌ها رو توی REPL یا یه فایل، تایپ و تست کنین. به شدت توصیه می‌کنیم بعد از اینکه کدِتون کار کرد، سعی کنین تغییرِش بدین و یه کم باهاش بازی کنین. این که یه فرضیه‌ای از نتیجه‌ی تغییرات‌تون بدین و بعد ببینین درست بوده یا نه، واقعاً مهمه! و اگه خودتون تایپ کنین (یعنی کپی/پیست نکنین) خیلی بهتره، چون باعث میشه بیشتر به کدها توجه کنین.

- بعضی مثال‌ها عمداً کار نمی‌کنن، ولی حتماً توی متن چیزی از خرابی‌شون نوشته شده. اگه خطای غیرمنتظره‌ای گرفتین که ما هم چیزی ازش نگفتیم، کدتون رو چک کنین تا خطای گرامری نداشته باشین.

- همه‌ی مثال‌ها برای تایپ در REPL طراحی نشدن؛ همه‌ی مثال‌ها برای نوشتن توی یه فایل هم طراحی نشدن. بعد از توضیحِ تفاوت‌های گرامری بینِ بیانیه‌ها در REPL و در فایل، انتظار میره که خودتون کدها رو از یکی به اون یکی منتقل کنین. تا انتهای کتاب دیگه باید خیلی راحت با کد کار کنین. بهتره که خورده‌خورده بجای نوشتنِ مثال‌ها و تمرین‌ها توی GHCi، سعی کنین به کار کردن با فایل‌های منبع عادت کنین (البته به جز موردهای خاص). ویرایش و تغییر کد در یه فایلِ منبع خیلی آسون‌تر و عملی‌تره. بعد هم کدِتون رو توی GHCi بارگذاری و اجرا می‌کنین.

- بهتره که تمرین‌ها، بخصوص تمرین‌های طولانی‌تر رو به عنوانِ ماژول‌های اسم‌دار ذخیره کنین. تمرین‌های خیلی زیادی هستن که بارها بهشون برمی‌گردیم (مخصوصاً آخرهای کتاب)، پس قابلیتِ بارگذاریِ مجددِ اونها، جلوی کدنویسیِ تکراری رو می‌گیره. سعی کردیم تو تمرین‌هایی که این کار لازمه، یه اشاره‌ای کرده باشیم.

- بعضی تمرین‌های آخرِ فصل ممکنه فقط برای دوره‌ی مطالب فصل‌های قبل آورده شده باشن، و اکثراً از آسون به سخت مرتب شدن.

- حتی تمرین‌هایی که به نظر آسون میان، به رَوون شدن در یه مبحث کمک می‌کنن. ما سختی رو به خاطرِ صرفاً سخت بودن تأیید نمی‌کنیم. فقط می‌خوایم تا میشه مطالب رو خوب درک کنین. برای همین ممکنه لازم بشه یک مسئله رو از چند زاویه بیان کنیم.

- ازتون خواستیم که خیلی از تابع‌ها رو بازنویسی کنین (با گرامرهای متفاوت). مسائلِ خیلی کمی هستن که فقط یه راهِ حل دارن، و حلِ یک مسئله از راه‌های مختلف به تسلط و راحتی‌تون با طرزِ کار هسکل (گرامرِش، مفاهیم‌ش، و در بعضی موارد، ترتیبِ محاسبات‌ش) کمک می‌کنه.

- خودتون رو به حلِ همه‌ی تمرین‌ها در مطالعه‌ی اولِ یه فصل ملزم نکنین. تکرارهای با فاصله تأثیرِ بیشتری دارن.

- بعضی تمرین‌ها، بخصوص در فصل‌های اول، ممکنه به نظر زورَکی یا ساختگی بیان. رُک بگیم، همینطور هم هستن. ولی به این دلیل ساختگی‌اند تا موضوع‌های بخصوصی رو درس بدن. همین که پیش میرین و بیشتر هسکل یاد می‌گیرین، تمرین‌ها یه کم از اون ساختگی بودن دور میشن و به "هسکلِ واقعی" نزدیک‌تر میشن.

- یکی دیگه از مزایای کدنویسی توی فایلِ منبع و بعد بارگذاری‌ش توی REPL اینه که می‌تونین در رابطه با فرایندی که سپری کردین تا به جواب رسیدین، کامِنت[1] بنویسین. نوشتنِ افکارِتون در راهِ رسیدن به جواب، به شفاف شدنِ افکارِتون و حلِ مسائل مشابه کمک می‌کنه. حداقل‌ش اینه که بعداً با خوندنِ کامنت‌هاتون یه چیزهایی از خودتون یاد می‌گیرین.

---

[1] Comment

- بعضی وقت‌ها عمداً تعریفِ یه تابع رو ناقص می‌نویسیم. چیزهایی مثل این زیاد می‌بینین:

```
f = undefined
--  ^-------^
--  تعریف نشده .م
```

با اینکه به احتمالِ زیاد f چندتا آرگومان هم می‌گیره، ولی ما پارامترهاش رو براتون نامگذاری نمی‌کنیم. موقعِ نوشتنِ پروژه‌های خودتون، کسی نیست دست‌تون رو بگیره، پس انتظار نداشته باشین که این کتاب هم این کار رو کنه.

فصل ۲

سلام هسکل!

در دریایی از آشفتگی، توابع
سَمبُلی از ثبات اند.
– مایک هموند

Functions are beacons of constancy in a sea of turmoil.
– Mike Hammond

۲ – ۱ سلام، هسکل

به اولین قدمتون برای یاد گرفتنِ هسکل خوش اومدین. قبل از ادامه‌ی این کتاب، باید ابزار لازم رو نصب کنین تا بتونین تمرین‌ها رو حل کنین. فعلاً ما نصب Stack رو پیشنهاد می‌کنیم، که هم GHC Haskell، هم محیط تعاملی یا interactive اِون به اسم GHCi، و هم یه ابزار برای ساختِ پروژه و مدیریتِ وابستگی[۱]ها رو با هم نصب می‌کنه.

راهنمای نصب رو می‌تونین از آدرسِ `http://docs.haskellstack.org/en/stable/README/` پیدا کنین.[۲] مستنداتِ[۳] خیلی خوبی هم برای Stack اونجا هست که برای شروع کمک می‌کنه. یه راهنمای نصبِ دیگه هم می‌تونین از `https://github.com/bitemyapp/learnhaskell` پیدا کنین؛

---

[۱] Dependency Management

[۲] م. نسخه‌ی فارسی‌ش رو هم می‌تونین روی سایتِ کتاب بخونین:
`http://haskellbook.ir/docs`

[۳] Documentation

اینجا منابع دیگه‌ای هم برای یادگرفتنِ هسکل گذاشتیم، به علاوه‌ی لینکِ تمرین‌های بیشتر که احتمالاً در کنار این کتاب مفید باشن.

از اینجا به بعد فرض بر اینه که هسکل رو نصب کردین و آماده‌این. تو این فصل:

- کدِ هسکل رو، هم در محیط محاوره‌ای و هم از فایل اجرا می‌کنیم؛
- دو جزء مهم هسکل رو می‌شناسیم: بیانیه‌ها و توابع؛
- چندتا از مشخصاتِ syntax یا گرامرِ هسکل، و سبک نوشتاریِ خوبِ کد هسکل رو یاد می‌گیریم؛
- و توابع ساده رو تغییر میدیم.

۲ – ۲ تعامل با کد هسکل

هسکل دو راهِ اصلی برای کار با کد در اختیار میذاره. اولی نوشتنِ کد مستقیماً تو محیط تعاملی یا interactiveِ هسکل به اسم GHCi یا

REPL ِه. راه دوم نوشتنِ کد به کمک یه برنامه‌ی ویرایش متن[۱] توی یه فایل، و بعد بارگذاریِ اون فایلِ منبع[۲] تو GHCi ِه. این بخشِ کتاب هر دو راه رو معرفی می‌کنه.

استفاده از REPL

REPL مخفف read-eval-print loop ِه، که به ترتیب میشه: خوندن، محاسبه، چاپ، تکرار. REPL ها محیط‌های برنامه‌نویسیِ تعاملی‌اند که می‌تونین کدتون رو وارد کنین و نتیجه‌ش رو بعد از محاسبه ببینید. اولین زبانی که چنین محیطی داشت Lisp بود، ولی الان اکثر زبانهای مدرن، REPL دارن.

---

[۱] Text Editor

[۲] Source File

اگه هسکل رو نصب کرده باشین، باید بتونین از terminal (یا command prompt در ویندوز) اجراش کنین. وقتی تایپ کنید `ghci` یا `stack ghci`[۱]، و enter رو بزنین، باید یه پیغامی مثل این ببینید:

```
GHCi, version 7.10.3:
  http://www.haskell.org/ghc/    :? for help
Prelude>
```

اگه `stack ghci` رو زدین[۲] احتمالاً نوشته‌های بیشتری اومدن، prompt[۳]ِ تون هم ممکنه `Prelude` نباشه. موردی نداره. ممکنه نسخه‌ی GHC تون هم فرق داشته باشه، که اگه بین ۷٫۸ و ۸٫۰ باشه، به احتمال قوی با این کتاب هم سازگاره.

خوب، حالا چندتا محاسبه‌ی ساده رو تو prompt بنویسین:

---

[۱] اگه GHC رو خارج از Stack نصب کردین، نوشتنِ `ghci`ِ خالی اجراش می‌کنه. ولی اگه تنها GHC ای که دارین اونیه که Stack نصب کرده، باید از `stack ghci` استفاده کنید.

[۲] الان هنوز `stack ghci` رو لازم ندارین، ولی جلوتر که میریم، وقتی پروژه می‌سازیم و چندتا ماژول وارد می‌کنیم، استفاده از Stack خیلی معقول‌تره.

[۳] م. منظور از prompt، علامت، نشونه یا نوشته‌ایه که میگه کامپیوتر آماده‌ی گرفتن ورودیه.

```
Prelude> 2 + 2
4
Prelude> 7 < 9
True
Prelude> 10 ^ 2
100
```

اگه می‌تونین معادلات ساده مثلِ اینها رو تو prompt وارد کنید و نتایج صحیح بگیرین... مبارکه – حالا دیگه یه برنامه‌نویس تابعی شدین! دیگه اینکه REPLِ تون کار می‌کنه و می‌تونین ادامه بدین.

از GHCi با دستور `quit:` یا `q:` میشه خارج شد.

`Prelude` چیه؟ `Prelude` یه کتابخونه یا library از توابع استانداره. وقتی GHCi یا Stack GHCi رو باز می‌کنین، همه‌ی این توابع خودبه‌خود بارگذاری یا load میشن، و دیگه لازم نیست کار خاصی انجام بدین تا بتونین صداشون بزنین. بعداً می‌بینیم که میشه `Prelude` رو خاموش کرد، و اینکه حتی prelude های دیگه‌ای هم وجود دارن (البته تو این کتاب ازشون استفاده نمی‌کنیم). `Prelude` از مشمولاتِ پکیج[۱]

[۱] Package

(بسته) base هسکل، که میشه از https://www.stackage.org/package/base گرفتش. هر از گاهی می‌نویسیم که یه چیزی "در base ه" که منظورمون اینه که از مشمولات اون پکیجِ بزرگِ استانداردِه.

دستورات GHCi

در طول کتاب از دستورات GHCi تو REPL مثلِ :quit و :info استفاده می‌کنیم. اینها دستورات خاصی اند که فقط GHCi میشناسه و همه‌شون با کاراکترِ : (دو نقطه) شروع میشن. :quit کد هسکل نیست؛ فقط یکی از امکانات GHCi ه.

ما این دستورات رو کامل می‌نویسیم، ولی عموماً میشه به حرف اولشون خلاصه بشن. مثلاً :quit میشه :q یا :info میشه :i و بقیه به همین ترتیب. چند بارِ اول خوبه که کامل تایپ کنین تا یادتون بمونه هر حرف مخفف چیه، ولی بعد از چند بار، دیگه خلاصه می‌نویسیم.

کدنویسی در فایل

بیشترِ وقت‌ها لازم میشه کد رو تو یه فایل ذخیره کنین تا بتونین خورده خورده بسازین‌ش. تقریباً هر برنامه‌ی درست‌وحسابی‌ای که می‌نویسین، باید براش کتابخونه‌های مختلف یا برنامه‌هایی که پوشه‌های تودرتو با فایل‌های هسکل دارن رو ویرایش کنین. فرایندِ اصلی اینجوریه که کد و واردات[۱] لازم (واردات رو بعداً بیشتر توضیح میدیم) برای برنامه رو تو یه فایل بنویسین، بیارین تو REPL و اونجا باهاش کار کنین. اینطوری می‌تونین برنامه‌تون رو خورده‌خورده تست و ویرایش کنین تا کامل شه.

یه فایل به اسم `test.hs` بسازین. پسوند `hs.` برای فایل‌های هسکل به کار میره. می‌تونین اول فایل رو بسازین بعد توی یه text editor بازش کنین و توش کد بنویسین، یا می‌تونین اول برنامه‌ی ویرایشِ متن رو باز کنین و کد رو توی یه فایل جدید بنویسین، بعد هم به اون نام ذخیره‌ش کنید. هر طور که راحتین.

---

[۱] Imports

کد زیر رو توی اون فایل تایپ و بعد هم ذخیره‌ش کنید:

```
sayHello :: String -> IO ()
sayHello x =
  putStrLn ("Hello, " ++ x ++ "!")
```

اینجا بعد از :: (دو تا دو نقطه)، *نشانِ نوع*[۱] یا *تایپ سیگنچرِ* تابع رو نوشتیم. میشه اینطوری خوندش: "دارای تایپِ ... است" که به جای سه نقطه، تایپ سیگنچرِ جلوش رو می‌خونیم. پس sayHello دارای تایپِ String -> IO () است. تمرکزِ این فصل‌های اول روی گرامره (syntax)، جلوتر تو یکی از فصل‌ها، تایپ یا نوع‌ها رو توضیح میدیم.

حالا تو همون پوشه‌ای که test.hs رو ذخیره کردین، ghci رو باز کنید و دستورات زیر رو وارد کنید:

```
Prelude> :load test.hs
Prelude> sayHello "Haskell"
Hello, Haskell!
Prelude>
```

---

[۱] Type Signature

بعد از اینکه با `:load` فایل `test.hs` رو بارگذاری کردین، تابعِ `sayHello` در دسترسِ REPL قرار گرفت. بعد یه آرگومانِ string، مثل `"Haskell"` رو بهش دادین (به اون علامتهای نقل قول دقت کنید)، و تونستین جوابش رو ببینید.

ممکنه متوجه شده باشین که بعد از بارگذاریِ اون فایل، prompt ِتون دیگه `Prelude>` نیست. اگه میخواین دوباره `Prelude>` بشه، دستور `:m` که خلاصه‌ی `:module` هست رو وارد کنین. این دستور فایل‌هاتون رو از GHCi برمیداره، و دیگه REPL به کدهای اون فایل‌ها دسترسی نداره.

۲ – ۳ درک بیانیه‌ها

هر چیز تو هسکل یا یه تعریف[۱]ه، یا یه بیانیه (expression). بیانیه‌ها می‌تونن مقادیر باشن، ترکیبی از مقادیر باشن، و یا توابعی که به مقادیر اعمال شدن باشن. *بیانیه‌ها* محاسبه میشن و به یک جواب میرسن. بیانیه‌هایی که یه مقدار مشخص‌اند، محاسبه‌ی پیش و پا افتاده‌ای دارن،

[۱] Declaration

چراکه خودشون جواب محاسبه‌اند. بیانیه‌هایی که یه معادله‌اند، محاسبه‌شون در واقع همون محاسبه‌ی عملگر[۱] با آرگومان‌هاشه. با اینکه برنامه‌های هسکل فقط شاملِ حل محاسباتِ ریاضی نیستن، ولی همه‌ی بیانیه‌های هسکل به نحوِ مشابهی کار می‌کنن: خیلی واضح و شفاف به یه نتیجه‌ی قابل پیش‌بینی محاسبه میشن. هسته‌ی وجودیِ برنامه‌های هسکل همین بیانیه‌ها اند، کلِ اون برنامه هم در واقع یه بیانیه‌ی بزرگه که متشکل از بیانیه‌های کوچکتره.

در مورد تعاریف یا declarations، فعلاً کفایت می‌کنه که بگیم انقیادهای سطح بالا[۲] اند، و این اجازه رو میدن که بیانیه‌ها رو نامگذاری کنیم. بعداً می‌تونیم به جای کپی کردن اون بیانیه‌ها از اسمشون استفاده کنیم.

چندتا مثال از بیانیه‌ها:

---

[۱] Operator

[۲] Top-level Bindings

```
1
1 + 1
"Icarus"
```

هر کدوم رو می‌تونین تو GHCi REPL بررسی کنین، کد رو تو prompt تایپ کنین و enter بزنین تا جواب محاسبه‌ی بیانیه رو ببینین. برای مثال، مقدار عددی ۱ ساده‌تر نمیشه، پس خودش برمی‌گرده.

اگه تا الان این کارو نکردین، GHCi رو باز کنید و این مثال‌ها رو همراهی کنید.

وقتی تو GHCi این رو تایپ می‌کنیم:

```
Prelude> 1
1
```

عدد ۱ چاپ میشه، چرا که ساده‌تر نمیشه.

در مثال بعد، GHCi بیانیه‌ی `1 + 2` رو به 3 ساده می‌کنه و بعد عدد ۳ رو چاپ می‌کنه. از ۳ ساده‌تر نمیشه چون جمله‌ای برای محاسبه نمی‌مونه:

```
Prelude> 1 + 2
3
```

تعداد دفعاتی که می‌تونین بیانیه‌ها رو تودرتویِ هم بنویسین، نامحدوده، فقط اگه حوصله‌ی نوشتنش رو داشته باشین!

```
Prelude> (1 + 2) * 3
9
Prelude> ((1 + 2) * 3) + 100
109
```

میشه همینطور ادامه داد، لایه‌های بیانیه‌تون رو بیشتر کنین و محاسبه کنین. ولی بیانیه‌ها به همین‌ها محدود نمیشن.

حالت معمولی وقتی نشه بیانیه‌ها رو بیشتر محاسبه کرد، یا به کلام دیگه ساده‌نشدنی باشن، میگیم در حالتِ معمولی[۱]اند. حالت معمولیِ 1 + 1 میشه ۲. چرا؟ چون بیانیه‌ی 1 + 1 رو میشه با اعمالِ اوپراتورِ جمع به دو تا آرگومانش ساده کرد. به زبان دیگه، 1 + 1 یه بیانیه‌س که ساده‌شدنیه، و ۲ هم یه بیانیه‌س، ولی ساده‌تر نمیشه — به هیچ چیزی

[۱] Normal Form

غیر از خودش محاسبه نمیشه. بیانیه‌های ساده‌شدنی رو redex[۱] هم میگن. ما توی کتاب به این فرایند، محاسبه یا ساده‌سازی میگیم، ولی ممکنه بعضی جاها لغات normalizing یا اجرا[۲] رو هم ببینید (خیلی لغات دقیقی نیستن).

۲ – ۴ توابع

بیانیه‌ها از پایه‌های اساسیِ هسکل‌اند، و توابع یه حالت خاص از بیانیه‌ها هستن. توابع توی هسکل خیلی شبیه توابع ریاضی می‌مونن، یعنی نگاشتی از یک مجموعه ورودی به یک مجموعه خروجی. تابع یه بیانیه‌ست که به یک آرگومان اعمال شده و همیشه جوابی رو برمی‌گردونه. این جواب به ازای یک ورودیِ مشخص، همیشه یکسانه، چرا که توابعِ هسکل از بیانیه‌های خالص درست شدن.

درست مثل جبر لاندا، تمامی توابعِ هسکل یک آرگومان می‌گیرن و یک جواب برمی‌گردونن. وقتی به نظر میاد داریم چندتا آرگومان به یه

---

[۱] م. ترکیبی از دو لغت reducible و expression.

[۲] Executing

تابع میدیم، در واقع داریم چندتا تابعِ تودرتو رو تک‌تک به یکی از آرگومان‌ها اعمال می‌کنیم. به این کار میگن currying.

ممکنه متوجه شده باشین که بیانیه‌هایی که تا الان استفاده کردیم فقط مقادیرِ معین داشتن و هیچ متغیر (یا تجرید) ی نداشتن. با توابع میشه اون بخشهایی از کد که می‌خوایم بعداً با مقادیرِ مختلف استفاده کنیم رو انتزاع یا abstract کنیم.

برای مثال فرض کنید چندتا بیانیه رو میخواین با ۳ ضرب کنید. تک‌تک بیانیه‌ها رو اینطوری وارد می‌کنین:

```
Prelude> (1 + 2) * 3
9
Prelude> (4 + 5) * 3
27
Prelude> (10 + 5) * 3
45
```

این راهِ خیلی درستی نیست... با توابع میشه چنین الگوهای تکراری رو فاکتور کرد تا بعداً دوباره ازشون استفاده کنیم. این کار رو با دادنِ یه اسم به تابع و معرفیِ یه متغیرِ مستقل به عنوان پارامترِ تابع انجام میدیم.

توابع می‌تونن در بدنه‌ی بقیه‌ی توابع ظاهر بشن، یا حتی به عنوان آرگومان به بقیه‌ی توابع (یا حتی خودشون) استفاده بشن، درست مثل هر مقدار یا value ِدیگه‌ای.

در مثالِ بالا، ما یه سری بیانیه داریم که می‌خوایم با ۳ ضرب بشن. اگه از دیدِ تابع نگاه کنیم، می‌پرسیم کدوم بخش بین همه‌ی بیانیه‌ها مشترکه؟ و کدوم بخش فرق می‌کنه؟ چه اسمی برای تابع انتخاب کنیم؟ و اینکه چه جور آرگومان‌هایی رو قبول می‌کنه؟

اون 3 * بخشِ مشترک بیانیه‌هاس. قسمتی که تو هر کدوم متفاوته، بیانیه‌ی جمعِ قبل از اون ضربه، پس بجاش یه متغیر میذاریم و یه اسم هم برای تابع‌مون تعریف می‌کنیم. وقتی یه مقدار رو با متغیر جایگزین می‌کنیم، تابع حسابش می‌کنه، با ۳ ضربش می‌کنه، و نتیجه‌ش رو برمی‌گردونه. در بخش بعد، همه‌ی اینها رو تو هسکل پیاده می‌کنیم.

تعریف کردن توابع

چندتا چیز بین همه‌ی تعاریفِ تابع‌ها مشترک‌ه. اول اینکه همشون با اسم تابع شروع میشن و جلوشون پارامتر[۱]های تابع میان (همه‌ی اینها فقط با فاصله[۲] از هم جدا میشن). بعد یه علامت مساوی میاد، که تساویِ جملات رو بیان می‌کنه. نهایتاً بدنه‌ی تابع میاد که در واقع همون بیانه‌ایه که بعد از اعمال شدن به یه آرگومان، میشه به جوابِ نهایی محاسبه بشه.

---

[۱] در عمل، دو لغتِ *آرگومان* و *پارامتر* به جای همدیگه استفاده میشن، ولی با هم فرق دارن. *آرگومان*ها مقادیری‌اند که بعد از اعمالِ تابع با *پارامتر*های اون جایگزین میشن. اون متغیرهایی که در definitionِ تابع (یا تایپ سیگنچرِ تابع)، نماینده‌ی پارامترهای تابع‌اند، آرگومان نیستن. برای اطلاعات بیشتر، بخش تعاریف در آخر فصل رو نگاه کنید.

[۲] Whitespace

بین تعریفِ توابع توی GHCi، با تعریف اونها توی یه فایل، یه کم فرق هست. برای تعریفِ مقادیر یا توابع در GHCi باید از `let` استفاده کنید[۱]:

```
Prelude> let triple x = x * 3
```

همین تابع رو توی فایل اینطوری می‌نویسیم:

```
triple x = x * 3
```

حالا هر بخش‌ش رو جداگانه بررسی می‌کنیم:

```
  triple x   =   x * 3
-- [1]  [2] [3]  [ 4 ]
```

۱. این اسمِ تابعه؛ که در واقع تعریف یا declarationِ یه تابع‌ست. دقت کنید که با حرف کوچیک شروع شده.

---

[۱] از نسخه‌ی GHC ۸٫۰٫۱ به بعد، دیگه استفاده از `let` برای تعریف در GHCi لازم نیست. با این حال ما فعلاً در این کتاب `let` ها رو نوشتیم تا کدها با نسخه‌های قدیمی‌تر هم سازگاری داشته باشند. البته نوشتنِ این `let` های اضافه نباید باعث خطایی بشن.

۲. پارامتر تابع. این پارامترهای تابع معادلِ سَرِ لاندا هستن و متغیرهای داخلِ بدنه‌ی تابع رو قید می‌کنن.

۳. علامت = برای *تعریفِ* مقادیر یا توابع در هسکل استفاده میشه، و برای بررسی تساوی بین دو مقدار *نیست*.

۴. بدنه‌ی تابع. در صورتِ اعمالِ تابع به یه مقدار، این بیانیه (بدنه‌ی تابع) رو میشه محاسبه کرد. اگه `triple` به یه آرگومان اعمال بشه، مقدار اون آرگومان با x های مقیّد جایگزین میشه. اینجا بیانیه‌ی `x * 3`، بدنه‌ی تابعه. پس در یه بیانیه‌ای مثلِ `triple 6`، x به ۶ مقیّد میشه.

اندازه‌ی حروف مهمه! اسم توابع با حروف کوچیک شروع میشن. گاهی سَبکِ شُتُری[۱] برای تفهیمِ بهترِ نقش یه تابع کار خوبیه. ولی حرفِ اول باید کوچک باشه.

---

[۱] camelCase

متغیرها هم با حرف کوچیک شروع میشن. لزومی هم نداره که تک حرفی باشن.

بازی با تابعِ `triple` اول سعی کنین تابعِ `triple` رو با `let` توی REPL تعریف کنین. بعد تابع رو با یه مقدارِ عددی (به ازای آرگومان‌ش) صدا بزنین:

```
Prelude> triple 2
6
```

حالا بدونِ `let`، توی یه فایل تعریف و ذخیره‌ش کنین. بعد که با دستور `load:` یا `l:` توی GHCi بارگذاریش کردین، می‌تونین تابع رو با اسم‌ش، یعنی `triple`، و یه مقدارِ عددیِ جلوش، صدا بزنین؛ همونطوری که بالاتر توی REPL انجام دادین. با مقادیر مختلف – اعداد صحیح یا حتی یه معادله‌ای مثل `(1 + 1)` – تست کنین. بعدش سعی کنین خود تابع رو توی فایل تغییر بدین. فایل رو دوباره بارگذاری کنید و نتیجه‌ی تغییرات رو ببینید. برای بارگذاریِ مجددِ همون فایل، می‌تونین از دستور `reload:` یا `r:` استفاده کنین.

۲ – ۵ محاسبه

منظور از محاسبه‌ی یک بیانیه، ساده کردن اون تا جای ممکنه. وقتی یه جمله به ساده‌ترین حالت‌ش برسه، میگیم محاسبه‌ش تموم شده، یا ساده‌نشدنی[۱] ِه. یه بیانیه تو این حالت، معمولاً یه مقدارِ مشخصه. رَوشِ محاسبه‌ی هسکل، *نا اَکید*[۲] ِه (*محاسبه‌ی تنبل*[۳] هم بهش میگن)، یعنی هسکل محاسبه‌ی جملات رو تا زمانی که از طرفِ جملاتِ دیگه اجبار نَشَن به تعویق میندازه.

مقادیر ساده‌نشدنی‌اند، ولی اعمالِ توابع به آرگومان‌ها رو میشه ساده کرد. ساده کردنِ یه بیانیه یعنی محاسبه‌ی جملات تا زمانی که به یه مقدار یا value برسیم. درست مثل جبر لاندا، اعمالِ تابع به یه آرگومان، امکان ساده شدن یا محاسبه‌ش رو فراهم می‌کنه.

---

[۱] Irreducible

[۲] Non-strict

[۳] Lazy Evaluation

مقادیر بیانیه‌اند، ولی ساده‌تر نمیشن. یعنی مقادیر نقطه‌ی پایانیِ ساده‌سازی‌اند:

```
1
"Icarus"
```

بیانیه‌های زیر رو میشه به یه مقدار ساده کرد:

```
1 + 1
2 * 3 + 1
```

هر کدوم رو میشه توی REPL محاسبه کرد. REPL اول ساده‌شون می‌کنه، و بعد نتیجه‌شون رو چاپ می‌کنه.

برگردیم به تابعِ `triple`. وقتی تابع رو با یه آرگومان صدا میزنیم، در واقع یه بیانیه داریم که میشه ساده‌ترش کرد. در یک زبانِ تابعیِ خالص مثل هسکل، می‌تونیم توابعِ اعمال‌شده رو با تعریفِ‌شون جابجا کنیم و همون جواب رو بگیریم؛ مثل ریاضی. پس وقتی جمله‌ی زیر رو می‌بینیم:

```
triple 2
```

میدونیم بیانیه‌های زیر باهاش معادل‌اند:

```
triple 2
-- [triple x = x * 3; x := 2]
2 * 3
6
```

تابع `triple` رو به عدد ۲ اعمال کردیم و بعد به جوابِ نهایی، یعنی ۶، ساده‌ش کردیم. *حالت معمولیِ* بیانیه‌ی `triple 2`، عدد ۶ِه، چون ساده‌تر نمیشه.

هسکل به طور پیش‌فرض همه‌ی بیانه‌ها رو تا حالت معمولی‌شون حساب نمی‌کنه، بلکه فقط تا *حالت معمولی با سرِ ضعیف*[۱] (WHNF) پیش میره. یعنی هر چیزی بلافاصله تا آخر ساده نمیشه. برای مثال، جمله‌ی زیر:

```
(\f -> (1, 2 + f)) 2
```

در WHNF به جمله‌ی زیر ساده میشه:

---

[۱] Weak Head Normal Form

```
(1, 2 + 2)
```

این یه ارائه‌ی تقریبی‌ه، ولی نکته اینجاست که 2 + 2 تا آخرین لحظه‌ی ممکن به ۴ ساده نمیشه.

تمرین‌ها: بررسی ادراک

۱. کدهای زیر توی یه فایل نوشته شدن. اگه بخواین این توابع رو مستقیماً توی REPL بنویسین، چه تغییراتی باید بدین؟

```
half x = x / 2

square x = x * x
```

۲. یه تابع با یک پارامتر بنویسین که به جای همه‌ی بیانیه‌های زیر کار کنه. براش اسم هم بذارین.

```
3.14 * (5 * 5)
3.14 * (10 * 10)
3.14 * (2 * 2)
3.14 * (4 * 4)
```

۳. یک مقدار به اسمِ `pi` در `Prelude` وجود داره. تابع‌تون رو با این `pi` به جای ۳٫۱۴ بازنویسی کنید.

۲ – ۶ عملگرهای میانوند

توابعِ هسکل در حالت پیش‌فرض، syntax یا گرامرِ پیشوندی[۱] دارن؛ یعنی تابعی که داره اعمال میشه، بجای وسط، اولِ بیانیه‌س. با تابعِ `triple` این رو دیدیم، بقیه‌ی تابع‌های استاندارد هم، مثل تابع همانی یا `id`، همینطوراند. این تابع فقط آرگومانش رو برمی‌گردونه:

```
Prelude> id 1
1
```

این گرامرِ پیش‌فرضِ توابعه، با این حال، همه‌ی توابع پیشوندی نیستند. یه گروهی از عملگرها هستن، مثل عملگرهای عددی[۲] (جمع،

---

[۱] Prefix

[۲] Arithmetic Operators

ضرب، و...)، که در حقیقت تابع‌اند (به آرگومان‌ها اعمال میشن و یه خروجی میدن) ولی به طور پیش‌فرض میانوند[۱] نوشته میشن.

عملگرها در واقع توابعی هستن که میشه به سبکِ میانوندی ازشون استفاده کرد. همه‌ی عملگرها تابع اند؛ ولی همه‌ی تابع‌ها عملگر نیستند. `triple` و `id` توابعِ پیشوندی اند (عملگر *نیستن*)، ولی تابعِ `+` یک عملگر یا اوپراتورِ میانونده:

```
Prelude> 1 + 1
2
```

چندتا عملگرِ عددی دیگه رو امتحان کنیم:

```
Prelude> 100 + 100
200
Prelude> 768395 * 21356345
16410108716275
Prelude> 123123 / 123
1001.0
Prelude> 476 - 36
440
```

---

[۱] Infix

```
Prelude> 10 / 4
2.5
```

با یه کم تغییر گرامری[۱]، میشه از توابع هم به سبک میانوندی استفاده کرد:

```
Prelude> 10 `div` 4
2
Prelude> div 10 4
2
```

برعکس‌ش هم ممکنه. اگه عملگرها رو توی پرانتز بذارین، تبدیل به تابعِ پیشوندی میشن:

```
Prelude> (+) 100 100
200
Prelude> (*) 768395 21356345
16410108716275
Prelude> (/) 123123 123
1001.0
```

---

[۱] م. با استفاده از اَکسان گراو یا backtick. این کاراکتر با آپاستروف فرق داره و توی بیشتر کیبوردها کنار کلید ۱ و بالای tab قرار گرفته.

اگه اسم تابع فقط متشکل از حروف و اعداد باشه، به طور پیش‌فرض پیشوندی میشه؛ و همه‌ی توابع پیشوندی رو نمیشه میانوندی کرد. اما اگه اسم تابع علامت یا symbol باشه[۱]، خودبه‌خود میانوندی میشه، و با پرانتز پیشوندی میشه[۲].

شرکت‌پذیری و تقدم

اگه از ریاضیات یادتون باشه، یه شرکت‌پذیری[۳] و تقدمِ[۴] پیش‌فرض برای عملگرهای میانوند، (*)، (+)، (-) و (/)، وجود داره.

با استفاده از دستور `:info` در GHCi، میشه اطلاعاتی مثل شرکت‌پذیری و تقدمِ توابع و عملگرها رو استعلام کرد. وقتی با `:info` درباره‌ی یه تابع یا عملگر می‌پرسین، GHCi تایپِ تابع رو بهتون میگه،

---

[۱] م. بقیه‌ی کاراکترها به غیر از حروف و اعداد.

[۲] برا عزیزانی که دوست دارن گیر بدن! نمیشه یه تابع پیشوندی رو با backtick میانوندی کنین، بعد اونو بذارین لای پرانتز تا پیشوندی شه... دلیل اینکه یه همچین چیزی رو میخواین برا ما واضح نیست!

[۳] Associativity

[۴] Precedence

و اینکه میانوند هست یا نه، و اگه بود، شرکت‌پذیری و تقدم اون رو هم چاپ می‌کنه. فعلاً اطلاعات راجع به تایپِ تابع و بقیه چیزها رو کنار میذاریم، فقط یه کم از شرکت‌پذیری و تقدم صحبت می‌کنیم.

جواب `:info` تو GHCi برای توابع (*)، (+)، و (-) از این قراره (تا این لحظه):

```
:info (*)
infixl  7   *
-- [1] [2] [3]

:info (+) (-)
infixl 6 +

infixl 6 -
```

۱. منظور از `infixl`، میانوند بودنِ اوپراتوره؛ اون `l`ِ آخرش هم برای شرکت‌پذیری از *left* یا چپ‌ه.

۲. عدد ۷، تقدم رو نشون میده: هر چیزی تقدم‌ش بیشتر باشه، اوّل اعمال میشه. بازه‌ی تقدم از ۰ تا ۹ تعریف شده.

۳. اسمِ تابعِ میانوند: در این مورد، ضرب[۱].

برای جمع و تفریق هم می‌بینیم که هردوشون عملگرِ میانوند با شرکت‌پذیری از چپ هستن، و تقدم ۶ دارن.

می‌تونیم به کمک پرانتز، مفهوم شرکت‌پذیری از چپ رو بهتر نشون بدیم. به همراهی این کدها با REPL ادامه بدین:

به خاطر شرکت‌پذیری از چپ، این:

```
2 * 3 * 4
```

به این ترتیب حساب میشه:

```
(2 * 3) * 4
```

حالا یه مثال برای شرکت‌پذیری از راست:

```
Prelude> :info (^)
infixr  8   ^
-- [1]  [2] [3]
```

---

[۱] Multiplication

۱. منظور از `infixr`، میانوند بودنِ اوپراتوره؛ اون `r`ِ آخرش هم برای شرکت‌پذیری از *right* یا *راست*ه.

۲. عدد ۸ برای تقدمه، که هر چقدر بزرگتر باشه، اون اوپراتور اول اعمال میشه. می‌بینید که این عملگر تقدمش از ضرب (۷) و جمع یا تفریق (۶) بیشتره.

۳. اسمِ تابعِ میانوند: در این مورد، توان[۱].

با ضرب، اهمیتِ شرکتپذیری واضح نبود، چون ضرب کلاً شرکت‌پذیره، و جابجاییِ پرانتزها تاثیری روی جواب نمیذاشت. ولی توان شرکت‌پذیر نیست، و گزینه‌ی خوبی برای نشون دادنِ فرق بین شرکت‌پذیری از چپ و شرکت‌پذیری از راسته:

```
Prelude> 2 ^ 3 ^ 4
2417851639229258349412352
Prelude> 2 ^ (3 ^ 4)
2417851639229258349412352
```

[۱] Exponentiation

```
Prelude> (2 ^ 3) ^ 4
4096
```

همونطور که می‌بینید، وقتی شرکت‌پذیری از راست داریم، اضافه کردن پرانتز از راست (دسته‌بندی از راست) تاثیری روی جواب نداره. ولی اگه از چپ پرانتز بذاریم، جواب نهایی‌مون فرق می‌کنه.

در کل، اون درکی که از تقدم، شرکت‌پذیری، و تاثیرِ پرانتزها توی ریاضیات دارین، توی هسکل هم صادقه:

```
2 + 3 * 4

(2 + 3) * 4
```

این دو تا چه فرقی با هم دارن؟ چرا فرق دارن؟

تمرین‌ها: پرانتز و شرکت‌پذیری

در زیر، هر جفتِ تابع‌ها فقط در پرانتزگذاری فرق دارن. با دقت هر کدوم رو بخونین و ببینین آیا پرانتز تغییری در نتیجه‌ی نهایی داره یا نه. جواب‌تون رو توی GHCi چک کنین.

۱. a) `8 + 7 * 9`
   b) `(8 + 7) * 9`

۲. a) `perimeter x y = (x * 2) + (y * 2)`
   b) `perimeter x y = x * 2 + y * 2`

۳. a) `f x = x / 2 + 9`
   b) `f x = x / (2 + 9)`

۲ – ۷ تعریف مقادیر

از اونجا که GHCi یه فایل منبع رو یکباره و کامل بارگذاری می‌کنه، ترتیب تعاریف (declaration) توی فایل اهمیتی نداره، GHCi همه‌ی مقادیر تعریف شده رو می‌شناسه. ولی تو REPL که یکی یکی تعاریف رو وارد می‌کنیم، ترتیب مهمه.

برای مثال، می‌تونیم یک سری از تعاریف رو توی REPL اینطوری وارد کنیم:

```
Prelude> let y = 10
Prelude> let x = 10 * 5 + y
Prelude> let myResult = x * 5
```

بالاتر، سرِ تابعِ `triple` دیدیم که تعریفِ هر چیزی توی REPL، باید با `let` باشه.

حالا اگه اسم اون مقادیر رو وارد کنیم، مقدارشون رو می‌بینیم:

```
Prelude> x
60
Prelude> y
10
Prelude> myResult
300
```

ببینیم چطوری همین مقادیر رو تو یه فایل به اسم `learn.hs` وارد کنیم. اول از همه اسمِ ماژول[۱] رو می‌نویسیم که بعداً با همین اسم بتونیم توی پروژه وارد یا import ِش کنیم (فعلاً سراغ پروژه به این بزرگی که چندتا ماژول داشته باشه نمیریم، ولی خوبه که از الان به نوشتنِ اسم ماژول‌ها عادت کنین):

[۱] Module

```haskell
-- Learn.hs

module Learn where

x = 10 * 5 + y

myResult = x * 5

y = 10
```

اسم ماژول‌ها با حروف بزرگ شروع میشه. برای اسم متغیرها، camelCase نوشتیم: حرف اول کوچیکه، ولی هر جا به لغتِ بعد رسیدیم، با حرف بزرگ شروع‌ش کردیم. این کار، خوندنِ اسم‌ها رو راحت‌تر می‌کنه.

عیب‌یابی

در نوشتنِ برنامه‌ای مثل `learn.hs` اشکالات زیادی ممکنه پیش بیاد. تو این بخش به چندتا از اشکالات رایج در کدنویسیِ هسکل نگاه میندازیم. یکی از چیزایی که باید حواستون بهش باشه، اینه که

توگذاری[۱] در کدِ هسکل مهمه و می‌تونه معنای کد رو تغییر بده. حتی می‌تونه کدتون رو از کار بندازه. برای توگذاری هم فقط از کاراکترِ space استفاده کنین، اصلاً کاراکترِ tab رو وارد کدتون نکنین.

کلاً فاصله یا whitespace در هسکل حائز اهمیته. استفاده‌ی به‌جا از اون، به جمع و جور شدنِ کدتون هم کمک می‌کنه. اگه با زبان دیگه‌ای تا الان کد می‌نوشتین، ممکنه اولش خیلی با این وابستگی به whitespace کنار نیاین. تنها راهِ اعمال توابع، whitespace ِه (به غیر از وقت‌هایی که به خاطر تلاقیِ تقدمِ توابع از پرانتز برای اعمالِ توابع استفاده می‌کنیم). اضافه کردنِ whitespace آخرِ خطِ کد[۲]، وِجهه‌ی خوبی نداره.

اکثر مواقع در فایل‌های هسکل، توگذاری جای علائم گرامری مثل آکولاد[۳]، نقطه ویرگول[۴]، و پرانتز رو می‌گیره. قاعده‌ی کلی اینه که اگه

---

[۱] Indentation

[۲] Trailing Whitespace

[۳] Curly Brackets

[۴] Semicolons

کُدی بخشی از یه بیانیه‌ی دیگه‌س، باید زیر اون بیانیه توگذاری یا indent بشه، حتی اگه اون بیانیه در چپ‌ترین ستونِ متن نباشه. همچنین، اون بخشهایی از کد که با هم دسته‌بندی میشن، باید همه‌شون به یه اندازه توگذاری بشن. برای مثال، در یه بلوکِ کد که با `let` یا `do` شروع شده، چنین توگذاری‌ای صحیحه:

```
let
  x = 3
  y = 4

-- یـا

let x = 3
    y = 4
```

البته این کد فقط در صورتی توی یه فایل صحیحه، که زیرمجموعه‌ی یک تعریفِ سطح بالاتر باشن (م. مثلاً یه تابع).

دقت کنید اون دو تا تعریف‌هایی که جزء یک بیانیه‌اند، از یک ستون شروع شدن. اینطوری نوشتن درست نیست:

```
let x = 3
 y = 4

-- یـا

let
 x = 3
  y = 4
```

اگه بیانیه‌تون از چند بخش تشکیل شده، مثل زیر توگذاری کنین:

```
foo x =
    let y = x * 4
        z = x ^ 2
    in 2 * y * z
```

می‌بینید که تعریف‌های y و z از یک ستون شروع شدن، `let` و `in` هم در یه راستا اند. در طولِ کتاب به طریقه‌ی توگذاریِ ما دقت کنین. خیلی پیش میاد که یه توگذاریِ نابجا کد رو خراب کنه. اگه کد رو از جایی کپی کنین، احتمال توگذاریِ اشتباه خیلی میره بالا.

اگه اشتباهی وسط یه تعریف برین خط بعد، طوری که ادامه‌ی بیانیه از اول خط شروع بشه:

```
module Learn where
-- تعریف ماژول در ابتدای فایل

x = 10
* 5 + y

myResult = x * 5

y = 10
```

یه پیغام خطا مثل این میاد:

```
Prelude> :l code/learn.hs
[1 of 1] Compiling Learn

code/learn.hs:10:1:
  parse error on input ‘*’
Failed, modules loaded: none.
```

اولین خطِ پیغام خطا، جایی که اشتباه پیش اومده رو میگه: `code/learn.hs:10:1:` به خطا در خطِ ۱۰ و ستونِ ۱ از فایل `learn.hs` در پوشه‌ی `code` اشاره می‌کنه. پیدا کردنِ عاملِ error با

چنین پیغامی خیلی آسون‌تر میشه. البته واضحه که خط و ستون خطای شما ممکنه جای دیگه‌ای باشن.

برای تصحیح این خطا، یا می‌تونین همه‌ش رو توی یه خط بنویسین:

```
x = 10 * 5 + y
```

یا دقت کنین اگه وسط بیانیه رفتین خط بعد، حتماً اون خط رو حداقل با یک فاصله از اون جایی که خط قبل شروع شده، بنویسین (هر دو مثالِ زیر کار می‌کنن):

```
x = 10
 * 5 + y

-- یا

x = 10
    * 5 + y
```

دومی یه کم بهتره. در کل سعی کنید فقط وقتی وسط جمله برین خط بعد که از ۱۰۰ ستون رد شدین.

خطای دیگه‌ای که ممکه رخ بده، اینه که یه تعریف رو از اولِ خط (سمت چپ) شروع نکنین:

```
-- Learn.hs

module Learn where

 x = 10 * 5 + y

myResult = x * 5

y = 10
```

اون فاصله رو قبل از x می‌بینین؟ همون باعثِ چنین پیغام خطایی شده:

```
Prelude> :l code/learn.hs
[1 of 1] Compiling Learn

code/learn.hs:11:1:
  parse error on input ‘myResult’
Failed, modules loaded: none.
```

یه کم گیج کننده‌ست... پیغام به `myResult` اشاره کرده، ولی خطا جای دیگه‌اییه. خطا فقط به خاطر یه فاصله‌ی اضافه‌ست؛ ولی همه‌ی تعاریف در ماژول باید از یک ستون شروع شن، و اون ستون توسط اولین تعریف در ماژول تعیین میشه. در این مثال، دلیل اینکه پیغامِ خطا به جای دیگه‌ای اشاره می کنه، اینه که ستونی که تعاریف باید ازش شروع بشن رو همون ستونی میدونه که تعریفِ x ازش شروع شده. پس از دیدِ کامپایلر[۱]، تعریفِ `myResult` یک ستون زود شروع شده.

یک راه اینه که تعاریف y و `myResult` رو یه ستون بیاریم جلو:

```
-- Learn.hs
module Learn where

  x = 10 * 5 + y

  myResult = x * 5

  y = 10
```

---

[۱] م. کامپایلر یا compiler یک واسط نرم‌افزاریه که یک زبان برنامه‌نویسی رو به یک زبان برنامه‌نویسی دیگه تبدیل می‌کنه.

ولی منظره‌ی خوبی نداره، و کدنویسیِ استاندارد هسکل نیست. تقریباً هیچ وقت پیش نمیاد که لازم بشه کلِ کد رو توگذاری یا indent کنین، ولی کمک می‌کنه رَوشِ کارِ کامپایلر رو بهتر بشناسین. نتیجه‌ای که می‌گیریم اینه که هر وقت چنین پیغام خطایی دیدین، مطمئن شین که اولین تعریف‌تون از چپ‌ترین نقطه‌ی خط شروع میشه، بعد دنبال خطاهای دیگه باشین.

یه اشتباه دیگه که ممکنه پیش بیاد، اینه که موقعِ نوشتن کامِنت[۱] (م. یا توضیحاتی که کامپایلر نمی‌خونه)، یکی از خطِ فاصله‌های اولش رو (--) نذارین.

پس این کد:

[۱] Comment

```
- learn.hs

module Learn where

x = 10 * 5 + y

myResult = x * 5

y = 10
```

چنین خطایی میده:

```
code/learn.hs:7:1:
  parse error on input ‘module’
Failed, modules loaded: none.
```

باز هم خطای پارس (parse)، جای دیگه‌ای اشاره شده. اشکال در واقع کم بودنِ یک خطِ تیره اولِ کُدِ ه. کامنت‌های یک خطی در هسکل با دو تا خط تیره (--) شروع میشن.

حالا ببینیم چه طوری از کُدی که توی فایل نوشته شده در GHCi استفاده کنیم. با فرض اینکه REPL رو از همون پوشه‌ای که `learn.hs` توشه اجرا کنیم، کارهای زیر رو میشه انجام داد:

```
Prelude> :l learn.hs
[1 of 1] Compiling Learn
Ok, modules loaded: Learn.
Prelude> x
60
Prelude> y
10
Prelude> myResult
300
```

تمرین‌ها: بیمار رو دریاب

مثال‌های زیر خرابن و کامپایل نمیشن. دوتا اولی برای نوشتنِ مستقیم توی REPL طراحی شدن؛ آخری توی یه فایله. ایرادات‌شون رو پیدا کنید و کاری کنین کامپایل بشن.

۱.
```
let area x = 3. 14 * (x * x)
```

۲.
```
let double x = b * 2
```

۳.
```
x = 7
 y = 10
f = x + y
```

۲ – ۸ توابع عددی در هسکل

تو این بخش به بررسی چندتا از توابع و عملگرهای پایه‌ایِ حساب می‌پردازیم، که از این قرارند:

عملگر	اسم	کاربرد
`+`	به علاوه	جمع
`-`	منها	تفریق
`*`	ستاره[۱]	ضرب
`/`	خط مورب	تقسیم کسری
`div`	تقسیم[۲]	تقسیم integral، گرد به پایین
`mod`	پیمانه[۳]	باقیمانده از تقسیم پیمانه‌ای یا ماژولار
`quot`	خارج‌قسمت[۴]	تقسیم integral، گرد به سمت صفر
`rem`	باقیمانده[۵]	باقیمانده از تقسیم

[۱] Asterisk

[۲] Divide

[۳] Modulo

[۴] Quotient

[۵] Remainder

منظور از تقسیمِ integral، تقسیمِ اعدادِ صحیحه. اون توابعْ اعدادِ کسری نمی‌گیرن و فقط اعداد صحیح می‌گیرن. به همین خاطر هم جوابشون گرد میشه.

در زیر از هر کدوم یه مثال تو REPL زدیم:

```
Prelude> 1 + 1
2
Prelude> 1 - 1
0
Prelude> 1 * 1
1
Prelude> 1 / 1
1.0
Prelude> div 1 1
1
Prelude> mod 1 1
0
Prelude> quot 1 1
1
Prelude> rem 1 1
0
```

به خاطرِ طریقه‌ی گرد کردنِ div و quot، کاربردِ div برای تقسیمِ صحیح بیشتره:

```
-- به پایین گرد میشه
Prelude> div 20 (-6)
-4

-- به سمت صفر گرد میشه
Prelude> quot 20 (-6)
-3
```

کاربرد rem و mod هم یه کم با هم فرق دارن؛ تو این فصل mod رو با جزئیاتِ بیشتری بررسی می‌کنیم. تقسیم کسری (/) رو تو یه فصل جلوتر توضیح میدیم، چون قبلش باید کمی با تایپ‌ها و تایپکلاس[۱]ها آشنایی پیدا کنیم.

[۱] Typeclass

قوانین خارج‌قسمت و باقیمانده

در برنامه‌نویسی، اکثراً توابعی که برای تقسیم و باقیمانده وجود دارن بیشتر از ریاضیه، و برای درک بهتر، آشنایی با قوانین مرتبط با quot و rem، و div و mod خیلی کمک می‌کنه.[۱] اینجا یه نگاه بهشون میندازیم.

```
(quot x y)*y + (rem x y) == x

(div x y)*y + (mod x y) == x
```

مراحل اثبات این دو تا رو اینجا نمی‌نویسیم، ولی می‌تونیم با مثال تا حدی مفهوم‌شون رو نشون بدیم:

```
(quot x y)*y + (rem x y) == x

با فرض x برابر ۱۰ و y برابر (۴-)

(quot 10 (-4))*(-4) + (rem 10 (-4))
```

---

[۱] گرفته شده از بلاگ لنارت آگوستسون (Lennart Augustsson) به آدرس: http://augustss.blogspot.com یا جواب سؤال در سایت Stack Overflow با آدرس: http://stackoverflow.com/a/8111203.

```
quot 10 (-4) == (-2) و rem 10 (-4) == 2

(-2)*(-4) + (2) == 10

10 == x
```

به جوابی که می‌خواستیم رسیدیم.

حالا برای div و mod:

```
(div x y)*y + (mod x y) == x
```

با فرض x برابر ۱۰ و y برابر (۴-)

```
(div 10 (-4))*(-4) + (mod 10 (-4))

div 10 (-4) == (-3) و mod 10 (-4) == -2

(-3)*(-4) + (-2) == 10

10 == x
```

به نتیجه میرسیم که در دنیای تقسیم اعداد صحیح، همه چیز خوب و خوشه.

استفاده از mod

ما اینجا به توضیح کامل حساب پیمانه‌ای[۱] نمی‌پردازیم، ولی می‌خوایم یه درکِ کلی از mod به کسانی که باهاش آشنایی ندارن بدیم. و اینکه دقیقاً تو هسکل چطوری کار می‌کنه.

بالاتر توی جدول گفتیم که mod باقیمانده‌ی تقسیم ماژولار رو میده. بدون آشنایی با تقسیم ماژولار، تفاوتِ پرفایده‌ی بین mod و rem خیلی واضح نمیشه.

حساب پیمانه‌ای یا ماژولار یک سیستمی از محاسبات اعداد صحیحه که اعداد بعد از رسیدن به یه مقدار معلوم (به اسم *مدول*[۲])، "دور میزنن" به مقدار اولیه. مثال رایج برای توضیح این سیستم، ساعته.

وقتی زمان رو بر حسب یه ساعت ۱۲-ساعته می‌شماریم، بعد از ساعت ۱۲ باید شمارش‌مون رو از اول شروع کنیم. برای مثال اگه الان

---

[۱] Modular Arithmetic

[۲] Modulus

ساعت ۸:۰۰ باشه، و بخوایم بدونیم ۸ ساعت دیگه ساعت چند میشه، نمیگیم ۸ + ۸، پس ساعت میشه ۱۶/[۱]

به جای اون کار، هر ۱۲ ساعت شمارش رو از اول شروع می‌کنیم. پس اگه بخوایم ۸ ساعت به ۸:۰۰ اضافه کنیم، اول ۴ ساعت اضافه می‌کنیم تا به ۱۲ برسیم، و بعد از اول شروع می‌کنیم، انگار ۱۲ همون صِفره، و ۴ ساعت باقیمانده از ۸ ساعت‌مون رو اضافه می‌کنیم تا به ساعت ۴:۰۰ برسیم. بنابراین، ۸ ساعت بعد از ۸:۰۰، ساعت میشه ۴:۰۰.

به چنین چیزی میگیم پیمانه‌ی[۲] ۱۲. در یک ساعت ۱۲-ساعته، ۱۲ با خودش و صفر برابره، و به نوعی ساعت ۱۲:۰۰ ساعت ۰:۰۰ هم هست. منظور از پیمانه‌ی ۱۲ اینه که ۱۲ هم ۱۲ِه، و هم صفر.

برای چنین کاربردی، بیشترِ مواقع `mod` و `rem` یه جواب میدن:

---

[۱] البته در ساعت‌های ۲۴-ساعته، ساعت ۱۶:۰۰ درسته و معنی میده؛ ولی اگه ۸ ساعت بعد از ۸:۰۰ بعدازظهر رو بخوایم، جواب ۱۶:۰۰ صبح نمیشه. مدول در ساعت‌های ۲۴-ساعته با ساعت‌های ۱۲-ساعته متفاوته.

[۲] Modulo

```
Prelude> mod 15 12
3
Prelude> rem 15 12
3

Prelude> mod 21 12
9
Prelude> rem 21 12
9

Prelude> mod 3 12
3
Prelude> rem 3 12
3
```

دو تا آخری شاید عجیب باشن... rem و mod فقط با تقسیم صحیح کار می‌کنن، پس جوابِ تقسیم با یه عدد بزرگتر، مساوی میشه با صفر، به اضافه‌ی باقیمانده‌ای که برابر با همون عددِ کوچکتر (صورت کسر یا مقسوم) هست. هر وقت بخوایم از تقسیم با یه عددِ بزرگتر جواب کسری بگیریم، از (/) استفاده می‌کنیم و دیگه باقیمانده هم نداریم.

فرض کنید یه تابع بخوایم که به ما بگه چند روز بعد یا قبل از امروز، چه روزی از هفته میشه. اول باید هر روزِ هفته رو با یه عدد معادل کنیم، پس شنبه رو با صفر نشون میدیم[۱]. اگه امروز یکشنبه باشه، و بخوایم بدونیم ۲۳ روز دیگه چند شنبه‌ست، چنین کاری جواب میده:

```
Prelude> mod (1 + 23) 7
3
```

عدد ۱ برای یکشنبه‌س (روز مبدأ)، ۲۳ هم تعداد روزیه که می‌خوایم اضافه کنیم. وقتی با mod عدد ۷ حساب کنیم، عددی رو برمی‌گردونه که نشون دهنده‌ی روز مقصده (طبق شماره‌گذاری خودمون).

پنج روز بعد از جمعه هم میشه چهارشنبه:

```
Prelude> mod (6 + 5) 7
4
```

---

[۱] ممکنه براتون عادی‌تر باشه که روزهای هفته رو از ۱ تا ۷ بشمرین، ولی برنامه‌نویس‌ها دوست دارن از صفر بشمارن.

اگه از rem استفاده کنیم، جوابی می‌گیریم که ظاهراً یکسانه:

```
Prelude> rem (1 + 23) 7
3
```

ولی اگه بخوایم بدونیم چند روز *قبل* چند شنبه بوده، جواب mod و rem فرق می‌کنن. ببینیم اگه امروز سه‌شنبه باشه، ۱۲ روز قبلش چند شنبه بوده:

```
Prelude> mod (3 - 12) 7
5
Prelude> rem (3 - 12) 7
-2
```

جواب mod درسته، ولی rem جوابی که می‌خوایم رو نمیده.

فرق کلیدیِ بین این دو تابع در هسکل (نه همه‌ی زبان‌ها) اینه که جوابِ mod همیشه با مقسوم‌علیه (مخرج کسر) هم‌علامته، ولی rem همیشه با مقسوم (صورت کسر) علامتِ یکسان داره:

```
Prelude> (-5) `mod` 2
1
Prelude> 5 `mod` (-2)
-1
Prelude> (-5) `mod` (-2)
-1
```

ولی:

```
Prelude> (-5) `rem` 2
-1
Prelude> 5 `rem` (-2)
1
Prelude> (-5) `rem` (-2)
-1
```

با کمی تجربه میشه تشخیص داد باید کجا از کدومشون استفاده کرد.

اعداد منفی

به خاطر تعامل پرانتزها، currying، و گرامرِ میانوندی (infix syntax) در هسکل، اعداد منفی شرایط خاصی پیدا می‌کنن.

وقتی فقط یه عدد منفی به تنهایی می‌خواین، کد زیر موردی نداره:

```
Prelude> -1000
-1000
```

ولی بعضی مواقع جواب نمیده:

```
Prelude> 1000 + -9
<interactive>:3:1
    Precedence parsing error
        cannot mix ‘+’ [infixl 6] and
        prefix `-` [infixl 6]
           in the same infix expression
```

خوشبختانه قبل از اجرای کد، خطا مون پیدا شد. دقت کنین که پیغام خطا از تقدم ایراد گرفته. جمع و تفریق هر دوشون تقدمِ یکسان (۶) دارن، و GHCi هم فکر می‌کنه که ما می‌خواستیم جمع کنیم بعد تفریق، نه اینکه با یه عددِ منفی جمع کنیم، به همین خاطر نمیدونه چطور تلاقیِ تقدم‌ها رو حل کنه و جواب بده. یه تغییراتی لازمه تا بتونیم اعداد مثبت و منفی رو با هم جمع کنیم:

```
Prelude> 1000 + (-9)
991
```

منفی کردن اعداد با یک مِنها (-) در هسکل، نوعی شکرِ گرامری[۱] ه. گرامر یا syntax، ساختارِ نوشتاریه که باهاش برنامه رو بیان می‌کنیم؛ و syntactic sugar راهی برای ساده‌تر کردن این متن‌ها برای خوندن و نوشتنه. شکر گرامری هیچ تأثیری در معنا و مفهوم برنامه‌ها نداره، فقط خوندن یا نوشتن کد رو راحت‌تر می‌کنه، روش حل مسائل با کد رو هم تغییری نمیده. معمولاً اگه در کُدی که به REPL یا کامپایلر میدیم از این نوع شکرها وجود داشته باشه، بعد از پارس (parse) شدن، یه تبدیلِ جزئی از فرم کوتاه (یا "شیرین‌تر") به فرم گویاتر روی کد انجام میشه.

در مورد این شکر گرامری، منها می‌تونه دو معنی داشته باشه: یکی اینکه مستعاری از تابعِ `negate` باشه، یا اینکه تابعِ تفریق باشه. در زیر، هر دو مثال دقیقاً معادل‌اند، چون - به `negate` ترجمه میشه:

```
Prelude> 2000 + (-1234)
766
```

[۱] Syntactic Sugar

```
Prelude> 2000 + (negate 1234)
766
```

ولی اینجا - برای تفریق استفاده شده:

```
Prelude> 2000 - 1234
766
```

خوشبختانه تو هسکل، گرامرِ سربار مثل این زیاد نیست.

۲ – ۹ پرانتزگذاری

اینجا مشخصاتی که GHCi برای چندتا از عملگرهای میانوند با دستور `:info` میده رو لیست کردیم. تایپ سیگنچرها رو هم نوشتیم، ولی فعلاً کاری باهاشون نداریم. شاید دیدنشون برای کسانی که کنجکاون جذاب باشه.

```
Prelude> :info (^)
(^) :: (Num a, Integral b) => a -> b -> a
infixr 8 ^
```

```
Prelude> :info (*)
class Num a where
  (*) :: a -> a -> a
infixl 7 *

Prelude> :info (+)
class Num a where
  (+) :: a -> a -> a
infixl 6 +

Prelude> :info (-)
class Num a where
  (-) :: a -> a -> a
infixl 6 -

Prelude> :info ($)
($) :: (a -> b) -> a -> b
infixr 0 $
```

اوپراتور ($) رو تو هسکل زیاد می‌بینید، ما هم اینجا یه کم روش وقت میذاریم. خبر خوب اینکه تقریباً هیچ کاری نمی‌کنه. خبر بد اینکه همین موضوع خیلی‌ها رو سردرگم می‌کنه.

اول از همه تعریف‌ش رو ببینیم:

```
f $ a = f a
```

در نگاه اول شاید بی‌فایده به نظر برسه، ولی دقت کنین که این یه عملگرِ میانوند با کمترین تقدمِ ه. این اوپراتور وقتی فایده داره که میخواین کمتر پرانتز بنویسین:

```
Prelude> (2^) (2 + 2)
16
-- میتونه جای اون پرانتزها رو بگیره
Prelude> (2^) $ 2 + 2
16
-- $ بدون پرانتز یا
Prelude> (2^) 2 + 2
6
```

اوپراتور ($) اجازه میده اول هر چیز سمت راستش هست حساب بشه، و میشه برای تعویقِ اعمالِ توابع ازش استفاده کرد. در فصل ۷، وقتی ترکیب توابع رو توضیح بدیم، منظورمون از تعویقِ توابع روشن‌تر میشه.

توی یه بیانیه از چندتا ($) هم میشه استفاده کرد. برای مثال:

```
Prelude> (2^) $ (+2) $ 3*2
256
```

ولی این کار نمی‌کنه:

```
Prelude> (2^) $ 2 + 2 $ (*30)
```

یه پیغام خطای طولانی و زشت در مورد تایپ‌ها میده و از اعمال اعداد به آرگومان (مثل توابع) ایراد می‌گیره. اگه مرحله به مرحله ساده کنیم، می‌فهمیم چرا این کد کار نمی‌کنه:

```
-- تعریف ($)
f $ a = f a

(2^) $ 2 + 2 $ (*30)
```

چون $ شرکت‌پذیری از راست (infixr) داره، ساده‌سازی رو از راست‌ترین نقطه شروع می‌کنیم.

```
2 + 2 $ (*30)
-- رو ساده می‌کنیم ($)
(2 + 2) (*30)
```

قبل از اعمال (2 + 2) باید ساده‌ش کنیم:

```
4 (*30)
```

خوب، جواب شد (4 * 30)، درسته؟ *نه!* این بیانیه میخواد عدد ۴ رو مثل یه تابع به (*30) اعمال کنه! و این هیچ مفهومی نداره. به نوشتنِ بیانیه‌ها به این شکل (*30) میگیم بخش‌بندی[۱].

این مثال رو یه ذره جابجا می‌کنیم تا جواب بده، بعد مراحل ساده شدنش رو می‌بینیم:

```
(2^) $ (*30) $ 2 + 2
-- اول باید سمت راست رو حساب کنیم
(2^) $ (*30) (2 + 2)
-- اعمال تابع (*30) به بیانیه‌ی (2 + 2)
-- محاسبه‌ش رو اجبار می‌کنه
(2^) $ (*30) 4
-- حالا 4 (*30) رو ساده می‌کنیم
(2^) $ 120
-- دوباره ($) رو ساده می‌کنیم
(2^) 120
```

[۱] Sectioning

```
(2^) رو ساده می‌کنیم --
132922799578491587290380706028034457 6
```

بعضی هسکل نویس‌ها پرانتز رو از علامت دلار (dollar sign) خواناتر میدونن، ولی استفاده‌ش انقدر رایجه، که لازمه حداقل باهاش آشنا شده باشین.

پرانتزگذاری عملگرهای میانوند

گاهی پیش میاد که فقط با خودِ یه اوپراتور میانوند کار داشته باشیم (بدون آرگومان)، و گاهی هم به عنوان تابع پیشوندی لازم میشن. در هر دو مورد باید عملگر رو بین پرانتز بذاریم. با یه مثال استفاده از عملگرها به عنوان تابع پیشوندی رو نشون میدیم.

اگه تابعِ میانوندی‌تون << باشه، هر وقت بخواین به عنوان یه مقدار (value) ازش استفاده کنین باید با پرانتز بنویسین، (<<). با استفاده از پرانتز، (+) یه تابعِ جمعه که هیچ آرگومانی بهش داده نشده، و (+1) همون تابعه که به یکی از آرگومان‌هاش اعمال شده. پس (+1) یه تابعه که ورودی‌ش رو با ۱ جمع می‌کنه و جواب رو خروجی میده:

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

موردِ آخر مثالی از بخش‌بندی یا sectioningِ ه، و یکی از راه‌های استفاده از توابعِ نیمه اعمال‌شده[۱] ست. به دلیل خاصیت جابجایی‌پذیریِ تابع جمع، تفاوتی بین (1+) و (+1) وجود نداره، یعنی ترتیب آرگومان‌ها تأثیری روی جواب نداره.

در مقابل، اگه از بخش‌بندی با توابعی که جابجایی‌پذیر[۲] نیستن استفاده کنیم، ترتیب اهمیت پیدا می‌کنه:

```
Prelude> (1/) 2
0.5
Prelude> (/1) 2
2.0
```

---

[۱] Partially Applied

[۲] Commutative

تفریق یه مورد خاصه. اینها کار می‌کنن:

```
Prelude> 2 - 1
1
Prelude> (-) 2 1
1
```

ولی اینطور بخش‌بندی کار نمی‌کنه:

```
Prelude> (-2) 1
```

وقتی یه مقدار رو به همراه منها تو پرانتز میذاریم، GHCi اون رو به چشم یک آرگومان برای توابع می‌بینه. و از اونجا که منها بعد از اعمال شدن به آرگومان دوم‌ش، به تابع `negate` تبدیل میشه، GHCi پیغام خطا میده که نمی‌تونه مقدار ۲- رو به عدد ۱ اعمال کنه. منها در این مورد، یه نمونه از سرباری گرامری[۱] ِه که باعثِ کمی ابهام میشه.

از بخش‌بندی یا sectioning برای تفریق میشه استفاده کرد، ولی فقط با آرگومانِ اول‌ش:

---

[۱] Syntactic Overloading

```
Prelude> let x = 5
Prelude> let y = (1 -)
Prelude> y x
-4
```

راه دیگه اینکه به جای (x -) بنویسین (subtract x):

```
Prelude> (subtract 2) 3
1
```

شاید کاربرد اینها الان واضح نباشه، ولی چنین گرامری رو در طول کتاب باز هم می‌بینید، برای مثال وقتی توابع رو به تک‌تکِ مقادیر یه لیست (یا یه ساختارِ داده) اعمال می‌کنیم، بخش‌بندی کاربرد داره.

۲ – ۱۰ let و where

در کد هسکل از let و where برای معرفیِ بخشهایی از یک بیانیه، زیاد استفاده میشه، و خیلی هم شبیهِ هم هستن. برای استفاده‌ی به‌جا از هر کدوم، کمی تمرین لازمه.

تفاوت‌شون اینجاست که let یک *بیانیه* یا *expression* رو معرفی می‌کنه، پس هر جایی که بشه بیانیه نوشت قابل جاگذاری‌ه. ولی where یک *تعریف* یا *declaration*ِ ه، و به سازه‌یِ گرامری اطرافش مقیّده.

با یه مثال از where شروع می‌کنیم:

```
-- FunctionWithWhere.hs
module FunctionWithWhere where

printInc n = print plusTwo
  where plusTwo = n + 2
```

و اگه بیاریمِش تو REPL:

```
Prelude> :l FunctionWithWhere.hs
[1 of 1] Compiling FunctionWithWhere ...
Ok, modules loaded: FunctionWithWhere.
Prelude> printInc 1
3
Prelude>
```

و حالا همون تابع رو می‌نویسیم، اما این دفعه با let:

```haskell
-- FunctionWithLet.hs
module FunctionWithLet where

printInc2 n = let plusTwo = n + 2
              in print plusTwo
```

هر وقت let بعدِش in بیاد، میشه یک *بیانیه‌ی let* یا *let expression*. تابعِ دوم در REPL:

```
Prelude> :l FunctionWithLet.hs
[1 of 1] Compiling FunctionWithLet ...
Ok, modules loaded: FunctionWithLet.
Prelude> printInc2 3
5
```

اگه FunctionWithLet رو بعد از FunctionWithWhere بارگذاری کردین، REPL قبلی رو تخلیه یا unload کرده:

```
Prelude> :l FunctionWithWhere.hs
[1 of 1] Compiling FunctionWithWhere ...
Ok, modules loaded: FunctionWithWhere.
Prelude> printInc 1
3
```

```
Prelude> :l FunctionWithLet.hs
[1 of 1] Compiling FunctionWithLet ...
Ok, modules loaded: FunctionWithLet.
Prelude> printInc2 10
12
Prelude> printInc 10

<interactive>:6:1:
    Not in scope: ‘printInc’
    Perhaps you meant ‘printInc2’ (line 4)
--  ^--------------------------^
-- م. شاید منظورتون ‘printInc2’ بوده --
```

وقتی با دستورِ load: فایل FunctionWithLet.hs رو بارگذاری کردین، GHCi هر چیزی رو که قبل از اون، تعریف یا لود کرده بودین رو تخلیه (unload) کرد. به همین خاطر تابعِ printInc از گستره[۱]ش خارج شده بود. منظور از گستره یا scope حوزه‌ای از کده، که انقیادِ یک متغیر اونجا اعمال میشه.

[۱] Scope

این یکی از محدودیتهای دستور `:load` در GHCiِ ه. وقتی شروع به ساخت پروژه‌های بزرگتر کنیم که لازم باشه چندتا ماژول رو در گستره داشته باشن، به جای GHCi از یک ابزار مدیریتِ پروژه به نام Stack استفاده می‌کنیم.

تمرین‌ها: کد ذهنی

وقتِ تمرینه. اول جواب بیانیه‌های زیر رو ذهنی پیدا کنید، بعد با REPL چک کنید:

۱. `let x = 5 in x`

۲. `let x = 5 in x * x`

۳. `let x = 5; y = 6 in x * y`

۴. `let x = 3; y = 1000 in x + 3`

الان این چندتا بیانیه‌ی `let` رو تو REPL تایپ کردین. حالا می‌خوایم یه فایل باز کنیم و چندتا بیانیه‌ی `let` رو با استفاده از تعاریف `where` بازنویسی کنیم. برای مقادیری که انقیاد یا bind می‌کنین، یه

اسم هم باید اختصاص بدین (که اگه بخواین میشه تک حرفی باشن). برای مثال:

```
-- این تو GHCi کار می‌کنه
let x = 5; y = 2 in x * y
```

میشه با where بازنویسی کرد:

```
-- این رو توی فایل بنویسین
mult1      = x * y
   where x = 5
         y = 6
```

اونطوری تساوی‌ها رو ردیف کردن، یه انتخاب سلیقه‌ایه. فقط کافیه چنین توگذاری‌ای رعایت بشه، لازم نیست مساوی‌ها زیر هم باشن. دقت کنین که ما یه اسم تعریف کردیم (mult1)، و با استفاده از اون اسم در REPL به مقدارش دسترسی پیدا می‌کنیم:

```
Prelude> :l practice.hs
[1 of 1] Compiling Main
Ok, modules loaded: Main.
*Main> mult1
30
```

توجه کنید که prompt از Prelude به Main* تغییر می‌کنه، این نشون میده یه ماژول به اسم Main رو بارگذاری کردین.

با where بازنویسی کنید:

۱.
```
let x = 3; y = 1000 in x * 3 + y
```

۲.
```
let y = 10; x = 10 * 5 + y in x * 5
```

۳.
```
let x = 7
    y = negate x
    z = y * 10
in z / x + y
```

اسم فایلی که انتخاب می‌کنین مهم نیست، فقط باید پسوند .hs داشته باشه.

۲ – ۱۱ تمرین‌های فصل

هدف کلی از تمرین‌های این فصل اینه که باعث بشه یه کم با کدها بازی کنین و اینکه سعی کنین قبل از اجرا، نتیجه رو پیش‌بینی کنید. کد رو با دقت بخونید، و به کمک مطالبی که تا اینجا خوندیم، یه فرضیه از

کاری که فکر می‌کنین کد باید انجام بده درست کنید. با REPL هم بازی کنین و ببینین کجاها فرضیه‌هاتون درست و کجاها غلط بودن.

پرانتزگذاری

با توجه به تقدم توابعِ (*)، (+)، (-)، و (^)، طوری پرانتزگذاری کنید که بیانیه‌های زیر واضح[۱]تر بشن ولی نتیجه‌شون تغییری نکنه. اول ذهنی به یه جواب برسین، بعد تو GHCi REPL تست کنید.

مثلاً می‌خوایم این رو کمی واضح‌تر بنویسیم:

```
2 + 2 * 3 - 3
```

جواب‌ش با این پرانتزها تغییری نمی‌کنه:

```
2 + (2 * 3) - 3
```

همین کار رو برای بیانیه‌های زیر هم انجام بدین:

۱. `2 + 2 * 3 - 3`

---

[۱] Explicit

۲. `(^) 10 $ 1 + 1`

۳. `2 ^ 2 * 4 ^ 5 + 1`

بیانیه‌های معادل

کدوم یکی از جفت بیانیه‌های زیر جواب یکسان دارن؟ سعی کنید اول ذهنی به جواب برسین بعد تو REPL وارد کنین:

۱. `1 + 1`

```
2
```

۲. `10 ^ 2`

```
10 + 9 * 10
```

۳. `400 - 37`

```
(-) 37 400
```

۴. `100 `div` 3`

```
100 / 3
```

۵. `2 * 5 + 18`

`2 * (5 + 18)`

تفریح بیشتر با تابع‌ها

کد زیر، به فرض، تو یه فایل نوشته شده. اگه یادتون باشه، ترتیبِ کد تو فایل بی‌اهمیته، ولی وقتی مستقیماً در REPL وارد می‌کنید، ترتیب مهمه. حالا با توجه به این موضوع، کد زیر رو برای تایپ مستقیم در REPL بازنویسی کنید (یادتون باشه که ممکنه نوشتن `let` لازم باشه). حتماً کدتون رو در REPL تست کنید.

```
z = 7

x = y ^ 2

waxOn = x * 5

y = z + 8
```

۱. حالا که یه مقدار به اسم `waxOn` دارین، فکر می‌کنید بیانیه‌های زیر چه جواب‌هایی بدن:

```
10 + waxOn
-- یا
(+10) waxOn
-- یا
(-) 15 waxOn
-- یا
(-) waxOn 15
```

۲. قبل‌تر یه تابع به اسم `triple` داشتیم. همینطور که `waxOn` رو در گستره‌ی REPL دارین، دوباره `triple` رو تعریف کنین:

```
let triple x = x * 3
```

۳. فکر می‌کنین نتیجه‌ی کد زیر در GHCi چی باشه؟ نقش `waxOn` اینجا چیه؟ اول سعی کنید به جواب برسین بعد با REPL چک کنید و مطمئن بشین که دلیل جواب نهایی‌ش رو کامل درک می‌کنین:

```
triple waxOn
```

۴. توی فایل‌تون، `waxOn` رو با `where` بازنویسی کنین و مطمئن بشین همون نتیجه‌ی قبلی رو می‌گیرین.

۵. توی فایلی که `waxOn` رو تعریف کرده بودین، تابع `triple` هم اضافه کنید. یادتون باشه که `let` لازم نیست، و اسم تابع رو سمت چپ بذارین (یعنی با توگذاری زیرمجموعه‌ی `waxOn` نشه). تو REPL بارگذاری کنید و با `waxOn triple` تست کنید. جوابتون باید با جواب بالا یکسان بشه.

۶. حالا بدون اینکه تغییری به اون فایل بدین، تابع `waxOff` رو هم به فایل اضافه کنید:

```
waxOff x = triple x
```

۷. فایل رو در REPL لود کنید و تو prompt تایپ کنید:

```
.waxOff waxOn
```

می‌تونین تابع `waxOff` رو نه فقط به `waxOn`، بلکه به هر مقدارِ عددی دیگه‌ای اعمال کنید. می‌تونین یه کم باهاش بازی کنین. جواب `waxOff 10` یا `waxOff (-50)` چی میشه؟ تابع `waxOff`ِ تون رو تغییر بدین تا یه کار دیگه انجام بده، مثلاً بعد از سه برابر کردنِ x مجذورش رو برگردونه، یا به ۱۰ تقسیم

کنه. کمی وقت بذارین تا با تغییر کدتون و بارگذاری مجددِش در REPL راحت بشین، و نتیجه‌ی تغییرات‌تون رو هم ببینین.

۲ – ۱۲ تعاریف

۱. معمولاً دو لغتِ *آرگومان* و *پارامتر* به جای همدیگه استفاده میشن، ولی خوبه که فرق‌شون رو یاد بگیریم. *پارامتر*، یا پارامترِ صوری[۱]، *نماینده‌ی* مقداریه که موقع صدا زدنِ تابع، به تابع داده میشه. پس پارامترها معمولاً متغیر اند. *آرگومان* یه مقدار ورودیه که تابع بهِش اعمال میشه. وقتی تابع به یه آرگومان اعمال میشه، پارامترِ تابع به مقدارِ اون آرگومان مقیّد یا bound میشه. برای مثال، در تابعِ `f x = x + 2` که ورودیش رو با ۲ جمع می‌کنه، x تنها پارامترِ تابعه. کد رو با اعمالِ f به یه آرگومان اجرا می‌کنیم، و اگه آرگومانی که به پارامترِ x میدیم ۲ باشه، جوابمون ۴ میشه. ولی آرگومان‌ها ممکنه خودشون متغیر، یا بیانیه‌های شامل متغیر باشن، به همین خاطر همیشه فرق بین

---

[۱] Formal Parameter

آرگومان و پارامتر واضح نیست. تو این کتاب هر وقت میگیم پارامتر، همیشه منظورمون پارامترِ صوریه (معمولاً در یه تایپ سیگنچر). از طرف دیگه، لغت آرگومان رو کمی آزادانه به کار بردیم.

۲. *بیانیه* یا *expression* ترکیبی از علائمیِ‌ه که قواعد گرامری رو رعایت می‌کنه و امکان داره که به یک جواب ساده بشه. به زبان هسکل، بیانیه یک ترکیبِ با قاعده از مقادیر ثابت، متغیرها و توابع‌ه. با اینکه مقادیر ثابت در واقع بیانیه‌اند، معمولاً به "مقدار" خطاب‌شون می‌کنیم، پس وقتی میگیم *بیانیه*، معمولاً منظورمون بیانیه‌ی ساده‌شدنیه.

۳. *مقدار* یا *value* بیانیه‌ایه که ساده‌تر نمیشه. 2 * 2 بیانیه‌ست، ولی مقدار نیست. در مقابل، جوابش (که میشه ۴) یه مقداره.

۴. *تابع* یا *function* یک جسم ریاضی‌ه، که فقط می‌تونه به یه آرگومان اعمال بشه و جوابی رو برگردونه. توابع رو میشه به عنوان یه لیستی از جفتهای مرتب (ورودی و خروجیِ حاصل از

اون ورودی) هم تعریف کرد، مثل یک نگاشت. با اعمال تابعِ `f x = x + 2` به ۲، جفتِ مرتب ورودی و خروجی `(2,4)` میشه.

۵. *نوشتارِ میانوندی* یا *infix notation* سبکی از نوشتاره که در حساب[۱] و منطق[۲] به کار میره. منظور از میانوندی اینه که عملگر، بین آرگومان‌هاش قرار می‌گیره. یک نمونه از این نوع عملگرها علامت جمع در بیانیه‌ای مثل `2 + 2` است.

۶. *عملگرها* یا *operators* توابعی هستن که در حالتِ پیش‌فرض میانوندی‌اند. در هسکل، اوپراتورها باید با علائم، و نه با حروف یا اعداد، تعریف بشن.

---

[۱] Arithmetic

[۲] Logic

۷. *شکر گرامری* یا *syntactic sugar* گرامری از یک زبانِ برنامه‌نویسی‌ه که فقط برای آسون‌تر شدن خوندن و نوشتن بیانیه‌ها درست شده.

۲ – ۱۳ منابع پیشنهادی

۱. Haskell wiki article on Let vs. Where
`https://wiki.haskell.org/Let_vs._Where`

۲. How to desugar Haskell code; Gabriel Gonzalez

فصل ۳

نوشته‌ها

برنامه‌نویسی، مثل جناس، بازی
با کلمات است.
– آلان پرلیس

Like punning,
programming is a play on
words.
– Alan Perlis

۳ – ۱ چاپ نوشته‌ها

تا اینجا، بیشتر با محاسبات عددیِ ساده آشنا شدیم. در این فصل روی یک نوع دیگه از داده‌ها، به نام `String` تمرکز می‌کنیم.

بیشتر زبان‌های برنامه‌نویسی، به داده‌های نوشتاری string میگن، که معمولاً به شکل یه رشته یا لیستی از حروف[۱] نمایش داده میشن. در این فصل:

- تایپ‌ها رو معرفی اجمالی می‌کنیم، تا ساختارِ داده به اسم `String` رو بهتر بشناسیم؛
- از گرامر خاص، یا شکر گرامری مرتبط با نوشته یا string ها صحبت می‌کنیم؛
- نوشته‌ها رو در محیط REPL چاپ می‌کنیم؛
- با چندتا از توابع استانداردِ این نوع‌داده کار می‌کنیم.

---

[۱] Characters

۳ – ۲ نگاهی به تایپ‌ها

قبل از کار با نوشته‌ها، می‌خوایم اول بفهمیم این ساختارهای داده در هسکل چی هستن، و کمی با گرامر یا syntax اونها آشنا بشیم. تا اینجا چیز زیادی از تایپ‌ها نگفتیم، ولی فصل قبل چند نمونه ازشون دیدین. از مباحثِ مهمِ هسکل، همین تایپ‌ها هستن، و دو فصلِ بعدی تماماً به این موضوع پرداختن.

به طور کلی، تایپ‌ها راهی برای دسته‌بندیِ مقادیر اند. برای مثال، تایپ‌های زیادی از اعداد وجود دارن، اینکه صحیح[۱] باشن یا کسری[۲] یا انواع دیگه. یه تایپ هم برای مقادیرِ بولییَن[۳]، `True` یا `False`، وجود داره. تایپ‌هایی که ما تو این فصل باهاشون کار داریم، `Char` یا "حرف"ها، و `String` اند.

---

[۱] Integer

[۲] Fractional

[۳] Boolean

در GHCi میشه تایپِ مقادیر، بیانیه‌ها و یا توابع رو با دستورِ `:type` پیدا کرد.

REPL رو باز کنین و توی promptِ‌ش تایپ کنین: `:type 'a'`. چنین پیغامی میاد:

```
Prelude> :type 'a'
'a' :: Char
```

به چندتا چیز توجه کنیم. اول اینکه حرف رو بین دو تا آپاستروف یا single quote گذاشتیم، که مشخص می‌کنن منظور ما یه متغیر نبوده. اگه بدون آپاستروف بنویسین، `:type a`، GHCi خطا میده که a در گستره یا scope نیست. یعنی متغیرِ a تعریف نشده، پس GHCi نمی‌تونه تایپ‌ش رو بگه.

دوم، همونطور که قبلاً هم گفتیم، علامتِ `::` اینطور خونده میشه: "دارای تایپِ ... است." کلاً توی هسکل زیاد دیده میشه. هر وقت دو تا دونقطه دیدین، میدونین جلوش یه تایپ سیگنچر یا type signature

نوشته شده. تایپ سیگنچر یک خط کُده که تایپ‌های یک مقدار، بیانیه، یا تابع رو تعریف می‌کنه.

مورد آخر هم، تایپِ Char ِه. Char، تایپی‌ه که شامل حروف و اعداد، کاراکترهای یونیکُد[۱]، علائم و غیره‌ست. پس وقتی تو GHCi می‌نویسیم `:type 'a'`، در واقع می‌پرسیم "تایپِ 'a' چیه؟" و جواب GHCi، `'a' :: Char`، یعنی "'a' دارای تایپِ Char است."

حالا یه string از نوشته رو امتحان کنیم. این بار باید از علامتِ نقل قول یا double quotation mark استفاده کنیم، تا GHCi متوجه بشه که ما یه string داریم، نه یه دونه حرف:

```
Prelude> :type "Hello!"
"Hello!" :: [Char]
```

اون کروشه[۲]ها اطراف Char، شکرِ گرامری برای لیست‌اند. String، *تایپِ مستعار*[۳] یا تایپِ مترادف، برای لیستِ Char هاست. تعریفِ تایپِ

---

[۱] Unicode

[۲] Square Bracket

[۳] Type Alias

مستعار از اسمش پیداست: یه استعاره یا مترادف از یه تایپِ دیگه‌ست، که معمولاً برای سادگی تعریف میشه، ولی در باطن، همون تایپِ اصلیه. `String` هم یه اسم دیگه برای لیستِ `Char`ِه. `String` ظاهرِ کاملاً متفاوتی نسبت به لیستِ بقیه‌ی تایپ‌ها داره، که گاهی اوقات کاربردیه. وقتی لیست‌ها رو بیشتر توضیح بدیم، دلیلِ syntactic sugar بودنِ اون کروشه‌ها رو هم می‌فهمیم؛ ولی تا اینجا کافیه بدونیم که "Hello!" دارای تایپ لیستِ `Char` است.

۳ – ۳ چاپ نوشته‌های ساده

ببینیم چطور میشه string ها رو در REPL چاپ کنیم:

```
Prelude> print "hello world!"
"hello world!"
```

اینجا با `print` دستور چاپ رو به GHCi دادیم، اون هم چاپ کرد (با نقل قول‌ها دو طرفش). تابع `print` مختص string ها نیست، و برای چاپ تنوعی از داده‌ها استفاده میشه.

مثال زیر هم به GHCi دستور چاپ میده، ولی فقط برای تایپِ String کار می‌کنه:

```
Prelude> putStrLn "hello world!"
hello world!
Prelude>

Prelude> putStr "hello world!"
hello world!Prelude>
```

احتمالاً متوجهِ فرق بین putStr و putStrLn شدین. اتفاق دیگه‌ای هم که افتاد، چاپ نوشته‌ها بدون علائم نقل قول بود. دلیل این اتفاق تفاوت باطنی بین print و این دو تابعه؛ شاید ظاهراً شبیه باشن، ولی تایپ هاشون فرق داره. توابعی که ظاهر مشابه دارن، بسته به تایپ یا رده[۱]ای که بهش تعلق دارن، ممکنه رفتار متفاوتی داشته باشن.

ببینیم چطور میشه همین کارها رو توی فایل انجام بدیم. کدهای زیر رو تو یه فایل به اسم print1.hs بنویسین:

---

[۱] Category

```haskell
-- print1.hs
module Print1 where

main :: IO ()
main = putStrLn "hello world!"
```

وقتی این فایل رو تو GHCi لود کنید و main رو اجرا کنین، چنین چیزی می‌بینید:

```
Prelude> :l print1.hs
[1 of 1] Compling Print1
Ok, modules loaded: Print1.
*Print1> main
hello world!
*Print1>
```

احتمالاً مثل بالا، prompt شما هم به اسمِ ماژول تغییر کرده. اگه بخواین می‌تونین با دستورِ :module یا :m ماژول رو تخلیه (unload) کنین تا prompt دوباره Prelude بشه. کار دیگه هم که میشه کرد،

تغییر prompt به یه نوشته‌ی دلخواهه، اینطوری با load و unloadِ ماژول‌ها، تغییر نمی‌کنه[۱]:

```
Prelude> :set prompt "new prompt!! "
new prompt!! :set prompt "\x03bb> "
λ> :r
Ok, modules loaded: Print1.
λ> main
hello world!
λ>
```

برگردیم به کد، وقتی یه برنامه رو می‌سازین (build می‌کنین) یا تو REPL اجرا می‌کنین، `main` دستورالعملِ پیش‌فرض برای اون برنامه‌ست. `main` در واقع تابع نیست، بلکه معمولاً یک سِری از دستورات برای اجراست، که ممکنه شاملِ اعمال توابع و ایجاد *عوارض*[۲] هم باشند. موقعِ ساختِ پروژه با Stack، داشتنِ یه فایل به اسم `Main.hs` که توش

---

[۱] می‌تونین تغییرش رو با نوشتن اون دستور داخلِ فایلِ `~/.ghci` دائمی کنین. اگه این فایل وجود نداره، باید درستش کنید.

[۲] Side-Effects

یه دستورالعملِ[۱] `main` باشه، الزامیه. ولی همونطور که قبلاً هم دیدیم، ممکنه فایلی `main` نداشته باشه و بدون مشکل تو GHCi بارگذاری‌ش کنیم.

همونطور که می‌بینین، `main` دارای تایپِ `()` `IO` است. IO، یا I/O، مخففِ input/output یا ورودی/خروجی‌ِه. `IO` یه تایپِ خاص در هسکل‌ه، و برای اجرای برنامه‌هایی که علاوه بر محاسبه‌ی توابع یا بیانیه‌ها، شامل اثرات[۲] هم میشن به کار میره. چاپ روی صفحه یک اثر یا effectِ‌ه، پس چاپِ خروجیِ یک ماژول هم باید داخل این تایپِ `IO` پوشونده بشه. موقع‌هایی که یه تابع رو مستقیماً در REPL وارد می‌کنین، GHCi به طور ضمنی تشخیص میده و `IO` رو پیاده می‌کنه. چون اجراییه‌ی[۳] `main` دستورالعمل پیش‌فرضه، ما هم از اینجا به بعد در خیلی از فایل‌هایی که درست می‌کنیم، قرارش میدیم. در یکی از

---

[۱] Executable

[۲] Effects

[۳] Action

فصل‌های آینده، این مطالبی که گفتیم رو با جزئیات بیشتری بررسی می‌کنیم.

یه فایل دیگه درست کنیم:

```haskell
-- print2.hs
module Print2 where

main :: IO ()
main = do
  putStrLn "Count to four for me:"
  putStr   "one, two"
  putStr   ", three, and"
  putStrLn "four!"
```

گرامرِ do، یه syntax خاص‌ه، که برای تسلسلِ[1] اجراییه‌ها کاربرد داره؛ و عموماً برای سکانس کردنِ action هایی که برنامه رو تشکیل میدن (و بعضی‌هاشون الزاماً یه اثر یا effect، مثل چاپ به صفحه، رو هم

[1] Sequencing

اجرا می‌کنند) استفاده میشه. به همین خاطر، تایپِ main باید IO () باشه. گرامرِ do واجب نیست، ولی خیلی خواناتر از کدِ معادلِ‌شه.

کد بالا رو که run کنید:

```
Prelude> :l print2.hs
[1 of 1] Compling Print2
Ok, modules loaded: Print2.
*Print2> main
Count to four for me:
one, two, three, four!
*Print2>
```

برای سرگرمی، putStr ها و putStrLn ها رو با هم جابجا کنین و نتیجه‌ها رو ببینین. Ln در putStrLn یعنی بعد از چاپِ متن، خطِ جدیدی شروع می‌کنه.

الحاق نوشته‌ها

الحاق‌کردن[۱]، یعنی به هم زنجیرکردن؛ و در برنامه‌نویسی، معمولاً برای تسلسل (یا sequence) های خطی مثل لیست‌ها و رشته‌ی نوشته‌ها (strings of text) بیان میشه. اگه string های "Curry" و " Rocks!" رو به هم الحاق کنیم، حاصل‌ش میشه "Curry Rocks!". به اون فاصله اولِ " Rocks!" دقت کنین. بدونِ اون space، جواب میشه "CurryRocks!".

کد زیر رو تو یه فایل جدید بنویسیم:

```
-- print3.hs
module Print3 where

myGreeting :: String
myGreeting = "hello" ++ " world!"

hello :: String
hello = "hello"
```

[۱] Concatenation

```
world :: String
world = "world!"

main :: IO ()
main = do
  putStrLn myGreeting
  putStrLn secondGreeting
  where secondGreeting =
          concat [hello, " ", world]
```

با استفاده از :: تایپِ همه‌ی بیانیه‌های سطح بالا [۱]رو تعیین کردیم. البته کامپایلر خودش می‌تونه تایپ‌ها رو استنتاج کنه[۲] و نیازی به نوشتنِ تایپ سیگنچرِ بیانیه‌ها نیست، با این حال، عادت به نوشتن اونها برای برنامه‌های بزرگتر مفیده.

گفتیم String و [Char] مترادف تایپی‌اند. می‌تونین با تغییرِ تایپ سیگنچرها در مثالِ آخر، و اجرای برنامه‌تون، امتحان کنین ببینین چیزی تغییر می‌کنه یا نه.

---

[۱] Top-Level Expressions

[۲] م. به این قابلیتِ کامپایلر، type inference گفته میشه.

اگه برنامه رو اجرا کنیم:

```
Prelude> :l print3.hs
[1 of 1] Compling Print3
Ok, modules loaded: Print3.
*Print3> main
hello world!
hello world!
*Print3>
```

در این مثالِ ساده، چندتا چیز رو نشون دادیم:

۱. مقادیر رو در سطحِ بالای برنامه تعریف کردیم: `myGreeting`، `hello`، `world`، و `main`. منظور اینه که اونها در کلِ ماژول قابل دسترسی بودند.

۲. تایپِ تعاریفِ سطح بالا رو صراحتاً مشخص کردیم.

۳. نوشته‌ها رو با `(++)` و `concat` به هم الحاق یا concatenate کردیم.

۳ – ۴ تعاریف سطح بالا و محلی

منظور از تعاریف سطح بالا، این نیست که حتماً بالای فایل نوشته شده باشن. وقتی کامپایلر یه فایل رو می‌خونه، همه‌ی تعاریفِ سطح بالا (top-level declarations) رو با هم می‌بینه و ترتیب‌شون توی فایل اهمیتی نداره (البته بعضی محدودیتهایی وجود دارن که بعداً می‌بینیم). تعاریفِ سطح بالا در scope یا گستره‌ی کلِ ماژول هستن، و زیرِ هیچ بیانیه‌ی دیگه‌ای تودرتو یا nest نشدن.

متضادِ تعاریف سطح بالا، تعاریفِ محلی[۱] هستن. چیزی که محلی تعریف شده، یعنی زیر یه بیانیه‌ی دیگه تودرتو (nest) شده، و خارج از اون بیانیه قابل دسترسی نیست. در فصلِ قبلی، چند بار با `let` و `where` مقادیر محلی تعریف کردیم. با مثال زیر دوره می‌کنیم:

---

[۱] Local

```
module TopOrLocal where

topLevelFunction :: Integer -> Integer
topLevelFunction x =
  x + woot + topLevelValue
  where woot :: Integer
        woot = 10

topLevelValue :: Integer
topLevelValue = 5
```

اگه از یه ماژول دیگه، ماژول بالا (TopOrLocal) رو وارد یا import کنین، به topLevelFunction و topLevelValue دسترسی پیدا می‌کنین. ولی woot، خارج از topLevelFunction عملاً نامرئی‌ه. در هسکل، عبارات let و where برای انقیاد یا تعریف‌های محلی به کار میرن. منظور از انقیاد (bind) یا تعریف (declare) کردن، دادنِ اسم به یه بیانیه‌ست. میشه توابع رو بی‌نام تعریف کرد و هر دفعه تابع رو تماماً نوشت، اما با نامگذاری و استفاده از اسمش (بجای کلِ تابع) تکرار کمتر میشه.

نکته‌ی دیگه اینکه ما برای woot هم، تایپ‌ِش رو با گرامرِ :: زیر where صراحتاً نوشتیم. این کار اصلاً لازم نبود (*استنتاجِ تایپِ* هسکل خودش تشخیص میداد)، اما اینجا نوشتیم تا گرامرِ تایپ سیگنچرِ محلی رو هم دیده باشین. حتماً این کد رو تو REPL بارگذاری و اجرا کنین:

```
Prelude> :l TopOrLocal.hs
[1 of 1] Compling TopOrLocal
Ok, modules loaded: TopOrLocal.
*TopOrLocal> topLevelFunction 5
20
```

با آرگومان‌های دیگه هم امتحان کنین، خودتون هم توی ذهن (یا روی کاغذ) مراحل محاسبات رو انجام بدین تا از درک نتایج حاصل مطمئن بشین.

تمرین‌ها: گستره

۱. این کُدها در یک جلسه[۱] توی REPL نوشته شدن. آیا y در گستره‌ی z هست؟

---

[۱] Session

```
Prelude> let x = 5
Prelude> let y = 7
Prelude> let z = x * y
```

۲. این کُدها در یک جلسه توی REPL نوشته شدن. آیا h در گستره‌ی g هست؟

```
Prelude> let f = 3
Prelude> let g = 6 * f + h
```

۳. این نمونه کد از یه فایلِ منبعه. آیا برای اجرای area، هر چیزی که لازم داریم در گستره هست؟

```
area d = pi * (r * r)
r = d / 2
```

۴. این کد هم از یه فایل منبعه. آیا r و d در گستره‌ی area هستن؟

```
area d = pi * (r * r)
   where r = d / 2
```

۳ – ۵ تایپ‌های توابعِ الحاق

یه نگاه به تایپِ توابعِ (++) و concat بندازیم. تابع ++ یک عملگرِ میانونده. هر وقت لازم بشه یه عملگر میانوندی رو پیشوندی استفاده کنیم (مثل زمانی که بخوایم پشت آرگومان‌ها باشن، یا بخوایم با دستورِ :t تایپِ‌شون رو پیدا کنیم)، باید دو طرفش پرانتز بذاریم. در مقابل، concat یه تابعِ معمولیه (میانوندی یا infix نیست)، پس پرانتز لازم نداره:

```
Prelude> :t (++)
(++) :: [a] -> [a] -> [a]
Prelude> :t concat
concat :: [[a]] -> [a]
```

تایپِ تابع concat میگه که یه لیستی از لیست‌ها رو به عنوان ورودی می‌گیره، و یه لیست برمی‌گردونه. و محتوای این لیستِ خروجی، همون تایپی رو دارند که لیستِ لیست در ورودی داره. این تابع، به کلامی، ورودیش رو از دو ساختار (لیست) به یک ساختار "لِه[۱]" می‌کنه.

---

[۱] Flatten

همونطور که گفتیم، String یه لیسته؛ لیستی از Char. ورودیِ تابع concat هم می‌تونه یه لیستی از string باشه، یا یه لیستی از لیستِ تایپ‌های دیگه:

```
Prelude> concat [[1, 2], [3, 4, 5], [6, 7]]
[1,2,3,4,5,6,7]
Prelude> concat ["Iowa", "Melman", "Django"]
"IowaMelmanDjango"
```

(نکته: اگه از GHC ۷٫۱۰ یا جدیدتر استفاده می‌کنین، تایپ سیگنچرِ متفاوتی برای concat می‌بینید. بعداً به تفصیل توضیح میدیم، ولی فعلاً تایپِ Foldable t => t [a] رو بخونین [[a]]. الان کفایت می‌کنه که Foldable t رو یه جور لیستِ دیگه بدونین. در حقیقت، لیست فقط یکی از تایپ‌های ممکن – تایپ‌هایی که نمونه‌ای از تایپکلاسِ Foldable دارند – برای این تابعه، ولی در حال حاضر فقط لیست برای ما مهمه.)

اما معنیِ این تایپ‌ها چیه؟ کمی دقیق‌تر بررسی می‌کنیم:

```
(++) :: [a] -> [a] -> [a]
--      [1]    [2]    [3]
```

هر چیزی بعد از :: مرتبط با تایپ‌هاست، و نه مقادیر. حرفِ a داخلِ نوع‌سازِ[۱] لیست، []، یک متغیرِ تایپی[۲] ِه.

۱. یک آرگومان با تایپِ [a] بگیر. این تایپ، یه لیستی از اِلمان‌ها با تایپِ a ِه که تابع نمی‌دونه اون a چیه. تایپِ a بالاخره در جایی از برنامه شناخته و معین میشه.

۲. یه آرگومانِ دیگه با تایپِ [a] (یه لیستی از اِلمان‌ها که تایپ‌شون رو نمی‌دونیم) بگیر. چون متغیرهای تایپی یکسان‌اند، پس تایپ‌شون هم باید در کلِ تابع یکسان باشه ($a == a$).

۳. یک جواب از تایپ [a] برگردون.

از اونجا که String یه نوع لیسته، عملگرهایی که برای String استفاده می‌کنیم رو میشه به لیستِ بقیه‌ی تایپ‌ها هم اعمال کرد، مثل لیستِ اعداد. تایپِ [a] یعنی یه لیستی از اِلمان‌ها با تایپِ a داریم، که

---

[۱] Type Constructor

[۲] Type Variable

هنوز چیزی از اون تایپ نمی‌دونیم. اگه از عملگرِ الحاقِ لیست‌ها برای لیست اعداد استفاده کنیم، اون موقع، a که در [a] هست، یکی از تایپ‌های اعداد، مثلاً اعدادِ صحیح، میشه. اگه لیستِ حروف رو الحاق کنیم، a تایپِ Char رو نشون میده (و String هم در واقع [Char] ه). متغیرِ تایپیِ a در [a] یک چندریخت[1] ه. *چندریختی* یا *پلی‌مورفیسم* (*Polymorphism*) از امکاناتِ مهمِ هسکل‌ه. برای الحاقِ چند لیست، همه‌ی اونها باید از یک تایپ باشن؛ برای مثال، الحاق یه لیست از اعداد با یه لیست از حروف ممکن نیست. البته متغیرِ a در سطح تایپ‌هاست، و اصلاً لازم نیست که خودِ مقادیرِ داخل هر کدوم از لیست‌ها یکسان باشن، فقط باید از یک *تایپ* باشن. به کلامی دیگه، a باید با a برابر باشه ($a == a$).

```
Prelude> "hello" ++ " Chris"
"hello Chris"
```

ولی:

[1] Polymorphic

```
Prelude> "hello" ++ [1, 2, 3]

<interactive>:14:13:
    No instance for (Num Char) arising
      from the literal ‘1’
    In the expression: 1
    In the second argument of ‘(++)’,
      namely ‘[1, 2, 3]’
    In the expression: "hello" ++ [1, 2, 3]
```

در مثال اول دو تا string داریم، پس تایپِ a ها یکسان‌اند – هردوشون Char اند (از [Char])، و اهمیتی نداره که مقادیرِشون یکسان نیست. همین که تایپ‌ها با هم جور هستن، هیچ خطای تایپی اتفاق نمی‌افته و نتیجه‌ی الحاق شده رو می‌بینیم.

در مثال دوم، دو تا لیست داریم (یکی String و اون یکی لیستی از اعداد) که تایپ‌شون با هم جور نیست، پس با خطا روبرو شدیم. GHCi دنبالِ یک نمونه[۱] از تایپکلاسِ اعداد به اسم Num برای Char میگرده. *تایپکلاس*‌ها تعاریفی از عملگرها، یا توابع هستند، که امکانِ به اشتراک

[۱] Instance

گذاشته شدنِشون بین یک دسته از تایپ‌ها وجود داره. اما فعلاً کفایت می‌کنه که این پیغامِ خطا رو به معنیِ جور نبودنِ تایپ‌ها برای الحاقِ دو لیست بدونین.

تمرین‌ها: خطاهای گرامری

گرامرِ توابع زیر رو بررسی کنید و بگین آیا کامپایل میشن یا نه. با REPL تست کنید، و هر کجا مشکل گرامری بود، سعی کنید اصلاح شده‌اش رو بنویسین.

۱. `++ [1, 2, 3] [4, 5, 6]`

۲. `'<3' ++ ' Haskell'`

۳. `concat ["<3", " Haskell"]`

۳ – ۶ الحاق و گستره

از پرانتز استفاده می‌کنیم تا تابعِ ++ رو به صورت پیشوندی (و نه میانوندی) استفاده کنیم:

```
-- print3flipped.hs
module Print3Flipped where
```

```haskell
myGreeting :: String
myGreeting = (++) "hello" " world!"

hello :: String
hello = "hello"

world :: String
world = "world!"

main :: IO ()
main = do
  putStrLn myGreeting
  putStrLn secondGreeting
  where secondGreeting =
          (++) hello ((++) " " world)
  -- میانوندی اینطور میشد:
  --    secondGreeting =
  --      hello ++ " " ++ world
```

با استفاده از ++ به عنوانِ تابع پیشوندی برای secondGreeting، مجبور شدیم چندتا چیز رو جابجا کنیم. ما با این پرانتزگذاری،

شرکت‌پذیری از راستِ تابع ++ رو تأکید کردیم. به خاطر میانوند بودنِ این تابع، خودتون هم می‌تونین جهت شرکت‌پذیری‌ش رو چک کنید:

```
Prelude> :i (++)
(++) :: [a] -> [a] -> [a]   -- از GHC.Base
infixr 5 ++
```

عبارتِ where، به بیانیه‌ها یه تعریفِ محلی میده که در سطح بالا قابل دسترسی نیستن. به عبارت دیگه، where داخل main، یک تعریفی میده که فقط در محدوده‌ی تابع یا بیانیه‌ی بالا سَرِش در دسترس‌ه تا اینکه در کل ماژول مرئی باشه. چیزی که در سطحِ بالا مرئی باشه، در گستره‌ی همه‌ی بخش‌های یه ماژول قرار داره، و امکانِ صادر شدن‌ش توسطِ ماژول، یا وارد شدن‌ش به ماژول‌های دیگه وجود داره. در مقابل، تعاریف محلی فقط برای تابعِ خودشون در دسترس‌اند، نمیشه secondGreeting رو به یه ماژول دیگه وارد و ازش دوباره استفاده کرد.

برای اینکه بهتر نشون بدیم:

```haskell
-- print3broken.hs
module Print3Broken where

printSecond :: IO ()
printSecond = do
  putStrLn greeting

main :: IO ()
main = do
  putStrLn greeting
  printSecond
  where greeting = "Yarrrrr"
```

یه پیغام خطا مشابه این می‌گیرین:

```
Prelude> :l print3broken.hs
[1 of 1] Compiling Print3Broken
  ( print3broken.hs, interpreted )

print3broken.hs:6:12: Not in scope: ‘greeting’
Failed, modules loaded: none.
```

اگه نگاه دقیق‌تری به این خطا بندازیم:

```
print3broken.hs:6:12: Not in scope: ‘greeting’
--              [1][2]     [3]          [4]
```

۱. شماره‌ی سطری که ایراد داشت و باعثِ error شد (در این مورد، ۶).

۲. ستونی که خطا داشت: ستونِ ۱۲. متونِ کامپیوتر عموماً با سطر و ستون توصیف میشن. شماره‌ی این سطرها و ستون‌ها به سطرها و ستون‌های فایلِ متنی که کُدتون رو توش نوشتین اشاره دارن.

۳. خودِ مشکل: یه چیزی در گستره نیست، یعنی اون چیز از دیدِ تابعِ `printSecond` پنهان‌ه.

۴. همون چیزی که در گستره نیست.

کاری کنین که `Print3Broken` کامپایل شه. باید در دو خط، دو بار نوشته‌ی "Yarrrrr" رو چاپ کنه.

۳ – ۷ چندتا تابعِ دیگه برای لیست‌ها

از اونجا که `String` یه حالت خاص از لیست‌هاست، امکانِ اعمالِ بقیه‌ی عملگرهای لیست هم روی اون وجود داره.

در زیر چندتا از این توابع رو مثال زدیم:

```
Prelude> :t 'c'
'c' :: Char
Prelude> :t "c"
"c" :: [Char]
-- .ه String همون [Char]
```

اوپراتورِ (:)، که *cons* خونده میشه، برای ساختنِ لیست به کار میره:

```
Prelude> 'c' : "hris"
"chris"
Prelude> 'P' : ""
"P"
```

تابعِ بعدی، head، سَر یا اولین اِلمان یه لیست رو برمی‌گردونه:

```
Prelude> head "Papuchon"
'P'
```

مکملِ اون تابع، تابعِ tail، لیست رو بدونِ اولین المان پس میده (سر بریده!):

```
Prelude> tail "Papuchon"
"apuchon"
```

take برای گرفتنِ تعداد مشخصی از المان‌های لیست به کار میره (از سمت چپ):

```
Prelude> take 1 "Papuchon"
"P"
Prelude> take 0 "Papuchon"
""
Prelude> take 6 "Papuchon"
"Papuch"
```

تابعِ drop هم بعد از حذفِ تعداد المان‌های تعیین شده، باقیمانده‌ی لیست رو برمی‌گردونه:

```
Prelude> drop 4 "Papuchon"
"chon"
Prelude> drop 9001 "Papuchon"
""
Prelude> drop 1 "Papuchon"
"apuchon"
```

عملگرِ (++) رو هم قبلاً دیدیم:

```
Prelude> "Papu" ++ "chon"
"Papuchon"
Prelude> "Papu" ++ ""
"Papu"
```

عملگر میانوندِ (!!) برای گرفتنِ n اُمین المانِ لیسته، که n اندیسِ[۱] المان، و از صفر شروع میشه. یعنی برای این تابع، اولین المان اندیسِ صفر داره، و نه یک.

```
Prelude> "Papuchon" !! 0
'P'
Prelude> "Papuchon" !! 4
'c'
```

همه‌ی این توابعِ استاندارد، جزئی از Prelude هستن، با این حال خیلی‌هاشون *نا اَمن* تلقی میشن. دلیلش هم پشتیبانی نکردن یکی از حالت‌های ورودی، یعنی لیستِ خالیه. اگه آرگومانِ اکثر این توابع یه لیست خالی باشه، پیغام خطا، یا *استثنا*[۲] میدن:

---

[۱] Index

[۲] Exception

```
Prelude> head ""
*** Exception: Prelude.head: empty list
Prelude> "" !! 4
*** Exception: Prelude.!!: index too large
```

چنین رفتاری از یه تابع اصلاً ایده‌آل نیست، و به همین دلیل استفاده از این توابع در برنامه‌های اساسی خیلی عاقلانه به حساب نمیاد، ولی ما ازشون تو این چندتا فصلِ اول استفاده می‌کنیم. در یکی از فصل‌های آینده، توضیح میدیم که چطور همه‌ی حالت‌های ورودی برای توابع رو از پیش در نظر بگیریم، و نسخه‌ی امنِ این توابع رو بنویسیم.

۳ – ۸ تمرین‌های فصل

خوندنِ گرامر

۱. سلامت گرامریِ هر کدوم از کُدهای زیر رو تعیین کنین. بعد از اینکه به نتیجه رسیدین، کدها رو توی REPL تست کنین. هر چندتا رو که تونستین تصحیح کنین.

a) `concat [[1, 2, 3], [4, 5, 6]]`

b) `++ [1, 2, 3] [4, 5, 6]`

c) `(++) "hello" " world"`

d) `["hello" ++ "world]`

e) `4 !! "hello"`

f) `(!!) "hello" 4`

g) `take "4 lovely"`

h) `take 3 "awesome"`

۲. اینجا دو دسته داریم. دسته‌ی اول چند خط کد اند، و دسته‌ی دوم جواب اونهاست. کد رو بخونین و جوابش رو از دسته‌ی دوم پیدا کنین. حتماً در REPL هم امتحان کنین.

a) `concat [[1 * 6], [2 * 6], [3 * 6]]`

b) `"rain" ++ drop 2 "elbow"`

c) `10 * head [1, 2, 3]`

d) `(take 3 "Julie") ++ (tail "yes")`

e)
```
concat [tail [1, 2, 3],
        tail [4, 5, 6],
        tail [7, 8, 9]]
```

جوابِ کدهای بالا رو این زیر نوشتیم، ولی ترتیب‌شون درست نیست.

a) `"Jules"`

b) `[2,3,5,6,8,9]`

c) `"rainbow"`

d) `[6,12,18]`

e) `10`

ساخت توابع

۱. بر مبنای توصیفی که از توابع داده شده، به همراه ورودی و خروجی موردِ نظرشون، تعریف تابع رو بنویسید. اینکار رو مستقیماً در REPL انجام بدین.

به عنوان مثال:

```
-- اگه تابع‌تون رو به این مقدار
-- اعمال کنید:
"Hello World"
-- باید چنین جوابی برگردونه:
"ello World"
```

این جوابِ مناسبیه:

```
Prelude> drop 1 "Hello World"
"ello World"
```

حالا بیانیه‌هایی بنویسین که تبدیلاتِ زیر رو انجام بدن. توابعی که تو این فصل دیدین کفایت می‌کنن و هیچ کدوم نکته‌ی گمراه‌کننده‌ای ندارن.

a)
```
-- داریم
"Curry is awesome"
-- جواب بده
"Curry is awesome!"
```

b)
```
-- داریم
"Curry is awesome!"
```

```
-- جواب بده
"y"
```

c)
```
-- داریم
"Curry is awesome!"
-- جواب بده
"awesome!"
```

۲. حالا جواب‌هاتون رو به صورت توابعِ جامع که آرگومان‌های نوشتاری می‌گیرن، تو یه فایل بنویسین. از یه متغیر به عنوان آرگومان تابعِ اسم‌دارِتون استفاده کنین. اگه مطمئن نیستین چطور این توابع رو بنویسین، می‌تونین تمرینِ `waxOff` از فصلِ قبل و یا ماژولِ `TopOrLocal` از این فصل رو دوره کنین.

۳. یه تابع با تایپِ `String -> Char` بنویسین که حرف سومِ ورودیش رو برمی‌گردونه. یادتون باشه که یه اسم به تابع بدین، و به یه متغیر هم اعمالِش کنین (نه به یه `String`ِ معین) تا بشه به `String` های مختلف اِعمالِش کرد. مقداری از کد رو اینجا نوشتیم (اگه دوست داشتین اسم تابع رو تغییر بدین.

حتماً تایپ سیگنچرِ تابع رو بنویسین؛ تعریف تابع رو هم بعد از علامت مساوی بیارین):

```
thirdLetter ::
thirdLetter x =

-- اگه اون تابع رو به این مقدار
-- اعمال کنید:
"Curry is awesome"
-- باید این جواب رو برگردونه:
'r'
```

فقط دقت کنین که مثلِ اکثر زبان‌های برنامه‌نویسی، اندیس‌ها در هسکل از صفر شروع میشن، پس صفرُمین اندیسِ یه string، حرف اولش میشه. و متعاقباً، سومین اندیس میشه حرف چهارم.

۴. حالا تابع‌تون رو طوری تغییر بدین که همیشه با یه نوشته کار کنه، و ورودیش اندیسِ حرفی که برمی‌گردونه باشه (می‌تونین از "Curry is awesome!" به عنوان نوشته‌ی تابع استفاده کنین، یا اگه دوست داشتین یه string ِدیگه بنویسین).

```
letterIndex  :: Int -> Char
letterIndex x =
```

۵. با استفاده از توابع take و drop که تو این فصل دیدیم، سعی کنین یه تابع به اسم rvrs بنویسین (خلاصه‌ای از لغت reverse به معنای معکوس. Prelude خودِش یه تابع به اسم reverse داره و اگه شما هم همون اسم رو به تابع‌تون بدین، پیغام خطا می‌گیرین) که نوشته‌ی “Curry is awesome” رو به نوشته‌ی “awesome is Curry” تبدیل می‌کنه. کُد خیلی جالبی نیست، اما چیزهایی که تا الان یاد گرفتیم برای نوشتنِ این تابع کاملاً کفایت می‌کنن. اول تابع رو تو یه فایل تعریف کنین. قرار نیست که این تابع لغات همه‌ی جملات رو معکوس کنه، فقط با drop و take لغاتِ همون یک جمله رو جابجا کنین.

۶. حالا سعی کنیم فایل‌مون رو به یه ماژول گسترش بدیم. این کار اجازه میده بعداً توابع بیشتری اضافه کنیم، همچنین می‌تونیم توابعِ این ماژول رو به ماژول‌های دیگه صادر کنیم تا اونها هم به توابعِ صادراتیِ این ماژول دسترسی پیدا کنن.

راههای متفاوتی برای این کار هست، ولی برای راحتی؛ این الگوی ساده رو در زیر آوردیم:

```
module Reverse where

rvrs :: String -> String
rvrs x =

main :: IO ()
main = print ()
```

داخل پرانتز جلوی print، تابع rvrs رو با آرگومانی که میخواین بهش اعمال شه (در این مورد “Curry is awesome”) بنویسین. حالا اون تابع rvrs به همراه آرگومانِش، مجموعاً آرگومان تابع print شدن. دقت کنید که حتماً داخل پرانتز بنویسین، تا نتیجه‌ی rvrs چاپ بشه.

البته همونطور که گفتیم، میشه از $ هم برای کاهش پرانتزها استفاده کرد. این روش هم امتحان کنین.

۳ – ۹ تعاریف

۱. یک *نوشته* یا *string* تسلسلی از حروفه. در هسکل، `String` با یه لیستی از مقادیر با تایپِ `Char`، یا به کلامی دیگه `[Char]`، ارائه میشه.

۲. *نوع‌داده*، *تایپ* یا *datatype* یک طبقه‌بندی از مقادیر یا داده‌هاست. تایپ‌ها در هسکل تعیین‌کننده‌ی مقادیری هستن که اعضای اون تایپ هستن (یا توی اون تایپ *زندگی می‌کنن*[۱]). برخلافِ بقیه‌ی زبان‌ها، نوع‌داده‌ها در هسکل، عملیات‌هایی که میشه روی مقادیرِشون پیاده کرد رو بطور پیش‌فرض محدود نمی‌کنند.

۳. به اتصالِ چندتا سکانسِ مقادیر، *الحاق* یا *concatenation* گفته میشه. در هسکل، منظور از الحاق اکثراً در رابطه با تایپِ `[]` یا لیسته، که `String` هم شامل میشه (`String` یه تایپِ مستعار

---

[۱] Inhabit

یا type alias برای [Char] ِه). تابعِ (++) در هسکل، تابع الحاقه که تایپِ [a] -> [a] -> [a] رو داره. برای مثال:

```
Prelude> "tacos" ++ " " ++ "rock"
"tacos rock"
```

۴. *گستره* یا *scope* جاییه که اسمِ یه متغیر اعتبار داره. یه لغتِ دیگه به همین معنی، *پدیداری* یا *visibility* ِه. چون اگه یه متغیر پیدا (visible) باشه، در گستره هم هست.

۵. *انقیادهای محلی* یا *local bindings* تعاریفی هستن که معمولاً داخل یک بیانیه نوشته میشن. تفاوتِ اصلی اونها با انقیادهای *سطح بالا* یا *top level* اینه که برنامه‌ها یا ماژول‌های دیگه به اونها دسترسی ندارن (نمی‌تونن اونها رو وارد کنن).

۶. *انقیادهای سطح بالا* یا *top level bindings* تعاریفی‌اند که بیرون از همه‌ی بیانیه‌های دیگه نوشته میشن. یک ماژول می‌تونه انقیادهای سطح بالای داخلِش رو در اختیار بقیه‌ی ماژول‌های برنامه‌تون یا برنامه‌های دیگران قرار بده.

۷. *ساختارِ داده* یا *data structure* راهی برای سازماندهیِ داده‌ها برای دسترسیِ آسون و یا به‌صرفه به اونهاست (م. برای مثال، لیست یک ساختارِ داده‌ست).

فصل ۴

تایپ‌های پایه

راههای زیادی برای درک برنامه‌ها وجود داره. راهی که مردم خیلی بهش اتکا می‌کنن، اشکال‌زداییه، یعنی یه برنامه که نیمه‌کاره درک کردی رو اجرا کنی و ببینی که آیا جواب مطلوب رو میده یا نه. راه دیگه که زبان ML هم ازش حمایت می‌کنه، نصب روشهایی برای درک برنامه در خودِ برنامه‌ست.
– رابین میلنر

There are many ways of trying to understand programs. People often rely too much on one way, which is called “debugging” and consists of running a partly-understood program to see if it does what you expected. Another way, which ML advocates, is to install some means of understanding in the very programs themselves.
– Robin Milner

۴ – ۱ تایپ‌های پایه

تایپ سیستمِ[۱] هسکل خیلی گویا، قابل اعتماد و مستحکم طراحی شده. تایپ‌ها نقشِ خیلی مهمی در خوانایی، امنیت، و نگهداری‌پذیریِ[۲] کُدهای هسکل دارن، چراکه به ما امکانِ طبقه‌بندیِ داده‌ها و اعمال محدودیتهایی روی اونها رو میدن، و متعاقباً برنامه‌هامون هم حالتهای محدودتری از داده‌ها رو پردازش می‌کنن. تایپ‌ها، که به اونها *datatype* هم میگیم، راههایی رو ارائه میدن که ساختِ سریع برنامه‌ها و نگهداریِ خیلی آسون‌ترِ اونها رو ممکن می‌کنه. همینطور که با هسکل آشناتر میشیم، یاد می‌گیریم چطور از خاصیتهای تایپ‌ها بهتر استفاده کنیم، و کارهای مشابه رو با کدِ کمتر انجام بدیم.

تا اینجا بیانیه‌های ما اکثراً شاملِ اعداد، حروف، و لیستی از حروف (یا همون string) بودن. اینها چندتا از تایپ‌های اساسی‌اند که از کتابخونه یا libraryِ استاندارد وارد شدن؛ بسیار کاربردی‌اند و شامل

---

[۱] Type System

[۲] Maintainability

خیلی از تایپ‌ها میشن، با این حال همه‌ی تایپ‌های داده محدود به اینها نیستن. در این فصل:

- تایپ‌هایی که فصل قبل دیدیم رو دوره می‌کنیم؛
- تایپ‌ها، نوع‌سازها[۱]، و داده‌سازها[۲] رو بیشتر می‌شناسیم؛
- با تایپ سیگنچرها بیشتر آشنا میشیم و خیلی کم از تایپکلاس‌ها صحبت می‌کنیم.

۴ – ۲ تایپ چیه؟

هر وقت یه بیانیه محاسبه میشه، به یه مقدار ساده میشه؛ هر مقداری هم یک نوع یا تایپ داره. با تایپ، مقادیری که یه چیزِ مشترکی بین‌شون هست رو با هم دسته‌بندی می‌کنیم. بعضی وقتها این چیزِ مشترک، کلی و abstract ِه؛ گاهی اوقات هم یه مدلِ خاص از یه مفهوم[۳] یا دامنه‌ی بخصوص‌ه. اگه تو کلاسهای ریاضی با مجموعه‌ها آشنا شدین، می‌تونین

---

[۱] Type Constructors

[۲] Data Constructors

[۳] Concept

هر تایپ رو یه مجموعه فرض کنین. این نگرش کمک می‌کنه ماهیتِ تایپ‌ها و اینکه چطور کار می‌کنن رو با زبان ریاضی[۱] درک کنین.

۴ – ۳ آناتومی تعریف داده

تعریفِ داده[۲] راهی برای تعریفِ تایپ‌هاست.

نوع‌ساز یا *type constructor*، اسمِ تایپه و با حروفِ بزرگ شروع میشه. وقتی تایپ سیگنچرها رو می‌خونین یا می‌نویسین (در سطحِ نوعیِ[۳] کد هستین)، از اسمِ تایپ‌ها، یا نوع‌سازها استفاده می‌کنین.

داده‌سازها یا *data constructors* مقادیری‌اند که داخلِ یک تایپ تعریف میشن. همون مقادیری‌اند که در سطح جمله‌ایِ[۴] کد ظاهر

---

[۱] نظریه‌ی مجموعه‌ها یا Set Theory، شاخه‌ای از ریاضی‌ه، که به مطالعه‌ی مجموعه‌های ریاضیاتی از اشیا می‌پردازه. این نظریه، پیشینه‌ای برای نظریه‌ی نوع‌ها یا Type Theory ه، که از نظریه‌ی نوع‌ها برای طراحی زبان‌هایی مثل هسکل، به وفور استفاده شده. عملگرهای منطقی، مثل فصل ("یا") و عطف ("و")، که برای مجموعه‌ها استفاده میشن، در "تایپ سیستمِ" هسکل هم معادل دارن.

[۲] Data Declaration

[۳] Type Level

[۴] Term Level

میشن، و نه سطح نوعی. منظور از *سطح جمله‌ای*، مقادیری‌اند که در خودِ کد ظاهر میشن، و یا کد به اونها حساب میشه.

برای آشنایی، و اینکه چطور datatype یا نوع‌داده‌ها درست میشن، با یه تایپِ پایه مثل `Bool` شروع می‌کنیم. تایپِ `Bool`، که به نامِ ریاضیدانِ بزرگ، جورج بول[۱] و سیستمِ منطق او نامگذاری شده، مقادیر واقعی[۲] رو تعریف می‌کنه. از اونجا که فقط دو مقدارِ واقعی وجود دارن، `Bool` هم فقط دو داده‌ساز داره:

```
-- تعریف Bool
data Bool = False | True
--    [1]    [2] [3] [4]
```

۱. نوع‌ساز برای تایپِ `Bool`. اسمِ تایپ که در تایپ سیگنچرها هم دیده میشه.

۲. داده‌ساز برای مقدار `False`.

---

[۱] George Boole

[۲] Truth Values

۳. خط عمودی | که تعیین کننده‌ی *نوعِ جمع*[۱] یا فصل منطقی[۲] ("یا") ِه.

۴. داده‌ساز برای مقدار `True`.

به کلِ اون عبارت میگیم تعریفِ داده. data declaration یا تعریف داده‌ها می‌تونن شکل‌های دیگه هم داشته باشن – بعضی تایپ‌ها عطفِ منطقی[۳] ("و") دارن (برخلافِ این مثال که فصلِ منطقی‌ه)، بعضی نوع‌سازها و داده‌سازها هم آرگومان دارن. با وجودِ این تفاوت‌ها، همه‌شون نقاطِ مشترکی هم دارن: یکی کلیدواژه‌ی `data` در اولِ تعریفه که بعد از اون، نوع‌ساز (یا همون اسمِ تایپ) نوشته میشه، علامت مساوی که نشون میده این یه تعریفه، و آخر هم داده‌سازها (یا اسم مقادیری که در سطحِ جمله‌ای کد قرار می‌گیرن) که بعد از مساوی نوشته میشن.

---

[۱] Sum Type

[۲] Logical Disjunction

[۳] Logical Conjunction

شما می‌تونین تعریف یک نوع‌داده‌ی از پیش تعریف شده رو با دستور `:info` در GHCi ببینین:

```
Prelude> :info Bool
data Bool = False | True
```

حالا ببینیم بخشهای مختلفِ یه نوع‌داده در کجاهای کد دیده میشن. اگه تایپِ تابعِ `not` رو استعلام یا query کنیم، می‌بینیم که یه مقدارِ `Bool` می‌گیره و یه مقدار `Bool`ِ دیگه برمی‌گردونه. پس تایپ سیگنچر، به نوع‌ساز یا اسمِ تایپ اشاره می‌کنه:

```
Prelude> :t not
not :: Bool -> Bool
```

ولی موقعِ استفاده از این تابع، از داده‌سازها (یا خودِ مقادیر) استفاده می‌کنیم:

```
Prelude> not True
False
```

بیانیه‌مون هم به یه داده‌ساز یا مقدارِ دیگه ساده میشه — که اینجا تنها داده‌سازِ دیگه‌ی همون تایپ رو نتیجه میده.

تمرین‌ها: نوسان خُلقی

با نوع‌داده‌ی زیر، سؤال‌های زیر رو جواب بدین:

```
data Mood = Blah | Woot deriving Show
```

هنوز اون تیکه‌ی `deriving Show` رو توضیح ندادیم. فعلاً فقط در همین حد توضیح میدیم که مشتق کردنِ[۱] `Show` باعث میشه مقادیرِ تایپ‌هایی که خودتون تعریف می‌کنین قابلِ چاپ بشن. به تایپکلاس‌ها که برسیم، بیشتر توضیح میدیم.

۱. نوع‌ساز، یا اسم این تایپ چیه؟

۲. اگه تابعی ورودیش یه مقدارِ `Mood` باشه، از چه مقادیری میشه استفاده کرد؟

۳. میخوایم یه تابعِ `changeMood` بنویسیم که خُلقِ کریس رو فوراً عوض کنه[۲]. باید کاری شبیهِ تابع `not` انجام بده: با یک مقدارِ

---

[۱] Deriving

[۲] م. کریس (Chris) اسمِ یکی از نویسنده‌های این کتابه. اسم انتخابی تابع هم به معنی "تغییرِ خلق"ه.

ورودی، تنها ورودی دیگه‌ی همون تایپ رو برگردونه. ما این تایپ سیگنچر رو براش نوشتیم:

```
changeMood :: Mood -> Woot
```

مشکلش چیه؟

۴. حالا خودِ تابع رو می‌نویسیم. با یه خُلقِ ورودی، خلق دیگه رو برمی‌گردونه. هر اشکالی هست برطرف و تابع رو تکمیل کنین:

```
changeMood Mood = Woot
changeMood    _ = Blah
```

اسمِ کاری که اینجا کردیم، *تطبیقِ الگو*[۱] ه. میشه در تعریفِ یه تابع، اون رو با داده‌سازها، یا مقادیرِ مختلف تطبیق داد، و رفتارِ اون رو بر اساس مقداری که باهاش منطبق میشه توصیف کرد. خطِ تیره‌ای که در خط دوم نوشتیم، نشونه‌ی "بقیه‌ی حالتها"ست. بنابراین، در خطِ اول، تعریف کردیم که به ازای یه ورودیِ مشخص، تابع باید چه کاری انجام بده. و در خط دوم،

---

[۱] Pattern Matching

رفتار تابع به ازای همه‌ی ورودی‌های ممکنِ دیگه رو توصیف کردیم. تو این مثال که تایپ ورودی فقط دو مقدار داره، استفاده از خط فاصله لازم نیست، ولی برای توابعِ پیچیده‌تر، ضروری میشه.

۵. همه‌ی چیزهایی که بالاتر نوشتین رو تو یه فایل هم بنویسین – نوع‌داده (به همراهِ اون `deriving Show`)، تایپ سیگنچرِ اصلاح شده، و تابع تکمیل شده. فایل رو در GHCi بارگذاری کنین تا از درست بودنِ همه چیز مطمئن بشین.

۴ – ۴ تایپ‌های عددی

اعداد رو کمی در فصل‌های گذشته دیدیم، حالا اینجا با جزئیات بیشتری بررسی شون می‌کنیم. مهمه که بدونیم هسکل فقط از یک تایپ برای اعداد استفاده نمی‌کنه. تایپِ عددهایی که بیشتر باهاشون سر و کار داریم، اینها هستن:

اعدادِ integral اعداد تام یا کامل‌اند، چه مثبت و چه منفی.

۱. تایپ `Int`: این تایپ، عدد صحیح[۱] با دقتِ ثابت[۲]ه. منظور از دقت ثابت اینه که محدوده دارن، با یه مقدار ماکسیمم و یه مقدار مینیمم؛ و نمی‌تونن هر چقدر که می‌خوایم بزرگ یا کوچک باشن – کمی جلوتر بیشتر توضیح میدیم.

۲. تایپ `Integer`: این تایپ هم برای اعداد صحیحه، ولی اعدادش محدوده‌ای ندارن (هر چقدر که بخوایم بزرگ یا کوچک میشن).

کسری این اعداد صحیح نیستن. مقادیر `Fractional` شامل این چهار تایپ‌اند:

۱. تایپ `Float`: این تایپ برای اعداد اعشاریِ[۳] تک-دقتی[۴] به کار میره. اعدادِ ممیز-ثابت[۵]، تعداد ثابتی از ارقام برای قبل و بعد از

---

[۱] Integer

[۲] Fixed-Precision

[۳] اعداد با ممیز شناور، یا floating point numbers.

[۴] Single-Precision

[۵] Fixed-Point

ممیز دارن. در مقابل، ممیز شناور می‌تونه تعداد بیت‌هایی که برای اعداد قبل و بعد از ممیز استفاده میشن رو جابجا کنه. البته این انعطاف‌پذیری منجر به نقضِ بعضی فرضیه‌های مشترک برای اعداد میشه، و بهتره با نهایتِ دقت از اونها استفاده بشه. به طور کلی، در کاربردهای تجاری اصلاً نباید از اعداد با نقطه‌ی شناور استفاده کرد.

۲. تایپ `Double`: تایپی برای اعداد اعشاریِ دو-دقتی[۱]. مشابه تایپ `Float`، با تفاوت اینکه دو برابر بیت (bit) برای توصیف اعداد دارن.

۳. تایپ `Rational`: نوعی عدد کسری که نسبتِ دو عدد صحیح رو نشون میده. مقدار `5 / 7 :: Rational` حامِل دو مقدار `Integer`ِه: یکی صورت ۵، و اون یکی مخرج ۷. `Rational` دقت نامحدود داره، ولی به اندازه‌ی `Scientific` مقرون به صرفه نیست.

---

[۱] Double-Precision

۴. تایپ `Scientific`: نوعی از اعداد که فضای مناسبی رو اشغال می‌کنن، و دقت تقریباً نامحدودی دارند. اعداد `Scientific` با نماد علمی[۱] نشون داده میشن، و متشکل از یک ضریب از تایپ `Integer`، و یک توان از تایپ `Int` هستن. به خاطر محدودیت `Int`، در حقیقت اعداد `Scientific` هم محدودن، ولی خیلی کم پیش میاد به اون حد برسیم. تایپ `Scientific` در یه کتابخونه[۲] تعریف شده، و میشه با دستور `cabal install` یا `stack install` نصب‌ش کرد.

همه‌ی این datatype ها یک نمونه (instance) از تایپکلاسی به اسمِ `Num` دارن. ما در فصل‌های بعدی تایپکلاس‌ها رو توضیح میدیم، ولی تو این بخش `Num` رو در اطلاعات تایپ‌ها می‌بینین.

با تایپکلاس‌ها میشه قابلیت‌هایی رو به تایپ‌ها اضافه کرد که برای همه‌ی تایپ‌های دارای یه نمونه (instance) از اون تایپکلاس، قابل

---

[۱] Scientific Notation

[۲] آدرس صفحه‌ی Hackage:
`https://hackage.haskell.org/package/scientific`

استفاده باشن. Num تایپکلاسی‌ه که اکثر تایپ‌های عددی یک نمونه ازَش دارند، دلیل‌ش وجودِ توابع استانداردی‌ه که برای همه‌ی تایپ‌های عددی قابلِ استفاده‌اند. تایپکلاس Num چیزیه که عملگرهای استاندارد مثل (+)، (-)، و (*) و چندتای دیگه رو در اختیار اعداد قرار میده. میشه این توابع رو به هر تایپی که یه نمونه از تایپکلاس Num داره اعمال کنیم. یک نمونه یا instance، چگونگیِ عملکردِ توابع برای یک تایپ خاص رو تعریف می‌کنه. به زودی تایپکلاس‌ها رو با جزئیات خیلی بیشتر توضیح میدیم.

اعداد صحیح

همونطور که بالاتر گفتیم، اعدادِ integral، دو تایپِ اصلی دارن: Int و Integer.

اعداد integral، اعداد کامل و بدون اعشارند. مثل اینها:

```
1 2 199 32442353464675685678
```

اما اینها integral نیستند:

```
1.3 1/2
```

Integer

اینها همون اعدادِ کاملی‌اند که بهشون عادت داریم. می‌تونن مثبت یا منفی باشن، و هر چقدر هم لازم باشه بزرگ یا کوچک میشن.

تایپِ `Bool` فقط دو مقدار داره که میشه صراحتاً به عنوان داده‌ساز بنویسیمِ‌شون. در مورد `Integer`، و بیشتر نوع‌داده‌های عددی که بینهایت مقدار دارن، داده‌سازهاشون نوشته نمیشن. از نظر تئوری، یک `Integer` رو میشه با جمعِ سه حالتِ: صفر، داده‌سازهای بازگشتی (خوداِتکا)[۱] به سمت منفی بینهایت، و داده‌سازهای بازگشتی به سمت مثبت بینهایت تعریف کرد. ولی چنین روشی اصلاً مقرون به صرفه نیست، به همین خاطر، در این مورد GHC یه کم جادو می‌کنه.

چرا `Int` داریم؟

تایپِ عددیِ `Int` در واقع بازمونده‌ای از امکاناتِ اولیه‌ی کامپیوترهاست. اکثر برنامه‌ها باید با `Integer` کار کنن، نه `Int`؛ مگر اینکه برنامه‌نویس

[۱] Recursive

احاطه‌ی کامل به محدودیتهای Int داشته باشه، و اینکه عملکرد یا performanceِ اضافه‌ی حاصل از اونها واقعاً مطلوب باشه.

خطرِ استفاده از Int (و تایپ‌های مشابه مثل Int8، Int16 و غیره) اینه که نمی‌تونن اطلاعاتِ خیلی بزرگ رو بَیان کنن. پس از اونجا که اینها اعدادِ integral هستن، نمی‌تونن در جهت مثبت یا منفی اندازه‌ی دلخواه داشته باشن.

تو این مثال می‌بینید که وقتی از عددی بزرگتر از حدّ Int8 استفاده می‌کنیم، چه اتفاقی میوفته:

```
Prelude> import GHC.Int
Prelude> 127 :: Int8
127
Prelude> 128 :: Int8

<interactive>:11:1: Warning:
    Literal 128 is out of the
      Int8 range -128..127
    If you are trying to write a large
      negative literal,
    use NegativeLiterals
```

```
-128
Prelude> (127 + 1) :: Int8
-128
```

گرامری که اینجا می‌بینید، `:: Int8`، وظیفه‌ی تخصیصِ تایپِ `Int8` به این اعداد رو داره. همونطور که در فصل بعد می‌بینیم، اعداد در باطن چندریختی یا پلی‌مورفیک‌اند، و کامپایلر تا زمانی که مجبور نباشه، هیچ تایپِ معینی بهشون اختصاص نمیده. یه کم عجیب و غیرمنتظره میشد اگه پیش‌فرضِ کامپایلر تخصیص تایپِ `Int8` به همه‌ی اعداد بود. ما هم به همین خاطر با اون گرامر، تایپِ معینِ `Int8` رو به اعدادمون اختصاص دادیم.

همونطور که می‌بینید، ۱۲۷ در بازه‌ی `Int8` هست و موردی نداره. ۱۲۸ بعد از یه اخطار سرریز شد، ۱+۱۲۷ هم سرریز شد و برگشت به کوچکترین مقدارِ عددی. به خاطر محدود بودنِ حافظه‌ای که مقادیر `Int8` می‌تونن اشغال کنن، توانایی تطابق با عددی مثل ۱۲۸ رو هم ندارن (برعکسِ `Integer`). عدد ۸ نشون دهنده‌ی تعداد بیت‌هاییه که

تایپ برای ارائه‌ی اعداد استفاده می‌کنه[۱]. گاهی اوقات ممکنه اندازه‌ی ثابت این تایپ‌ها مفید باشه، ولی عمدتاً `Integer` ارجح‌ه.

بیشترین و کمترین مقادیرِ تایپ‌های عددی رو می‌تونین با استفاده از `minBound` و `maxBound` از تایپکلاسِ `Bounded` پیدا کنین. در زیر چندتا مثال زدیم:

```
Prelude> import GHC.Int
Prelude> :t minBound
minBound :: Bounded a => a
Prelude> :t maxBound
maxBound :: Bounded a => a

Prelude> minBound :: Int8
-128
Prelude> minBound :: Int16
-32768
Prelude> minBound :: Int32
-2147483648
Prelude> minBound :: Int64
-9223372036854775808
```

[۱] تایپ‌های `Int` که اندازه‌ی ثابت دارند، با روش *مکملِ دو* اعدادشون رو ارائه میدن.

```
Prelude> maxBound :: Int8
127
Prelude> maxBound :: Int16
32767
Prelude> maxBoudn :: Int32
2147483647
Prelude> minBound :: Int64
9223372036854775807
```

اگه تایپی، این تایپکلاسِ بخصوص رو داشته باشه، می‌تونین محدوده‌ی مقادیرِ ممکن اون تایپ رو پیدا کنین. در مثال بالا دیدیم که بازه‌ی مقادیرِ قابل ارائه با Int8 از ۱۲۸- تا ۱۲۷ بود.

با دستورِ info:، می‌تونین همه‌ی تایپکلاس‌های یه تایپ رو تو GHCi ببینین. این دستور، تعریفِ نوع‌داده‌ای هم که استعلام کردین رو نشون میده. برای مثال، می‌خوایم بدونیم که آیا تایپِ Int تایپکلاسِ Bounded رو داره یا نه:

```
Prelude> :i Int
data Int = GHC.Types.I# GHC.Prim.Int#
-- Defined in ‘GHC.Enum’
instance Bounded Int
```

البته Int تایپکلاسهای خیلی بیشتری داره که اینجا ننوشتیم.

اعداد کسری

چهار تایپِ رایجِ اعداد Fractional در هسکل Float، Double، Rational، و Scientific اند. Float، Double، و Rational از ابتدا توی GHC هستن. اما Scientific، همونطور که گفتیم از یه کتابخانه میاد. Rational و Scientific هر دو دقتِ نامحدود دارن، ولی Scientific بهینه‌تره. منظور از این نامحدود بودن اینه که از این دو تایپ، بر خلاف Float و Double که به یه دقت مشخص محدودن، میشه برای محاسباتی که نیاز به دقتهای خیلی بالا دارن استفاده کرد. تقریباً هیچ وقت Float لازم نمیشه، مگر برای کارهایی مثل برنامه‌نویسیِ گرافیکی با OpenGL.

بعضی محاسبه‌های عددی با اعداد کسری کار می‌کنن. یه مثالِ خوب، تابع تقسیم (/) ِه، که تایپِش اینه:

```
Prelude> :t (/)
(/) :: Fractional a => a -> a -> a
```

نوشته‌ی `Fractional a =>` یه *محدودیتِ تایپکلاسی*[۱] هِ. یعنی متغیرِ a مقیّد به داشتن تایپکلاس `Fractional`ِ ه. مهم نیست a چه نوع عددی باشه، فقط باید یک نمونه از تایپکلاسِ `Fractional` داشته باشه؛ به عبارت دیگه، باید یک تعریف وجود داشته باشه که عملیاتِ اون تایپکلاس رو برای اون تایپ توصیف کنه. تابعِ `/` یک عددی که تایپکلاس `Fractional` داره رو می‌گیره، به یه عددِ دیگه از همون تایپ تقسیم می‌کنه، و یه مقدار از همون تایپ رو به عنوان جواب برمی‌گردونه.

`Fractional` تایپکلاسیه که تایپ‌هاش ملزم به داشتنِ یک نمونه از تایپکلاس `Num` هستن. در چنین رابطه‌ای، میگیم `Num` یه *سوپرکلاس* از `Fractional`ِ ه. بنابراین، `(+)` و توابع دیگه از تایپکلاس `Num` رو میشه با اعداد `Fractional` استفاده کرد، ولی توابعِ تایپکلاسِ `Fractional` رو نمیشه با همه‌ی تایپ‌های `Num` استفاده کرد.

این نتیجه‌ی استفاده از `(/)` در REPLِ ه:

---

[۱] Typeclass Constraint

```
Prelude> 1 / 2
0.5
Prelude> 4 / 2
2.0
```

دقت کنید با اینکه جوابمون یه عدد کامل بود، باز هم کسری شد. دلیلش اینه که مقادیر `Fractional a => a` به صورت پیش‌فرض به تایپِ `Double` تبدیل میشن. اکثر مواقع، بهتره صراحتاً از `Double` استفاده نکنین، `Scientific` با دقت نامحدودی که داره (میشه گفت داداشِ `Integer` ه!)، گزینه‌ی بهتریه. اکثر مردم با محاسباتِ ممیز شناور، به خاطرِ رفتارهای نامتعارف‌ش، راحت کنار نمیان (اون رفتارها به عمد طراحی شدن، ولی اون یه مبحث دیگه‌س)؛ شما هم برای آسودگیِ خاطر، از تایپ‌های با دقتِ نامحدود استفاده کنین.

۴ – ۵ مقایسه‌ی مقادیر

تا اینجا بیشترِ عملیاتی که با اعداد داشتیم محاسبات بودن. اعداد رو میشه برای تساوی، بزرگتر بودن، یا کوچکتر بودن هم مقایسه کرد:

```
Prelude> let x = 5
Prelude> x == 5
```

```
True
Prelude> x == 6
False
Prelude> x < 7
True
Prelude> x > 3
True
Prelude> x /= 5
False
```

دقت کنید که در خط اول با یک علامت تساوی مقداری برای x تعریف کردیم. پس می‌دونیم که تا آخر این جلسه‌ی REPL همه‌ی x ها مقدارِ ۵ دارن. چونکه از علامت تساوی برای تعاریف استفاده شده، برای *"آیا مساوی هست با"* باید از دو تا علامت تساوی استفاده کنیم. علامتِ =/ معادلِ *"مساوی نیست با"* و بقیه‌ی علائم هم حتماً می‌شناسین.

با توجه به نتیجه‌ی بیانیه‌ها، GHCi دو مقدارِ True یا False رو برمی‌گردوند. قبلاً دیدیم که True و False داده‌سازهای نوع‌داده‌ی Bool بودن. اگه تایپِ هر کدوم از این عملگرها رو از GHCi استعلام کنین، می‌بینین که تایپِ جوابشون Bool ِه:

```
Prelude> :t (==)
(==) :: Eq a => a -> a -> Bool
Prelude> :t (<)
(<) :: Ord a => a -> a -> Bool
```

به محدودیت‌های تایپکلاسی دقت کنین. Eq تایپکلاسیه که شامل هر چیزِ قابل مقایسه برای تساوی میشه؛ Ord هم یه تایپکلاسه که شامل هر چیزِ ترتیب‌دار میشه. و هیچ کدومِشون هم محدود به اعداد نیستن. اعداد رو میشه مقایسه و مرتب کرد، ولی حروف هم همینطور، پس این محدودیتِ تایپکلاس خیلی هم محدودکننده نیست. هر مقداری که بشه براش تساوی یا ترتیب تعریف کرد، می‌تونه ورودی به این توابعِ تساوی و مقایسه باشه. مابقی اطلاعات تایپ به ما میگه که تابع یکی از این مقادیر رو می‌گیره، با یه هم‌نوعِ خودش مقایسه می‌کنه، و یه Bool برمی‌گردونه. همونطور که قبلاً هم دیدیم، مقادیرِ Bool، True یا False اند.

یه کم با مقادیر دیگه بازی کنیم:

```
Prelude> 'a' == 'a'
True
```

```
Prelude> 'a' == 'b'
False
Prelude> 'a' < 'b'
True
Prelude> 'a' > 'b'
False
Prelude> 'a' == 'A'
False
Prelude> "Julie" == "Chris"
False
```

می‌دونیم که حروف الفبا ترتیب دارن، با این حال معمولاً ’a‘ رو "کوچکتر" از ’b‘ نمیگیم. به جاش میگیم که در الفبا *’a‘ قبل از ’b‘ میاد*. در مثال آخر دیدیم که با نوشته هم همین قاعده کار می‌کنه. GHCi به درستی تساوی اون دو نوشته رو نقض کرده.

حالا با REPL ببینید ’a‘ بزرگتره یا ’A‘.

بعد مثال‌های زیر رو بررسی کنید. می‌تونین دلیل جواب‌های REPL رو تشخیص بدین؟

```
Prelude> "Julie" > "Chris"
True
```

```
Prelude> "Chris" > "Julie"
False
```

خوبه که هسکل هم واقعیت رو میگه! "Julie" از "Chris" بزرگتره چون J > C، اگه دو لغتِ "Back" و "Brack" رو مقایسه می‌کردیم، اون موقع حرف اول رد میشد (چون 'B' == 'B') و با مقایسه‌ی حرف‌های دوم، "Brack" بزرگتر میشد ('a' < 'r' با ترتیب الفبایی). دقت کنید که این مقایسه بر مبنای نمونه‌ی تایپکلاسِ Ord برای هر دو تایپِ لیست و Char انجام شد. فقط لیست‌هایی رو میشه با هم مقایسه کرد که المان‌هاشون هم نمونه‌ی Ord داشته باشن. لیست‌های زیر قابل مقایسه‌اند، چون Char و Integer هر دو یکی یه نمونه از Ord دارن:

```
Prelude> ['a', 'b'] > ['b', 'a']
False
Prelude> 1 > 2
False
Prelude> [1, 2] > [2, 1]
False
```

نوع‌داده یا datatype ی که نمونه‌ی Ord نداره با این توابع کار نمی‌کنه:

```
Prelude> data Mood = G | B deriving Show
Prelude> [G, B]
[G, B]
Prelude> [G, B] > [B, G]

<interactive>:28:14:
    No instance for (Ord Mood) arising
      from a use of ‘>’
    In the expression: [G, B] > [B, G]
    In an equation for ‘it’:
      it = [G, B] > [B, G]
```

معنی پیغامِ `No instance for (Ord Mood)` اینه که نمونه‌ی `Ord` وجود نداره، پس نمیدونه چطور مقادیر رو مرتب کنه.

چیز دیگه‌ای که با این توابع کار نمی‌کنه:

```
Prelude> "Julie" == 8

<interactive>:38:12:
    No instance for (Num [Char]) arising
      from the literal ‘8’

    In the second argument of ‘(==)’,
      namely ‘8’
```

```
In the expression: "Julie" == 8
In an equation for ‘it’:
  it = "Julie" == 8
```

بالاتر گفتیم که توابعِ مقایسه، چندریختی اند و متعاقباً می‌تونن با تایپ‌های متنوعی کار کنند. چیز دیگه‌ای که گفتیم این بود که طبق اطلاعات تایپ‌شون که دیدیم، فقط تایپ‌های یکسان رو قبول می‌کردند. به محض اینکه یک مقدارِ String (در سطح جمله‌ای)، مثل "Julie" به چنین توابعی میدیم، تایپِ تابع تعیین میشه، و آرگومانِ دوم هم باید از همون تایپ باشه. پیغام خطای بالا میگه که تایپِ مقدارِ لیترالِ[1] ۸ با تایپِ مقدار اول منطبق نیست، که بر خلافِ انتظارِ این تابعه.

۴ – ۶ من رو بول بزن

نوع‌داده‌ی Bool از Prelude میاد، و همونطور که دیدیم، یک تایپِ جمع یا sum type با دو داده‌سازه:

```
data Bool = False | True
```

[1] Literal (یا لفظ)

این تعریف، یک نوع‌داده با نوع‌سازِ `Bool` درست می‌کنه. نوع‌سازها در تایپ سیگنچرها نوشته میشن، و نه تویِ بیانیه‌های سطحِ جمله‌ای. نوع‌سازِ `Bool` هیچ آرگومانی نمی‌گیره (بعضی نوع‌سازها می‌گیرن). تعریفِ بالا دو تا داده‌ساز هم ایجاد می‌کنه، `True` و `False`. هر تابعی که مقادیرِ تایپ `Bool` قبول می‌کنه، حتماً باید هر دو حالتِ `True` یا `False` رو در نظر بگیره؛ در تایپِ تابع نمیشه تعیین کرد که فقط یک مقدار رو قبول کنه. اگه بخوایم تعریفِ بالا رو به کلام بگیم، اینطور میشه: "نوع‌داده‌ی `Bool` با دو مقدارِ `True` یا `False` ارائه میشه."

یادتون باشه که تایپ هر مقداری رو میشه از GHCi پرسید، مثلِ توابع:

```
Prelude> :t True
True :: Bool
Prelude> :t "Julie"
"Julie" :: [Char]
```

یه کم با `Bool` سرگرم شیم. اول `not` رو دوره کنیم:

```
Prelude> :t not
not :: Bool -> Bool
```

```
Prelude> not True
False
```

True و False رو با حرفِ اولِ بزرگ می‌نویسیم، چون داده‌ساز اند. اگه not رو به اونها بدونِ حرفِ بزرگ اعمال کنین چی میشه؟

یه چیز دیگه رو امتحان کنیم، یه ذره پیچیده‌تر:

```
Prelude> let x = 5
Prelude> not (x == 5)
False
Prelude> not (x > 7)
True
```

می‌دونیم که توابعِ قیاس به مقادیر Bool ساده میشن، پس می‌تونن آرگومانِ not بشن.

حالا عملگرهای میانوندِ منطق بولی رو نگاه کنیم، و ببینیم چطور می‌تونیم از Bool و این توابع استفاده کنیم.

اول از همه، (&&) عملگر میانوندِ عطف بولی‌ه. مثال اول رو اینطور می‌خونیم، "true و true."

```
Prelude> True && True
True
Prelude> (8 > 4) && (4 > 5)
False
Prelude> not (True && True)
False
```

عملگر میانوند برای فصل بولی، (||)ِه. پس خطِ اول "false یا true" خونده میشه:

```
Prelude> False || True
True
Prelude> (8 > 4) || (4 > 5)
True
Prelude> not ((8 > 4) || (4 > 5))
False
```

با دستور i: در GHCi، می‌تونیم اطلاعاتِ نوع‌داده‌هایی که در گستره هستن رو نگاه کنیم (اگه در گستره نباشن، باید ماژول‌شون رو وارد کنیم). گستره یا scope یعنی جاهایی که یک اسمِ تعریف شده برای یه بیانیه اعتبار داره. وقتی یه چیزی در گستره‌ست، یعنی میشه از اون بیانیه به واسطه‌ی اسمش استفاده کرد؛ حالا یا به خاطرِ اینکه توی

تابع یا ماژولِمون تعریف شده، یا به خاطر اینکه وارد (import) شده. پس در برنامه‌مون مرئیِه. همه‌ی چیزهایی که در Preludeِ هسکل درست شدن، خودبه‌خود وارد میشن و در گستره قرار می‌گیرن. فعلاً فقط همین‌ها رو لازم داریم و لازم نیست همه‌ی تابع‌هامون رو از اول بنویسیم.

تمرین‌ها: مشکلات رو پیدا کن

بعضی از کدهای زیر ایراد دارن – اصلاً کامپایل نمیشن! میدونین چی کار کنین.

۱. `not True && true`

۲. `not (x = 6)`

۳. `(1 * 2) > 5`

۴. `[Merry] > [Happy]`

۵. `[1, 2, 3] ++ "look at me!"`

شروط با `if-then-else`

هسکل، دستورِ if نداره (*if statement*)، ولی *بیانیه‌ی* if داره (*if expression*). یه گرامر داخلی‌ه، که با نوع‌داده‌ی `Bool` کار می‌کنه.

```
Prelude> let t = "Truthin'"
Prelude> let f = "Falsin'"
Prelude> if True then t else f
"Truthin'"
```

بیانیه‌ی `if True` به `True` محاسبه میشه، پس t رو برمی‌گردونیم.

```
Prelude> if False then t else f
"Falsin'"
Prelude> :t if True then t else f
if True then "Truthin'" else "Falsin'"
  :: [Char]
```

و `if False` به `False` ساده میشه، پس مقدار `else` رو برمی‌گردونیم. تایپِ کلِ بیانیه `String` (یا `[Char]`) ِه، همون تایپی که تایپِ مقدارِ نهاییِ بیانیه‌س.

ساختار اینطوریه:

```
If CONDITION
then EXPRESSION_A
else EXPRESSION_B
```

اگه شرط یا CONDITION (که حتماً باید به یه Bool ساده بشه) مقدارِ True داشته باشه، EXPRESSION_A نتیجه میشه، در غیر اینصورت، EXPRESSION_B.

میشه به بیانیه‌های if به چشمِ یه راهی برای انتخابِ بین دو مقدار نگاه کرد. تنها محدودیتی که برای بیانیه‌های جلوی if وجود داره، اینه که باید به یه مقدارِ Bool ساده بشن. تایپِ بیانیه‌های جلوی then و else هم باید منطبق باشن، مثل زیر:

```
Prelude> let x = 0
Prelude> let a = "AWESOME"
Prelude> let w = "wut"
Prelude> if (x + 1 == 1) then a else w
"AWESOME"
```

مراحل ساده شدنش از این قراره:

```haskell
-- داریم:
x = 0

if (x + 1 == 1) then "AWESOME" else "wut"
-- صفر است x

if (0 + 1 == 1) then "AWESOME" else "wut"
-- رو ساده می‌کنیم تا ببینیم 0 + 1
-- آیا برابر 1 هست یا نه

if (1 == 1) then "AWESOME" else "wut"
-- آیا 1 برابر 1 هست؟

if True then "AWESOME" else "wut"
-- یکی رو براساس مقدار Bool انتخاب کن

"AWESOME"
-- حله!
```

ولی این کار نمی‌کنه:

```
Prelude> let dog = "adopt a dog"
Prelude> let cat = "or a cat"
Prelude> let x = 0
Prelude> if x * 100 then dog else cat
```

```
<interactive>:15:7:
    No instance for (Num Bool) arising
    from a use of ‘*’

    In the expression: (x * 100)
    In the expression:
        if (x * 100)
          then "adopt a dog"
          else "or a cat"
    In an equation for ‘it’:
        it = if (x * 100)
               then "adopt a dog"
               else "or a cat"
```

دلیل این خطا، تایپِ شرطیِه که به بیانیه‌ی if دادیم، `Num a => a`، که `Bool` نیست و تایپکلاسِ `Num` هم برای `Bool` تعریف نشده. ساده‌تر بگیم، `(x * 100)` به یه عدد ساده میشه و اعداد مقادیر واقعی (truth values) نیستن. اگه یه چیزی مثل `x * 100 == 0` یا `x * 100 == 9001` بود، اون موقع کار می‌کرد، چون بررسیِ یه تساوی بود و به یه `Bool` ساده میشد.

مثال زیر یه تابعه که از یه مقدار Bool در بیانیه‌ی if استفاده می‌کنه:

```
-- greetIfCool1.hs
module GreetIfCool1 where

greetIfCool :: String -> IO ()
greetIfCool coolness =
  if cool
    then putStrLn "eyyyyy. What's shakin'?"
  else
    putStrLn "pshhhh."
  where cool = coolness == "downright frosty yo"
```

اگه تو REPL چک کنین، چنین چیزی میشه:

```
Prelude> :l greetIfCool1.hs
[1 of 1] Compiling GreetIfCool1
Ok, modules loaded: GreetIfCool1.
Prelude> greetIfCool "downright frosty yo"
eyyyyy. What's shakin'?
Prelude> greetIfCool "please love me"
pshhhh.
```

اگه می‌خواستین، میشد cool ِداخل greetIfCool رو، بجای یه مقدار که مستقیماً با آرگومانِ تابع تعریف شده، به عنوان یه تابع بنویسین. اینطوری:

```
-- greetIfCool2.hs
module GreetIfCool2 where

greetIfCool :: String -> IO ()
greetIfCool coolness =
  if cool coolness
    then putStrLn "eyyyyy. What's shakin'?"
  else
    putStrLn "pshhhh."
  where cool v = v == "downright frosty yo"
```

۴ – ۷ توپل‌ها

چندتایی[۱] تایپی‌ه که امکانِ استفاده از چند مقدار داخلِ یک مقدار رو ممکن می‌کنه. چندتایی‌ها گرامرِ خاصی دارن که هم در سطح جمله‌ای و هم در سطح نوعی نوشته میشه، و هر توپِل هم تعداد اجزاء ثابتی داره.

---

[۱] Tuple

چندتایی‌ها رو براساس تعداد مقادیر داخل‌شون شناسایی می‌کنیم: برای مثال، توپلِ دوتایی یا جفت، دو مقدار داخل‌ش هست، `(x,y)`؛ توپلِ سه‌تایی یا سه‌گانه، سه مقدار، `(x,y,z)`؛ و همینطور بقیه‌ی چندتایی‌ها. به این تعدادِ مقادیر چندتایی‌ها، *آریتیِ*[1] توپِل هم میگن. جلوتر می‌بینیم که لازم نیست مقادیر داخل چندتایی‌ها از یک تایپ باشند.

با یه جفت شروع می‌کنیم، توپِلی که دو تا اِلمان داره. توپلِ دوتایی در هر دو سطحِ جمله‌ای و نوعی با سازنده‌ی `(,)` نشون داده میشه. تایپ‌ش اینطوری تعریف شده:

```
Prelude> :info (,)
data (,) a b = (,) a b
```

علاوه بر گرامرِ خاص‌ش، فرق‌های مهمی بین این تعریف با تعریفی مثل `Bool` وجود داره. اول اینکه دو تا پارامتر داره، که با متغیرهای تایپیِ a و b معلوم شدن. همونطور که در سطحِ جمله‌ایِ کد، پارامترهای توابع بعد از اعمال شدن به آرگومان‌ها با مقادیرِ معین جایگزین میشن،

[1] Arity

این پارامترهای تایپی هم باید به تایپ‌های معین اعمال بشن. تفاوتِ اساسی دوم اینه که تعریف بالا یه تایپِ ضربِ[۱]ه، نه یه تایپِ جمع مثل Bool. تایپِ ضرب معادلِ عطفِ منطقیه، یعنی برای ساختِ یه مقدار، باید هر *دو* آرگومان رو براش تأمین کنید.

دقت کنید که در بالا، دو تا متغیرِ تایپ با هم فرق دارن؛ پس یه توپلِ دوتایی، می‌تونه دو مقدار با تایپ‌های متفاوت رو نگه داره. با این حال، این تفاوت *الزامی* نیست:

```
λ> (,) 8 10
(8,10)
λ> (,) 8 "Julie"
(8,"Julie")
λ> (,) True 'c'
(True,'c')
```

اگه فقط به یه آرگومان اعمالش کنیم:

```
λ> (,) 9
```

---

[۱] Product Type

```
<interactive>:34:1:
No instance for (Show (b0 -> (a0, b0)))
  (maybe you haven't applied enough
    arguments to a function?)
  arising from a use of ‘print’,
In the first argument of ‘print’,
  namely ‘it’
In a stmt of an interactive
  GHCi command: print it
```

عجب خطایی... به زودی با جزئیات بررسی‌ش می‌کنیم. اما فعلاً پرانتزِ دوم برامون مهمه:

```
maybe you haven't applied enough
arguments to a function?
```

شاید آرگومانهای کافی به یه تابع ندادین؟

تابع‌مون — در این مورد داده‌سازِ (,) — رو به آرگومانهای کافی اعمال نکردیم.

در هسکل، بطور پیش‌فرض چند تابعِ استاندارد برای گرفتن مقدار اول و مقدار دومِ توپل‌های دوتایی تعریف شدن، `fst` و `snd`:

```
fst :: (a, b) -> a
snd :: (a, b) -> b
```

از تایپ سیگنچرِشون مشخصه که اون دو تابع هیچ کاری غیر از برگردوندنِ اولین و دومین مقدار نمی‌کنن.

اینجا چند مثال از کارهایی که با توپل‌ها میشه انجام داد آوردیم:

```
Prelude> let myTup = (1 :: Integer, "blah")
Prelude> :t myTup
myTup :: (Integer, [Char])
Prelude> fst myTup
1
Prelude> snd myTup
"blah"
Prelude> import Data.Tuple
Prelude> swap myTup
("blah",1)
```

تابع `swap` در `Prelude` تعریف نشده، برای همین `Data.Tuple` رو وارد کردیم.

توپل‌ها رو با بیانیه‌های دیگه هم میشه ترکیب کرد:

```
Prelude> 2 + fst (1, 2)
3
Prelude> 2 + snd (1, 2)
4
```

گرامرِ توپل‌ها، `(x,y)`، گرامر خاصیه. سازنده‌ها[1]یی که برای توپل‌ها استفاده می‌کنین، چه در تایپ سیگنچرها و چه در خودِ کد (سطح جمله‌ای)، با اینکه دو کاربرد متفاوت دارن، ولی گرامرِشون یکسانه. این نوع‌ساز رو گاهی اوقات به تنهایی می‌بینید، یعنی `(,)` بدون متغیرهای نوعی‌ش؛ بقیه‌ی مواقع هم (بخصوص در تایپ سیگنچرها) اینطوری `(a,b)`.

در نوشتنِ توابع هم می‌تونین از اون گرامر برای *تطبیق الگو* (*pattern matching*) استفاده کنین. یکی از خوبی‌هاش اینه که بعضی اوقات تعریف تابع خیلی شبیه تایپ سیگنچرِش میشه. برای مثال، توابع `fst` و `snd` رو خودمون می‌نویسیم:

---

[1] م. سازنده یا constructor، برای اطلاق به data constructor و type constructor استفاده شده.

```
fst' :: (a, b) -> a
fst' (a, b) = a

snd' :: (a, b) -> b
snd' (a, b) = b
```

یه مثال دیگه برای تطبیق الگو روی توپل‌ها:

```
tupFunc :: (Int, [a])
        -> (Int, [a])
        -> (Int, [a])
tupFunc (a, b) (c, d) =
  ((a + c), (b ++ d))
```

استفاده از توپل‌های زیادی بزرگ کار عاقلانه‌ای نیست، بازدهیِ مناسبی هم نداره. بیشتر توپل‌هایی که می‌بینین (,,,,) (توپلِ پنج‌تایی) یا کوچکتراند.

۴ – ۸ لیست‌ها

یه تایپ دیگه‌ای که برای نگهداریِ چند مقدار توی یک مقدار استفاده میشه، لیسته. ولی لیست‌ها سه تا فرق اساسی با توپل‌ها دارن: اول اینکه همه‌ی المان‌های یه لیست باید از یک تایپ باشن. دوم، لیست‌ها

گرامر مخصوص خودشون رو دارن، []؛ مثل گرامرِ توپل‌ها، هم برای نوع‌سازها در تایپ سیگنچرها استفاده میشه، و هم در سطح جمله‌ای برای بیانِ لیست مقادیر. سوم، تعداد مقادیری که در لیست قرار می‌گیرن، در تایپِ لیست مشخص نمیشه – بر عکسِ توپل که آریتیِ اون در تایپ‌ِش تعیین میشد و تغییرناپذیر[۱] بود.

یه مثال در REPL:

```
Prelude> let p = "Papuchon"
Prelude> let awesome = [p, "curry", ":)"]
Prelude> awesome
["Papuchon","curry",":)"]

Prelude> :t awesome
awesome :: [[Char]]
```

awesome، یه لیستی از لیست‌های با مقادیرِ Char ِه (یا یه لیستی از String، چون String تایپِ مستعارِ [Char] ِه). پس همه‌ی توابع و عملگرهایی که با لیست‌های هر تایپی (که با [a] نشون داده میشن)

[۱] Immutable

کار می‌کنن، با String هم کار می‌کنن چون [Char] حالت خاصی از [a] ِه.

```
Prelude> let s = "The Simons"
Prelude> let also = ["Quake", s]
Prelude> :t (++)
(++) :: [a] -> [a] -> [a]
Prelude> awesome ++ also
["Papuchon",
 "curry",
 ":)",
 "Quake",
 "The Simons"]
Prelude> let allAwesome = [awesome, also]
Prelude> allAwesome
[["Papuchon", "curry", ":)"],
 ["Quake", "The Simons"]]
Prelude> :t allAwesome
allAwesome :: [[[Char]]]
Prelude> :t concat
concat :: [[a]] -> [a]
Prelude> concat allAwesome
["Papuchon",
 "curry",
 ":)",
```

```
  "Quake",
  "The Simons"]
```

توضیحاتِ کامل لیست‌ها رو برای فصل‌ش نگه میداریم. لیست‌ها پیچیدگیهای جالبی دارن و ما هم به کمک اونها، مثال‌هایی از محاسبه‌ی ناآکید (nonstrict) ِهسکل رو نشون میدیم. از طرف دیگه، توابع و ساختارهای استانداردِ *خیلی زیادی* هستن که با لیست‌ها کار می‌کنن؛ به همین خاطر یه فصلِ کامل رو به لیست‌ها اختصاص دادیم.

۴ – ۹ تمرین‌های فصل

مثل فصل‌های قبل، باز هم پیشنهاد می‌کنیم قبل از اجرای کدها در REPL، اول به جواب برسین؛ اما جوابتون رو حتماً با REPL چک کنین. برای تمرین‌های زیر، مقادیر `awesome`، `also`، و `allAwesome` که بالاتر تعریف کردیم رو هم لازم دارین. برای راحتی اینجا هم نوشتیم:

```
awesome = ["Papuchon", "curry", ":)"]
also = ["Quake", "The Simons"]
allAwesome = [awesome, also]
```

`length` تابعی‌ه که یه لیست می‌گیره و عددِ خروجی‌ش تعداد المان‌های اون لیست رو نشون میده.

۱. با توجه به تعریف بالا برای `length`، تایپ سیگنچرِش چی میشه؟ چندتا آرگومان (و با چه تایپی) می‌گیره؟ جواب نهاییِ این تابع از چه تایپی میشه؟

۲. جواب بیانیه‌های زیر چی میشن؟

a) `length [1, 2, 3, 4, 5]`

b) `length [(1, 2), (2, 3), (3, 4)]`

c) `length allAwesome`

d) `length (concat allAwesome)`

۳. یکی از این دو بیانیه‌ی زیر کار می‌کنه، ولی اون یکی خطا میده. با توجه به تایپ تابع `length` و اطلاعاتی که از تایپ‌های عددی داریم، بگین کدوم یکی و به چه دلیلی خطا میده.

(نکته: مثل تابع concat در فصل قبل، اینجا هم برای length تایپ Foldable t => t a رو – بجای [a] – می‌بینین. در واقع لیست فقط یکی از تایپ‌های ممکنه، ولی باز Foldable t رو لیست فرض کنید.)

```
Prelude> 6 / 3
-- و
Prelude> 6 / length [1, 2, 3]
```

۴. چطور می‌تونین با استفاده از یه تابع/اوپراتورِ تقسیمِ دیگه، کد خرابِ بالا رو درست کنین؟

۵. تایپ بیانیه‌ی 2 + 3 == 5 چیه؟ فکر می‌کنین جوابش چیه؟

۶. تایپ و جواب کد زیر چیه؟

```
Prelude> let x = 5
Prelude> x + 3 == 5
```

۷. بین کدهای زیر، کدوم‌ها کار می‌کنن؟ چرا کار می‌کنن/نمی‌کنن؟ اگه جواب دارن، به چه مقداری ساده میشن؟

```
Prelude> length allAwesome == 2
Prelude> length [1, 'a', 3, 'b']
Prelude> (8 == 8) && ('b' < 'a')
Prelude> (8 == 8) && 9
```

۸. یک تابع بنویسین که بگه آیا string (یا لیست) ورودی واروخوانه[۱] هست یا نه. تابعِ از پیش تعریف شده‌ی reverse (م. به معنای معکوس) رو لازم دارین:

```
reverse :: [a] -> [a]
reverse "blah"
"halb"

isPalindrome :: (Eq a) => [a] -> Bool
isPalindrome x = undefined
```

۹. با استفاده از if-then-else تابعی بنویسین که قدر مطلق یه عدد رو برگردونه.

```
myAbs :: Integer -> Integer
myAbs = undefined
```

---

[۱] Palindrome

۱۰. با توابع `fst` و `snd`، توصیف تابع زیر رو بنویسین:

```
f :: (a, b) -> (c, d) -> ((b, d), (a, c))
f = undefined
```

تصحیح گرامر

مثال‌های زیر ایرادهای گرامری دارن. سعی کنید در برنامه‌ی ویرایش متن (یا text editor) اِنتخابی‌تون، اونها رو تصحیح کنین، و بعد با GHC یا GHCi چک کنین.

۱. اینجا تابعی می‌خوایم که یکی به طولِ آرگومان stringِ ش اضافه کنه، و نتیجه رو برگردونه.

```
x = (+)

F xs = w 'x' 1
    where w = length xs
```

۲. این قراره تابع همانی، `id`، باشه.

```
\X = x
```

۳. نسخه‌ی سالمِ این تابع، با ورودیِ (1,2)، مقدار ۱ رو برمی‌گردونه.

```
f (a b) = A
```

اسم توابع رو با تایپ‌شون تطبیق بدین

۱. کدوم یکی تایپِ show ـه؟

a) `show a => a -> String`

b) `Show a -> a -> String`

c) `Show a => a -> String`

۲. کدوم یکی تایپِ (==) ـه؟

a) `a -> a -> Bool`

b) `Eq a => a -> a -> Bool`

c) `Eq a -> a -> a -> Bool`

d) `Eq a => A -> Bool`

۳. کدوم یکی تایپِ `fst`ِه؟

a) `(a, b) -> a`

b) `b -> a`

c) `(a, b) -> b`

۴. کدوم یکی تایپِ `(+)`ِه؟

a) `(+) :: Num a -> a -> a -> Bool`

b) `(+) :: Num a => a -> a -> Bool`

c) `(+) :: num a => a -> a -> a`

d) `(+) :: Num a => a -> a -> a`

e) `(+) :: a -> a -> a`

۴ – ۱۰ تعاریف

۱. چند*تایی*، *توپِل* یا *tuple* یک گروهِ ترتیب‌دار از مقادیره. در هسکل، توپل رو با یک المان نمیشه درست کرد، ولی توپل با

صفر المان، به اسم *واحد*[۱] و با نماد ()، تعریف شده. تایپِ المان‌های توپل‌ها ممکنه متفاوت باشن، پس برای مثال هم `(String, String)` صحیحه، و هم `(Integer, String)`. معمولاً از توپل‌ها برای نگهداریِ موقت و بدون اسمِ چند مقدار استفاده میشه.

۲. *تایپکلاس* یا *typeclass* مجموعه‌ای از عملیاته که بر پایه‌ی یک تایپِ چندریختی (یا پلی‌مورفیک) تعریف شدن. اگه تایپی یک نمونه از یه تایپکلاس رو داشته باشه، میشه از عملیاتِ تعریف شده در اون تایپکلاس برای مقادیرِ اون تایپ استفاده کرد. در هسکل، جفتِ تایپکلاس و نمونه‌ش یکتاست. منظور از این حرف اینه که اگه یه تایپِ a یک نمونه از `Eq` داشته باشه، *فقط* یک نمونه `Eq` داره.

۳. *داده‌ساز* یا *data constructor* ها در هسکل روشی برای ساخت مقادیری از یک تایپ هستن. داده‌سازهای هسکل یه تایپ دارن،

---

[۱] Unit

و ممکنه مقدارِ ثابتی باشن (پوچ‌گانه[۱]) یا مثلِ توابع، یک آرگومان یا بیشتر بگیرن. در مثال زیر، Cat یه داده‌سازِ پوچ‌گانه یا nullary برای Pet، و Dog داده‌سازیه که یک آرگومان می‌گیره:

```
-- گربه که جواب نمیده،
-- اسم میخواد چی کار؟
type Name = String

data Pet = Cat | Dog Name
```

این داده‌سازها تایپ‌های زیر رو دارن:

```
Prelude> :t Cat
Cat :: Pet
Prelude> :t Dog
Dog :: Name -> Pet
```

۴. *نوع‌ساز* یا *type constructor* ها در هسکل مقدار نیستن، و فقط در تایپ سیگنچرها کاربرد دارن. یک تعریفِ داده، همونطور که

[۱] م. زین پس بجای واژه‌ی غریب و... هیچی... پیشنهاد بنده برای nullary.

داده‌سازهایی رو برای ساختِ مقادیرِ یک تایپ تعریف می‌کنه، نوع‌سازهایی هم برای اشاره به اون تایپ ایجاد می‌کنه. در مثالِ بالا، `Pet` نوع‌سازِ ایجاد شده در اون تعریفه. راه ساده برای تشخیص بین داده‌سازها و نوع‌سازها اینه که در یک تعریفِ داده، نوع‌سازها همیشه سمت چپِ تساوی نوشته میشن.

۵. *تعریفِ داده* یا *data declaration* ها، نوع‌داده (تایپ) های جدید تعریف می‌کنن. تعاریفِ داده همیشه یک نوع‌سازِ جدید میسازن، ولی *الزاماً* داده‌ساز جدید ایجاد *نمی‌کنن*. تعریف داده، در واقع اسم کلِ جمله‌ایه که با کلیدواژه‌ی `data` شروع میشه.

۶. *تایپِ مستعار* یا *type alias* راهی برای اشاره به یک نوع‌ساز یا تایپِ ثابت (constant type) با یه اسم دیگه‌ست. معمولاً برای خوانایی بیشترِ کد، یا خلاصه‌نویسی از این تایپ‌های مستعار استفاده می‌کنیم.

```
type Name = String
-- یک نام مستعار به اسم Name برای
```

```
-- تایپ String؛ تعریف داده *نیست*
-- فقط تعریف برای یه تایپ مستعاره
```

۷. *آریتی* یا *Arity* تعداد آرگومان‌هاییه که یه تابع قبول می‌کنه. البته این واژه در هسکل، کمی بی‌معنی‌ه، چرا که به خاطر currying همه‌ی توابع تک-آریتی اند. توابعِ چند آرگومانی با چند تابعِ تودرتو معادل میشن.

۸. چندریختی، *پلی‌مورفیسم* یا *polymorphism* در هسکل یعنی بتونیم بر مبنای مقادیری که ممکنه تایپ‌شون یکی از چندتا، یا همه‌ی تایپ‌ها باشن، کُد بنویسیم. پلی‌مورفیسم در هسکل یا پارامتری[۱]ِه یا محدود[۲]. تابع همانی، `id`، مثالی از پلی‌مورفیسمِ پارامتری‌ه:

```
id :: a -> a
id x = x
```

---

[۱] Parametric

[۲] Constrained

اینجا `id` از هیچ اطلاعاتی مختصِ یه تایپ، یا مجموعه‌ای از تایپ‌ها استفاده نمی‌کنه، و با همه‌ی تایپ‌ها کار می‌کنه. ولی تابع زیر:

```
isEqual :: Eq a => a -> a -> Bool
isEqual x y = x == y
```

یه تابع پلی‌مورفیکِ محدود یا مقیّد[۱] به مجموعه‌ای از تایپ‌هاست که یه نمونه از تایپکلاس `Eq` داشته باشن. در یکی از فصل‌های آینده، انواع پلی‌مورفیسم رو با جزئیات توضیح میدیم.

۴ – ۱۱ اسم‌ها و متغیرها

اسم‌ها

در هسکل، هفت رده (یا کاتگوری) از اشیائی که اسم دارن وجود داره: توابع، متغیرهای سطحِ جمله‌ای، داده‌سازها، متغیرهای تایپی، نوع‌سازها، تایپکلاس‌ها، و ماژول‌ها. متغیرهای سطح جمله‌ای و داده‌سازها در

[۱] Bounded

جملات‌اند. *سطح جمله* جاییه که مقادیر و کُدی که به همراه برنامه اجرا میشه وجود دارن. در *سطح نوعی* (که برای آنالیزِ ایستا[۱] و تأیید برنامه استفاده میشه) متغیرهای تایپی، نوع‌سازها، و تایپکلاس‌ها رو داریم. و در نهایت، برای سازماندهیِ کُدِمون به گروه‌های مرتبط بین فایل‌های مختلف، ماژول‌ها رو داریم.

رسم و رسوم برای متغیرها

متغیرها زیاد کاربرد دارن، و به همین دلیل رسم و رسوم‌هایی[۲] به وجود اومدن. واجب نیست اینها رو حفظ کنین، چون قاعده و قانون نیستن، ولی دونستن‌شون به خوندنِ کد هسکل کمک می‌کنه.

متغیرهای تایپی (که در تایپ سیگنچرها نوشته میشن) معمولاً از a شروع میشن و ادامه پیدا می‌کنن: a، b، c، و الی آخر. ممکنه گاهی اوقات یه عدد هم جلوشون بیاد، مثل $a1$.

---

[۱] Static Analysis

[۲] Conventions

وقتی توابع به عنوان آرگومان استفاده میشن، عموماً با حرف f و حرف‌های بعدش نامگذاری میشن (یعنی g، h، و غیره). گاهی اوقات با عدد (مثل $f1$)، یا کاراکترِ $'$ هم دیده میشن، مثل f' (میخونیم اِف-پریم[۱]) که معمولاً زمانی کاربرد داره که f' رابطه‌ی نزدیکی با f داره یا یه تابع کمکی‌ه. توابع ممکنه با حروف دیگه هم برای کمک به یادآوریِ کاری که انجام میدن، نامگذاری بشن؛ مثل p برای تابعی که اعداد اول[۲] رو برمی‌گردونه، یا txt برای تابعی که متنی رو از جایی برمیداره.

اسم متغیرها نباید حتماً تک حرفی باشن. تو برنامه‌های کوچیک معمولاً هستن؛ ولی تو برنامه‌های بزرگ بهتره که تک حرفی نباشن. وقتی تعداد متغیرها زیاد میشه (که زیاد هم اتفاق میوفته) خوبه که اسم‌های "توصیف‌کننده" داشته باشن. پیشنهاد هم میشه که برای هر زمینه‌ای، از اصطلاحات همون زمینه برای اسم متغیرها استفاده بشه.

---

[۱] م. یا اف-پرایم.

[۲] Prime Numbers

پارامترِ توابع معمولاً با x و حروف بعد از اون نامگذاری میشن، و باز هم ممکنه با اعداد دیده بشن، $x1$. بقیه‌ی حروف هم برای متغیرهای تک-حرفی استفاده میشن، مثل r برای شعاع دایره.

اگه لیستی دارین که چیزهای داخل‌ش رو با x نشون دادین، عُرف اینه که خودِ لیست رو با xs، که حالت جمعِ xِه (sِ جمع) نمایش بدیم. این نوع نگارش رو معمولاً با این حالت می‌بینین: (x:xs)، که یعنی لیستی داریم که سَرِش با x و مابقی‌ش با xs مشخص شدن.

باز هم تکرار می‌کنیم که اینها فقط عُرفِ نگارشی‌اند، و اصلاً قاعده برای کدنویسیِ هسکل به حساب نمیان. ما بیشترِ کُدهامون رو با همین سبک نوشتیم، ولی همه‌ی کدهای هسکل که بعداً می‌بینین اینطور نیستن. استفاده از x بجای a برای یک متغیرِ تایپ مشکلی ایجاد نمی‌کنه (اینجا هم مثل جبر لاندا، اسم‌ها به خودیِ خود مفهومی ندارن). اینها رو بیان کردیم که بیشتر با رسم و رسومِ رایجِ هسکل آشنا بشین.