دانشگاه تهران
پردیس دانشکده‌های
فنی
دانشکده مهندسی برق و
کامپیوتر

عنوان
تعریف معیار پوشش توصیف تابعی برای آزمودن برنامه های امری

پایان‌نامه برای دریافت درجه کارشناسی
در رشته مهندسی کامپیوتر گرایش نرم افزار

ایمان مرادی
شماره دانشجویی
810196560

استاد راهنما:
دکتر رامتین خسروی

شهریورماه ۱۴۰۰


بسم الله الرحمن الرحيم

تعهدنامه اصالت اثر

باسمه تعالی

اینجانب ایمان مرادی تائید می کنم که مطالب مندرج در این پایان نامه حاصل تلاش اینجانب است و به دستاوردهای پژوهشی دیگران که در این نوشته از آنها استفاده شده است مطابق مقررات ارجاع گردیده است. این پایان نامه قبلاً برای احراز هیچ مدرک هم سطح یا بالاتر ارائه نشده است.

نام و نام خانوادگی دانشجو :

امضای دانشجو :

تشکر و قدردانی

.....

لازم اســت که در این آغاز از اســتاد راهنمایم جناب دکتر رامتین خســروی که با صــبر و شکیبایی رهنمون من در این پروژه بودند کمال تشکر را داشته باشم.

چکیده[1]

همواره یکی از معیار های توسعه ی نرم افزار در طول عمر برنامه نویسی، تست نرم افزار جهت بررسی درستی، کشف خطا، و درجه اطمینانی از صحت عملکرد محصول بوده است. چنانچه امروز، تست نرم افزار بخش غیرقابل جدایی از توسعه ی نرم افزار بوده و با به وجود آمدن ابزار های استقرار پیوسته برنامه نویسان کد های جدید خود را همزمان با توسعه، تست هم می‌کنند. یکی از روش های تست نرم افزار، این است که محصول را به صورت یک کامپوننت یا یک پکیج و بدون وارد شدن به جزئیات پیاده سازی آن ببینم و با روش end to end با تولید تست کیس های اتوماتیک، آن را تست کنیم. در این پروژه قصد داریم روشی آزمایشی و جدید را برای این روش تست به کار ببریم، به این صورت که ماژول و کامپوننت تحت تست (پیاده سازی شده با جاوا) را با زبانی تابعی مانند هسکل، پیاده سازی کرده و با تولید تست کیس های اتوماتیک و سنجیدن پوشش های تعریف شده ی گراف، بررسی کنیم که آیا این روش می تواند روش جدیدی برای تست درستی برنامه ها باشد یا خیر. درواقع مسئله ی مطرح شده در این پروژه این است که آیا میتوان بین دو زبان تابعی[2] و امری[3] با تولید تست ها به روش black box به پوشش مطلوبی رسید یا خیر و بهترین معیار برای این پوشش چه معیاریست، به صورتی که با تولید تست کیس ها و رسیدن به پوشش مطلوب در کد زبان تابعی، بتوان گفت به پوشش مطلوبی بر روی زبان امری هم دست خ. در صورت مثبت بودن نتیجه و با به کار بردن این روش و با اتوماتیک شدن فرآیند تولید تست ها برای ماژول ها میتوان سربار تست نویسی را که امروزه بخش زیادی از زمان برنامه نویسان را میگیرد کاهش داد و به اتوماتیک شدن هر چه بیشتر این فرآیند کمک کرد.

[1] Abstract

[2] Declarative language

[3] Imperative language

کلمات کلیدی:

Criteria code coverage – Automatic testing – Black–box testing – language testing – Imperative language testing – Haskell code coverage

فهرست مطالب

فهرست شکل‌ها

فهرست جدول‌ها

فصل 1:

مقدمه و بیان مسئله

در این فصل نخست به بیان مقدمات کار، تاریخچه‌ای کوتاه از مساله تحقیق و روش کلی تحقیق پرداخته، سپس مساله و موضوع مورد بررسی در این پایان‌نامه و اهداف و آرمان‌های کلی تحقیق را بیان می‌کنید و در نهایت به ساختار پایان‌نامه‌ی پیش رو اشاره خواهید کرد.

1-1- مقدمه

با پیشرفت صنایع کامپیوتری و توسعه و همه گیری متدولوژی های توسعه ی agile نیاز به شتاب در تولید نرم افزار در هر بیزینسی که در آن رقابت وجود دارد حس می‌شود. از طرفی به وجود آمدن فرآیند های جدید CI/CD[1] این امکان را به صنعت کامپیوتر آورده است که بتوان محصول را به طور مداوم و به سرعت تغییر داده و حتی دقایقی پس از اتمام پیاده سازی کد، محصول جدید در اختیار مشتری قرار داد، به طوری که حتی deploy strategy هایی به همین منظور طراحی شده اند. نکته ای که در کنار این سرعت نیاز آن به شدت حس میشود، اطمینان از درست بودن source code جدید که قصد داریم artifact آن را deploy کنیم است. در همین راستا ابزارهای CI/CD این امکان را در اختیار قرار داده اند که pipeline ای از تست های دلخواه را بر روی آنها تعریف کرده که پیشنیاز deploy باشند و چنانچه این تست ها بر روی کد جدید با شکست مواجه شد، از deploy محصول جدید که با خطا همراه است جلوگیری شود. به علاوه هدف دیگری نیز از این تست وجود دارد، اینکه تست مداوم بر روی محصول باعث پیدا شدن سریع خطا ها و عدم وجود خطا برای مدت بسیاری در سیستم شود و به علاوه خود فرآیند testing هم به صورت اتوماتیک انجام شود که این موضوع نیز اهمیت بسیاری دارد. هر چه بتوان این فرآیند پر هزینه را به سمت اتوماتیک شدن پیش برد، بیشتر این اصل که فرآیند های تکراری را اتوماتیک کنیم، رعایت کرده ایم.

در راستای اتوماتیک شدن هر چه بیشتر فرآیند تست، میتوان پیاده سازی دیگری از محصول تحت توسعه به زبانی دیگر داشت و با اجرای test generator ای که هدف آن تولید مجموعه تست‌کیس برای این برنامه هاست، هر دو را اجرا کرد. در واقع در این روش هدف این است که با تولید مجموعه ای از تست ها و بدون وارد شدن به جزئیات پیاده سازی هر یک و کاملا به روش black box تست هایی را به صورت اتوماتیک تولید و به هر دو پیاده سازی را با آن اجرا کرد.

[1] Continuous integration continuous delivery/continuous deploy

در این روش چنانچه برای پیاده سازی محصول موازی (محصولی که محصول اصلی نیست و برای تست محصول اصلی تولید شده) از یک زبان سطح بالا و تابعی مانند haskell استفاده کنیم، میخواهیم که با قرار دادن تمرکز بر روی گرفتن پوشش مناسب در کد پیاده سازی شده ی این زبان فرآیند تست را انجام دهیم. اما نکته ای که حائز اهمیت است این است که کدام معیار های پوشش را باید ملاک قرار دهیم که پوشش مناسب بر روی زبان تابعی، منجر به گرفتن پوشش مناسب بر روی زبان امری گردد.

2-1- تاریخچه‌ای از موضوع تحقیق

همانگونه که ذکر شد، در سالیان اخیر، تلاش برای توسعه ی روش های CI/CD و به دنبال آن افزودن تست اتوماتیک به این فرآیند باعث شده است که تحقیقات و تجربیات فراوانی در زمینه ی automatic test به وجود بیاید. به علاوه معیار های پوشش کد در سالیان اخیر موضوع تحقیق بسیاری بوده و به علاوه در صنعت نیز ابزارهای پرکاربردی برای این موضوع ایجاد شده اند. در این بین پرداختن به این مسئله که در این فرآیند مدل اصلی را با استفاده از یک مدل توصیفی دیگر به زبان تابعی بسنجیم و نیز بین این دو زبان زبان معیار پوشش مشترکی بیابیم که با گرفتن پوشش کامل بر یکی از آنها بر روی دیگری نیز پوشش مناسبی بگیریم، کمتر صورت گرفته است که ما در این پروژه قصد داریم این موضوع را بررسی کنیم.

1-3- شرح مسئله تحقیق

همانطور که در قسمت مقدمه ذکر شد، در این پروژه به دنبال این هستیم که معیارهای پوشش کدی را انتخاب کنیم که با تولید یک مجموعه تست و گرفتن پوشش مناسب بر روی کد پیاده سازی شده به زبان تابعی، بتواند پوشش مناسبی بر روی کد پیاده سازی شده به زبان امری را نیز تولید کند. در این پروژه قصد داریم که برای زبان تابعی از زبان هسکل که زبانی pure functional محسوب می‌شود، و برای زبان امری هم از زبان جاوا استفاده کنیم.

نکته ای که در این پروژه چالش برانگیز محسوب می‌شود وجود ساختار های متفاوت در زبان های تابعی و امری است.به نحوی که حتی تفکر برنامه نویس‌به هنگام کد زدن در هر یک از این دو زبان متفاوت است. به علاوه تفاوت ساختار های این دو زبان و higher order بودن زبان هسکل باعث میشود که نتوان mapping مناسبی برای تبدیل این زبان تابعی به یک زبان امری به دست آورد. در نتیجه ی عدم وجود چنین mapping ای، نمی‌توان دانست که برای تست یک کد هسکل حتی‌به صورت white box ، چه قسمت هایی رابلید در تولید test case ها رعلیت کرد که پوشش کاملی بر روی زبان امری گرفته شود.

1-4- روش انجام تحقیق

در این پروژه به کمک ابزار های موجود برای پوشش کد های جاوا و هسکل، در سه فاز به ترتیب به پیاده سازی مسائل کوچک و ساده، مسائل الگوریتمی موجود در مسابقات آنلاین برنامه نویسی و در نهایت پیاده سازی پروژه هایی کوچک رفته ایم. برای به دست آوردن پوشش جاوا از jacoco و برای به دست آوردن پوشش بر روی زبان هسکل از hpc[2] استفاده شده است. در هر مرحله پس

[2] Haskell program coverage

از پیاده سازی برنامه ها میزان پوشش معیار های تعریف شده ی خود را به دست آورده و در نهایت با بررسی آنها، معیار مناسب را که هدف پروژه است انتخاب میکنیم.

5-1- ساختار پایان‌نامه

در فصل دوم، ابتدا به معرفی و مقایسه ی زبان هسکل با جاوا پرداخته و تفاوت های کلیدی زبان های تابعی با زبان های امری را به منظور شناخت بهتر آن بیان میکنیم. در ادامه معیار های پوشش بر روی زبان را معرفی میکنیم و با ابزار های به کار رفته در پروژه برای گرفتن code coverage آشنا شده و بیان می‌کنیم که هر یک این معیار ها را چگونه در خود جای داده اند.

در فصل سوم سه فاز انجام شده در پروژه را پیش برده که در هر یک روش به کار رفته از جمله طراحی، انتخاب مسائل و ... را به همراه چند مثال پیاده سازی شده بررسی کرده و نتایج به دست آمده در هر قسمت را عنوان می‌کنیم. به علاوه در انتهای هر بخش، جداولی را که برای هر معیار در پیاده سازی های مسائل آن فاز به دست آمده است، آورده ایم.

در فصل چهارم با استفاده از نتایج و تحلیل هایی که در فاز قبل صورت گرفت، معیار ها را بررسی کرده و با به دست آوردن correlation موجود بین معیار ها، معیارهایی که ما را به خواسته ی مطرح شده می‌رسانند مشخص میکنیم.

و در نهایت فصل آخر شامل جمع بندی مطالب و مشکلات و پیشنهادهایی که در طول پروژه به آنها برخوردیم خواهد بود.در نهایت، در فصل پنجم، نتیجه‌گیری‌های کلی حاصل شده در این تحقیق، پیاده سازی‌ها و نوآوری‌های انجام شده و محدودیت‌ها مورد بحث قرار می‌گیرد و پیشنهادهایی برای ادامه‌ی مسیر به علاقمندان این حوزه‌ی ارائه خواهد شد.

فصل 2: مفاهیم اولیه و پیش زمینه ها

در فصل پیش رو مقدمات، مفاهیم اولیه و پیش‌زمینه‌هایی را که جهت درک هرچه بهتر موضوع های مطرح شده در این پایان‌نامه مورد نیاز است، از مفاهیم مربوط به زبان های تابعی تا تفاوت های بنیادین آن‌ها در مقایسه با زبان های امری و معرفی معیار های پوشش ارائه خواهد شد

2-1- بررسی یک الگوریتم ساده

در ابتدا برای آشنایی با فضای کد هسکل و پیش از بیان تفاوت های موجود در این دو دنیای متفاوت قصد داریم پیاده سازی مرتب سازی سریع[1] را در دو زبان هسکل و جاوا بررسی و مقایسه کنیم و در همین ابتدای امر با تفاوتهایی که بین دو زبان به طور طبیعی وجود دارند و باعث میشوند که نقاطی در پیاده سازی به وجود آیند که برای تست آنها باید حالات خاصی را بررسی کنیم مشخص شوند.

در شکل *2-1* پیاده سازی جاوا را مشاهده میکنیم که همانطور که از الگوریتم مرتب سازی سریع میدانیم، شامل دو قسمت اصلی partition و تابع بازگشتی sort است. بدیهی است که برای پیاده سازی آن، می بایست به شرط خاتمه ی تابع بازگشتی، درستی عملکرد تابع partition، خروجی صحیح مقدار *pivot*، جا به جایی نهایی pivot و ... می‌بایست کاملا مورد توجه قرار گیرد تا بتوان الگوریتم را به درستی پیاده سازی کرد.

[1] Quick sort

```java
public static void swap(int[] numbers, int firstIndex, int secondIndex) {
    int number = numbers[firstIndex];
    numbers[firstIndex] = numbers[secondIndex];
    numbers[secondIndex] = number;
}

public static int partition(int[] numbers, int beginIndex, int endIndex) {
    int pivotNumber = numbers[endIndex];
    int lastBiggestIndex = beginIndex - 1;
    for (int i = beginIndex; i < endIndex; i++) {
        if (numbers[i] < pivotNumber) {
            lastBiggestIndex++;
            swap(numbers, lastBiggestIndex, i);
        }
    }
    swap(numbers, ++lastBiggestIndex, endIndex);
    return lastBiggestIndex;
}

public static void sort(int[] numbers, int begin, int end) {
    if (end <= begin)
        return;
    int pivot = partition(numbers, begin, end);
    sort(numbers, begin, pivot - 1);
    sort(numbers, pivot + 1, end);
}
```

شکل (2-1) پیاده سازی مرتب سازی به زبان جاوا

در شکل ۲-۲ پیاده سازی همین الگوریتم را به زبان هسکل می‌بینیم:

```haskell
quickSort :: (Ord a) => [a] -> [a]
quickSort [] = []
quickSort (x:xs) = leftSortedArray ++ [x] ++ rightSortedArray
    where
        leftSortedArray = quickSort $ filter (<x) xs
        rightSortedArray = quickSort $ filter (>=x) xs
```

شکل (2-2) پیاده سازی مرتب سازی سریع به زبان هسکل

سینتکس آن برخلاف جاوا ناآشناست و به همین دلیل کمی نیاز به توضیح دارد.

در خط اول signature ـتابع آورده شـــده:ـنام ـتابع quickSort اســـت و آرگومان های آن از ـتلیپ a خواهند بود که می بایســت از نوع Ord باشــد(یعنی قابل مقایســه باشــد)، ورودی تابع یک آرایه از مقادیری با تایپ a اســت و خروجی آن هم آرایه ای از مقادیر با تایپ a. (درواقع a میتواند هر تایپی باشد که قابلیت مقایسه را داشته باشد، از جمله اعداد، رشته ها و یا تایپ های تعریف شده توسط برنامه نویس).

خط دوم و خط سـوم بدنه ی تابع را پیاده سـازی میکنند و معرف یکی ویژگی های زبان هسـکل به نام pattern matching اند که با توجه به ورودی تابع یکی از آنها انتخاب شده و بدنه ی تابع بر اساس خروجی آن تعیین میشـود. به عنوان مثال خط دوم بیان میکند که خروجی تابع quickSort به ازای ورودی آرایه ی خالی، یک آرایه ی خالی خواهد بود. خط سـوم هم حالتی که آرایه خالی نیسـت را پوشـش میدهد، مقدار (x:xs) ورودی تابع را با در نظر گرفتن مقدار ایندکس نخسـت در x و بقیه ی آرایه جز ایندکس اول در xs پیاده سازی میکند. بقیه ی بدنه با توجه به نام آنها و دانستن الگوریتم واضـح تر اسـت و بیان میکند که leftSortedArray برابر آرایه ی مرتب شـــده از مقادیر کوچکتر یا مساوی x و rightSortedArray هم برابر آرایه ی مرتب شده ی مقادیر بزرگتر از x است که در نهایت خروجی برابر خواهد بود.با توالی leftSortedArray، خود مقدار x و rightSortedArray در واقع در اینجا مقدار x همان pivot است.

با وجود سینتکس ناآشنا برای کسانی که تاکنون با زبان هسکل آشنایی ندارند هم مشخص است که تا چه میزان این پیاده سـازی عینا الگوریتم را توضـیح میدهد، در واقع میتوان گفت که میتوان این پیاده سـازی را به نوعی pseudocode هم تلقی کرد، و چنان چه در نظر بگیریم که pseudocode خود حالت abstract الگوریتم اسـت، در واقع این پیاده سـازی خود الگوریتم بدون نوشـتن هیچ کد اضافه ایست. به معنای دیگر میتوان گفت که در این پیاده سازی به تنها چیزی که می بایست فکر کرد خود الگوریتم اســـت. ابزار های زبان ـتابعی هســـکل در بســـیاری از موارد این امکان را ایجاد خواهد کرد که به همین میزان نزدیک به pseudocode الگوریتم و در پروژه های بزرگ تر آن چیزی را پیاده سازی کنیم که نیازمندی خواسته شده است.

2-2- تفاوت های دو فلسفه ی برنامه نویسی

حال که یک مثال از این زبان را مشاهده کردیم قصد داریم تفاوت های اساسی این دو نوع زبان زا که باعث میشوند پیاده سازی ها از هم فاصله بگیرند، ذکر کنیم. همانطور که میدانیم در زبان های تابعی متغیر ها immutable اند، یا به تعبیری متغیر ها فقط declare میشوند و مجددا مقدار دیگری به آنها assign نمیشود، از تبعات مهم این موضوع این است که تابع ها هیچ مقداری را در خود تغییر نمی دهند و تنها اثر آنها مقدار خروجی ایست که برمیگردانند(به زعم بسیاری خواندن کدی که اثر تابع های اینچنین باشد و مقدار ورودی هایش را با رفرنس عوض نکند بسیار آسان تر است) خوب یا بد این امر موجب میشود که هر متغیر که در یک زبان تابعی تعریف شده همیشه یک مقدار را در خود ذخیره کند. این تفاوت مهمی است که باعث میشود که در این زبان ها توابع نقش مهمی پیدا کنند و زمینه ی فکری پیاده سازی کد باشند، چنانچه در زبان های امری این آبجکت ها و متغیر ها هستند که مقادیر را در خود ذخیره کرده و نقش اصلی را در پیاده سازی و اجرا بر عهده دارند.

فارغ از این تفاوت بنیادین، در زبان های تابعی به دلیل استفاده از توابع، اهمیت بیشتری بر روی توابع و استفاده ی راحت تر از آنها در نظر گرفته شده و توابع higher order ای که قابلیت استفاده ی گسترده ای را به برنامه نویس میدهند در آنها تدارک دیده شده. همین توابع و ابزار ها باعث شده اند که همانطور که در مثال بالا دیدیم، بتوان به حلت انتزاعی تر و ساده تری پیاده سازی یک ماژول را نسبت به یک زبان امری انجام داد.

3-2- معرفی معیار های پوشش

با این توضیحات فرض کنید که میخواهیم به دست آوردن کاوریج خوبی از کد زبان امری (جاوا) با تست هایی که کاوریج خوبی بر پیاده سازی زبان تابعی (هسکل) داشته اند را بررسی کنیم. شاید در ابتدا به نظر برسد که متخصر بودن پیاده‌سازی در هسکل نکته ی مثبتی در رسیدن به این هدف است، اما دقیقا عکس این موضوع صادق است، در واقع هنگامی که پیاده سازی پیاده سازی به خود الگوریتم نزدیک باشد، باعث میشود که با بررسی چند تست ساده و گاها با یک تست‌کیس هم بر روی آن به پوشش کاملی رسید، ولی از طرف دیگر هنوز بدون دانستن پیاده سازی زبان امری نمی توان تست‌کیس هایی برای پوشش کامل آن طراحی کرد. به بیانی دیگر پیاده سازی زبان تابعی کمک خاصی برای افزایش پوشش پیاده سازی امری نکرده است و کار ما با وقتی که می دانستیم که باید برای یک تابع مرتب سازی تست کیس طراحی کنیم تفاوتی ندارد و پیاده سازی تابعی تنها به عنوان یک مقایسه‌گر برای مقایسه ی خروجی ها می‌تواند مورد استفاده قرار بگیرد.

در قسمت قبل یک الگوریتم معروف با پیاده سازی هر دو زبان مشاهده شد و کمی در مورد تفاوت های بنیادی هر دو زبان صحبت شد، در این قسمت قصد داریم پیش از آغاز مراحل بعد، در مورد معیارهایی که میتوانند به ما کمک کنند و در ادامه در هر مورد بررسی و تحلیل می شوند توضیحاتی ارائه دهیم. همانطور که میدانیم معیارهای بسیاری برای پوشش کد وجود دارد که برخی از آنها در ابزار های تست و پوشش کد به خوبی پیاده سازی شده اند. در ادامه پرکاربردترین این معیار ها که در این پروژه به بررسی آنها پرداخته ایم به همراه توضیح آن آورده شده است:

i. **معیار پوشش تابعی[۲]: این معیار بیانگر به کار رفتن تابع ها در تست کیس هاست.**
ii. **معیار پوشش دستوری[۳]: بیانگر به کار رفتن statement ها و expression ها در اجرای برنامه است.**

[2] Function coverage

[3] Statement coverage

iii. **معیار پوشش شاخه ای[4]: بیانگر این است که آیا ساختار های کنترلی برنامه هر دو حالت خود را شامل شده باشند (به عنوان مثال یک** if .. else **ساده)**

iv. **معیار پوشش مسیر[5]: همه ی مسیرهای کنترلی و مقادیر شرطی موجود در یک کد، پوشش داده میشود.**

بین معیارهای پوشش این معیار ها به خوبی در اکثر ابزار های پوشش کد پیاده سازی شده و به علاوه برای بررسی پوشش در پروژه های صنعتی وب که غیر حیاتی اند کافی به نظر میرسند (در واقع معیار های کامل تری نیز وجود دارند که در پروژه های خاص تر و بسته به کیس پروژه مناسبت ترند، مثلا معیار modified condition coverage **در برنامه های حساس تر امنیتی و جانی لزومشان بیشتر حس میشود).**

4-2- معرفی ابزارها

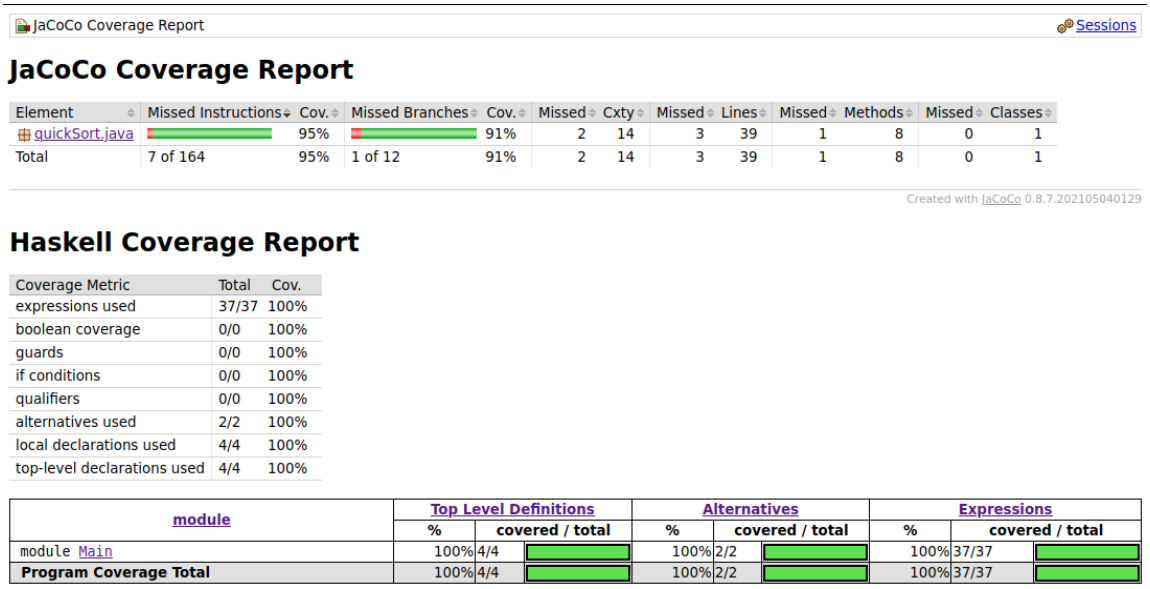
برای به دست آوردن این معیار ها از دو ابزار hpc **برای زبان هسکل و** jacoco **برای زبان جاوا استفاده شده است. این دو ابزار با در اختیار قرار دادن** cli[6] **مناسب و ارائه ی گزارش در قالب** html **به صورت آماری و حتی ارائه ی خروجی ترکیبی کد به همراه پوشش آن، ابزار مناسبی برای پروژه ی ما تلقی میشوند. در ادامه با بررسی مثالی مشخص میکنیم که هر یک از این دو ابزار در گزارش خود، معیار های ما را چگونه در خود جای داده اند.**

[4] Branch coverage

[5] Path coverage

[6] Command-line interface

بنابراین گزارش مثال نخست یعنی تابع quick sort خود را که در هر دو زبان پیاده سازی کرده ایم در شکل 2-3 آورده شده است. در اینجا هدف صرفا معرفی خروجی گزارش و تطابق پارامتر های آن با معیار های معرفی شده است.



JaCoCo Coverage Report — Sessions

JaCoCo Coverage Report

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
quickSort.java		95%		91%	2	14	3	39	1	8	0	1
Total	7 of 164	95%	1 of 12	91%	2	14	3	39	1	8	0	1

Created with JaCoCo 0.8.7.202105040129

Haskell Coverage Report

Coverage Metric	Total	Cov.
expressions used	37/37	100%
boolean coverage	0/0	100%
guards	0/0	100%
if conditions	0/0	100%
qualifiers	0/0	100%
alternatives used	2/2	100%
local declarations used	4/4	100%
top-level declarations used	4/4	100%

module	Top Level Definitions		Alternatives		Expressions	
	%	covered / total	%	covered / total	%	covered / total
module Main	100%	4/4	100%	2/2	100%	37/37
Program Coverage Total	100%	4/4	100%	2/2	100%	37/37

شکل (2-3) نمونه ی گزارش معیارهای پوشش (مثال مرتب سازی سریع)

i. معیار پوشش تابعی: در خروجی jacoco این معیار با استفاده از دیتای موجود در ستون Methods و مقدار Missed مربوط به آن مشخص میشود(بقیه ی معیار های گزارش jacoco به همین ترتیب اند) بنابراین می توان گفت که در ماژول مورد نظر هشت تابع یافت شده که تنها یکی از آن ها فراخوانی نشده است. در خروجی HPC لما این معیاربه طور جدا نیامده ولی شامل فیلد top-level declaration used میشود. این معیار همچنین شامل توابعی است که record type های زبان هسکل با deriving یک تایپ میتوانند به کار ببرند.

ii. معیار پوشش دستوری: این معیار در خروجی jacoco با ستون missed instructions و cov مربوط به آن و در خروجی HPC با expression used مشخص شده (jacoco بر اساس java

bytecode کـار میکنـد و هر instruction را معـادل یـک دســتور java bytecode آن در نظر میگیرد)

iii. معیار پوشـــش شـــاخه ای: این معیار در خروجی jacoco با ســـتون missed branches و cov مربوط به آن و در خروجی HPC با مقدار if condition مشخص میشود.

iv. معیار پوشـش مسـیر: این معیار در خروجی jacoco با سـتون Cxty[7] و مقدار missed آن و در خروجی HPC میتوان آن را مجموع مقدار guard و alternative used به حساب آورد.

به علاوه ی خروجی این معیار ها، این دو ابزار خروجی ای رابا نمایش source code در اختیار قرار میدهند که میتوان پوشش هر خط کد را در آن مشاهده کرد. شکل 2-4 و 2-5 به ترتیب خروجی پوشـش کد جاوا و هسـکل می‌باشـند. این دو خروجی همزمان با گرفتن خروجی html این دو ابزار تولید میشوند.

```
26.     public static void swap(int[] numbers, int firstIndex, int secondIndex) {
27.         int number = numbers[firstIndex];
28.         numbers[firstIndex] = numbers[secondIndex];
29.         numbers[secondIndex] = number;
30.     }
31.
32.     public static int partition(int[] numbers, int beginIndex, int endIndex) {
33.         int pivotNumber = numbers[endIndex];
34.         int lastBiggestIndex = beginIndex - 1;
35.         for (int i = beginIndex; i < endIndex; i++) {
36.             if (numbers[i] < pivotNumber) {
37.                 lastBiggestIndex++;
38.                 swap(numbers, lastBiggestIndex, i);
39.             }
40.         }
41.         swap(numbers, ++lastBiggestIndex, endIndex);
42.         return lastBiggestIndex;
43.     }
44.
45.     public static void sort(int[] numbers, int begin, int end) {
46.         if (end <= begin)
47.             return;
48.         int pivot = partition(numbers, begin, end);
49.         sort(numbers, begin, pivot - 1);
50.         sort(numbers, pivot + 1, end);
51.     }
52.
```

[7] Cyclomatic coverage

شکل (2–4) گزارش jacoco بر روی source code (مثال مرتب سازی سریع)

```
1  quickSort :: (Ord a) => [a] -> [a]
2  quickSort [] = []
3  quickSort (x:xs) = leftSortedArray ++ [x] ++ rightSortedArray
4          where
5                  leftSortedArray = quickSort $ filter (<x) xs
6                  rightSortedArray = quickSort $ filter (>=x) xs
7
8  toInt :: String -> Int
9  toInt x = read x :: Int
10
11 parseInputs :: [String] -> [Int]
12 parseInputs inputs = map toInt inputs
13
14 main = do
15   numberCounts <- getLine
16   args <- getLine
17   let inputs = parseInputs  (words args)
18   let sortedNumbers =  quickSort inputs
19   putStrLn ("sorted:\n" ++ (show sortedNumbers :: String))
```

شکل (2–5) گزارش hpc بر روی source code (مثال مرتب سازی سریع)

فصل 3: پیاده‌سازی مسائل[1]

پس از معرفی معیار های پوشش مورد بررسی و آشنایی مختصر با toolkit **های به کار رفته در پروژه، قصد داریم هر یک از مراحل طی شده برای دستیابی به معیار پوشش را مرحله به مرحله شرح داده و نتایج به دست آمده در هر یک را تحلیل کنیم. مسائل و کد هایی که معیار ها را بر اساس آن های ارزیابی کرده ایم را به سه فاز تقسیم کرده ایم.**

در فاز نخست به پیاده سازی الگوریتم های ساده پرداخته ایم، در این قسمت دو خانواده از الگوریتم ها، الگوریتم های جستجو و الگوریتم های مرتب سازی پیاده سازی شده اند. در الگوریتم های جستجو دو الگوریتم جست و جوی خطی[2] و جستجوی باینری[3]، و در بین الگوریتم های مرتب‌سازی، الگوریتم های مرتب سازی سریع، مرتب سازی ادغامی[4]، مرتب سازی حبابی[5]، مرتب سازی درجی[6] و مرتب سازی انتخابی[7] پیاده سازی شده اند. به علاوه یک مسئله ی خاص مرتب سازی، که مرتب سازی شبکه ای نام دارد هم در این قسمت بررسی شده است.

در مرحله ی بعد و در بخش دوم به سراغ مسائل پیچیده تر رفته و چهار مسئله از contest

---

[1] Git repository: https://github.com/youngPieros/analysis-of-criteria-code-coverages-for-functional-and-imperative-language

[2] Linear search

[3] Binary search

[4] Merge sort

[5] Bubbke sort

[6] Insertion sort

[7] Selection sort

های آنلاین انتخاب شـده اند، دلیل این انتخاب علاوه بر بررسـی مسـائل پیچیده تر نیاز به این بود که بررســی معیار ها به یک راه حل در زبان امری خلاصــه نشــود و راه حل های مختلف پیاده سـازی شـده توسـط افراد مختلف مورد بررسـی قرار گیرد، به عبارت دیگر، معیار ها را نه فقط روی یک پیاده سازی بلکه روی چند پیاده سازی متفاوت مورد بررسی قرار دهیم.در این قســمت ســعی شــد که علاوه بر انتخاب کد هایی که در کانتســت ۱۰۰ درصــد تســت کیس ها را پاس کرده بودند،نمونه هایی هم که پاســخ اشــتباه هم تولید میکنند بررسی شده، تا بتوان روی این موضوع هم بررسی لازم صورت گیرد.

در انتها و در فاز آخر دو پروژه در مقیاس کوچک که یکی برنامه ی ســفارش آنلاین غذا و دیگری سامانه ی انتخاب واحد است و هر دو پروژه جزو پروژه های تعریف شده در درس مهندسـی اینترنت دانشـکده ی برق و کامپیوتر دانشـگاه تهران بوده اند انتخاب شـده‌اند، و با پیاده سـازی کد هسـکل آنها و تولید تسـت کیس های اتوماتیک به بررسـی معیار های تعریف شده پروژه بر روی آنها پرداخته ایم.

در هر قسمت نحوه ی پیاده سازی، روش تولید تست‌کیس ها و در نهایت معیار های پوشش کد و تحلیل آنها آورده شده است.

3-1- بررسی چند الگوریتم ساده

در ابتدا مسئله ی جستجوی خطی که هدف آن یافتن درایه ی یک عدد جستجو شده بین آرایه ای از اعداد است مورد بررسی قرار گرفت. نکته ای که باید توجه داشت این است که در این پروژه قصد داریم تا حد امکان ایده ی black box testing را دنبال کنیم و بدون وارد شدن به جزئیات پیاده سازی، تست کیس هایی برای پوشش کد های خود طراحی کنیم. در مسئله ی جستجو به نظر میرسد که بررسی دو حالت بتواند ما را به پوشش مناسبی روی کد، فارغ از اینکه کد چگونه پیاده سازی شده باشد برساند: وجود داشتن و نداشتن مقدار جستجو شده در آرایه.

در این قسمت هدف این است که در زبان هسکل پیاده سازی ای بدون استفاده از ویژگی های سطح بالای آن داشته باشیم، در نتیجه برای پیاده سازی این مسئله از توابع بازگشتی استفاده کرده ایم. بنابراین دو پیاده سازی خواهیم داشت و تست آنها شامل چند نمونه است که هر کدام شامل آرایه ای از اعداد و یک عدد برای جستجو اند.

پیاده سازی زبان هسکل این الگوریتم در شکل 3-1 آمده است.

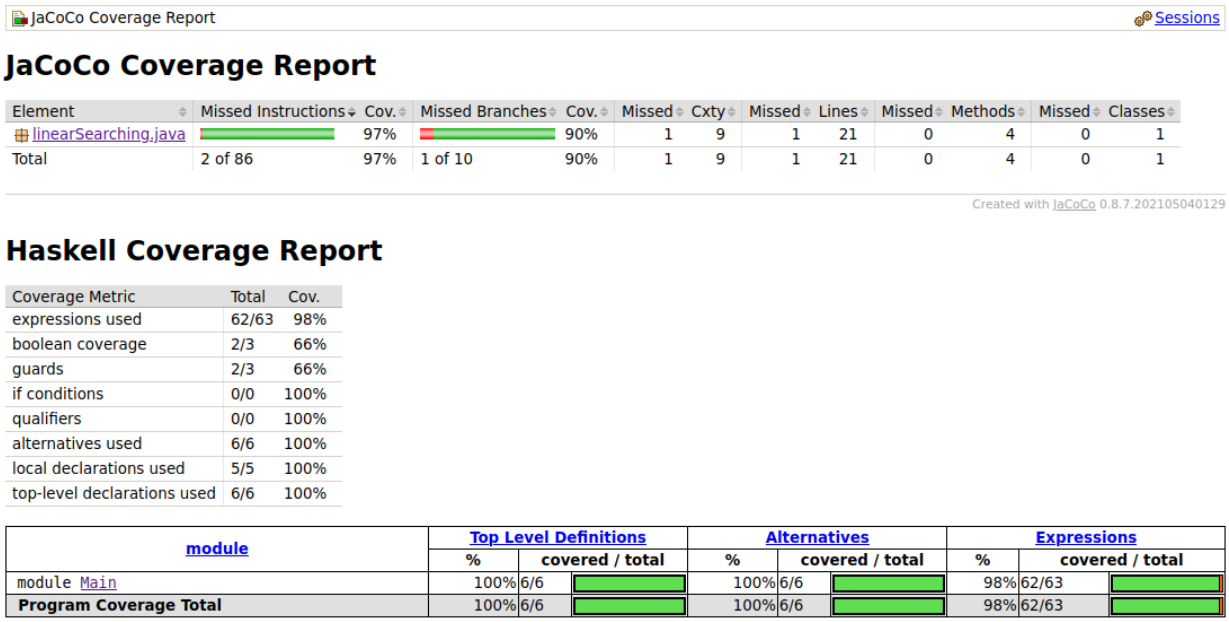
```haskell
linearSearch :: (Eq a) => a -> [a] -> Int
linearSearch _ [] = -1
linearSearch a (x:xs)
    | x == a = 0
    | indexOfTailSearch == -1 = -1
    | otherwise = indexOfTailSearch + 1
    where
        indexOfTailSearch = linearSearch a xs
```

شکل (3-1) پیاده سازی تابع جستجوی خطی با هسکل

در این پیاده سازی به روش بازگشتی، در صورت خالی بودن آرایه مقدار درایه ی منفی یک را بازخواهد گرداند، در حالتی که آرایه خالی نباشد، چنانچه عضو اول آرایه برابر مقدار جستجو شده بود، درایه ی صفر و در غیر این صورت با استفاده از خروجی بازگشتی همین تابع (بسته به اینکه این

مقدار در بقیه ی آرایه یافت شـــده باشـــد یا نه) یا درایه ی منفی یک یا خروجی تابع بازگشـــتی + ۱ را برمیگردانیم.

پیش از ارائه ی راه حل جاوا، نتایج به دســت آمده از معیار های پوشــش در الگوریتم linear search که در شکل 3-2 آمده اند را بررسی مختصری میکنیم:


JaCoCo Coverage Report

Sessions

JaCoCo Coverage Report

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
linearSearching.java		97%		90%	1	9	1	21	0	4	0	1
Total	2 of 86	97%	1 of 10	90%	1	9	1	21	0	4	0	1

Created with JaCoCo 0.8.7.202105040129

Haskell Coverage Report

Coverage Metric	Total	Cov.
expressions used	62/63	98%
boolean coverage	2/3	66%
guards	2/3	66%
if conditions	0/0	100%
qualifiers	0/0	100%
alternatives used	6/6	100%
local declarations used	5/5	100%
top-level declarations used	6/6	100%

module	Top Level Definitions		Alternatives		Expressions	
	%	covered / total	%	covered / total	%	covered / total
module Main	100%	6/6	100%	6/6	98%	62/63
Program Coverage Total	100%	6/6	100%	6/6	98%	62/63

شکل (3-2) معیارهای پوشش بر روی تابع linear search

همانطور که مشـاهده می شـود در جاوا نتوانسـته ایم به پوشـش کاملی دسـت پیدا کنیم، مشـکل با نگاه به پیاده سازی کد جاوا مشخص میشود:

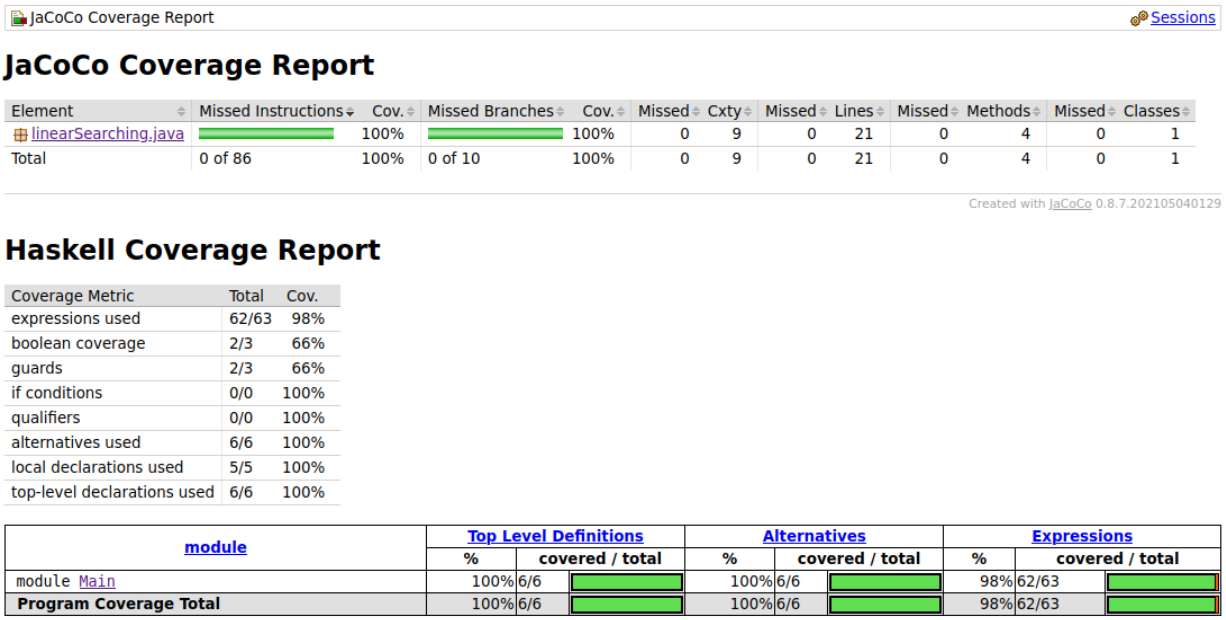
```
    public int findWithLinearSearchAlgorithm(int number, ArrayList<Integer> numbers) {
        if (numbers.size() == 0)
            return -1;
        for (int i = 0; i < numbers.size(); i++)
            if (numbers.get(i) == number)
                return i;
        return -1;
    }
```

شکل (3-3) تابع پیاده سازی شده ی linear search در جاوا

در پیاده سازی جاوا در ابتدای جست و جو خالی بودن آرایه چک شده است، هر چند که دو خط اول این برنامه قابلیت حذف دارند ولی ممکن است در یک پیاده سازی چنین چیزی پیاده شده باشد. همین شرط در پیاده سازی هسکل نیز وجود دارد، اما چون در پیاده سازی کد هسکل تابع به صورت بازگشتی زده شده است، هنگامی که مقدار جستجو شده در آرایه وجود نداشته باشد، پوشش داده میشود. با همین مثال ساده مشخص میشود که برای به دست آوردن تستی که پوشش مناسبی در هر دو پیاده سازی داشته باشد، رعایت نکاتی را در طراحی حالت مختلف تست می طلبد که عدم توجه به آنها در طراحی تست ها ممکن است باعث شود که corner case هایی از پیاده سازی ها بدون پوشش بمانند. در واقع مشابه این اتفاق در پروژه های بزرگ که برای اجرای یک درخواست ساده ی کاربر، ممکن است مسیر کدی چند صد خطی اجرا شود، بسیار محتمل است.

نکته ی مهم دیگر استفاده از recursion در کد زبان تابعی است، با اینکه کد هر دو شامل یک شرط خاص (یعنی بازگشت مقدار منفی یک در حالتی که آرایه خالی است) بوده اند ولی در کد جاوا بدون پوشش و در هسکل پوشش داده شده است، دلیل این امر هم این است که در یک تابع بازگشتی، شرط خاتمه در هربار اجرا چک میشود و در نتیجه نمی‌توان گفت که entry point برای یک تابع بازگشتی برابر با همان entry point در تابع غیر بازگشتی است، چرا که تابع بازگشتی در تکرار های بعدی با ورودی متفاوت دیگری هم فراخوانی میشود. اما عکس این موضوع می تواند صادق باشد. به عبارت دیگر چنانچه در زبان تابعی از روش غیر بازگشتی استفاده کنیم میتوان گفت که در صورت بازگشتی بودن یا نبودن تابع در زبان امری، چنانچه شروطی مشترک در آغاز دو تابع داشته باشیم، میتوان با پوشش کامل زبان تابعی، پوشش مناسبی روی زبان امری هم به دست آورد. در نتیجه میتوان ادعا کرد که چنانچه پوششی مناسب بر روی کدی غیر بازگشتی در زبان هسکل داشته باشیم، لازم نیست نگران بازگشتی بودن پیاده سازی آن در جاوا باشیم.

پس از اضافه کردن حالت خاص به تست کیس (خالی بودن آرایه)، همه ی معیار های پوشش کامل شده اند(تنها expression پوشش داده نشده مربوط به کدی غیر مربوط به تابع الگوریتم است به علاوه در پوشش guards ها در کد هسکل hpc همیشه آخرین guard را در یک دنباله بدون پوشش حساب میکند، چرا که otherwise است و همیشه مقدار true راه به همراه خواهد داشت. در واقع در مثال زیر هر سه guard موجود در کد پوشش داده شده اند):



JaCoCo Coverage Report — Sessions

JaCoCo Coverage Report

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
linearSearching.java		100%		100%	0	9	0	21	0	4	0	1
Total	0 of 86	100%	0 of 10	100%	0	9	0	21	0	4	0	1

Created with JaCoCo 0.8.7.202105040129

Haskell Coverage Report

Coverage Metric	Total	Cov.
expressions used	62/63	98%
boolean coverage	2/3	66%
guards	2/3	66%
if conditions	0/0	100%
qualifiers	0/0	100%
alternatives used	6/6	100%
local declarations used	5/5	100%
top-level declarations used	6/6	100%

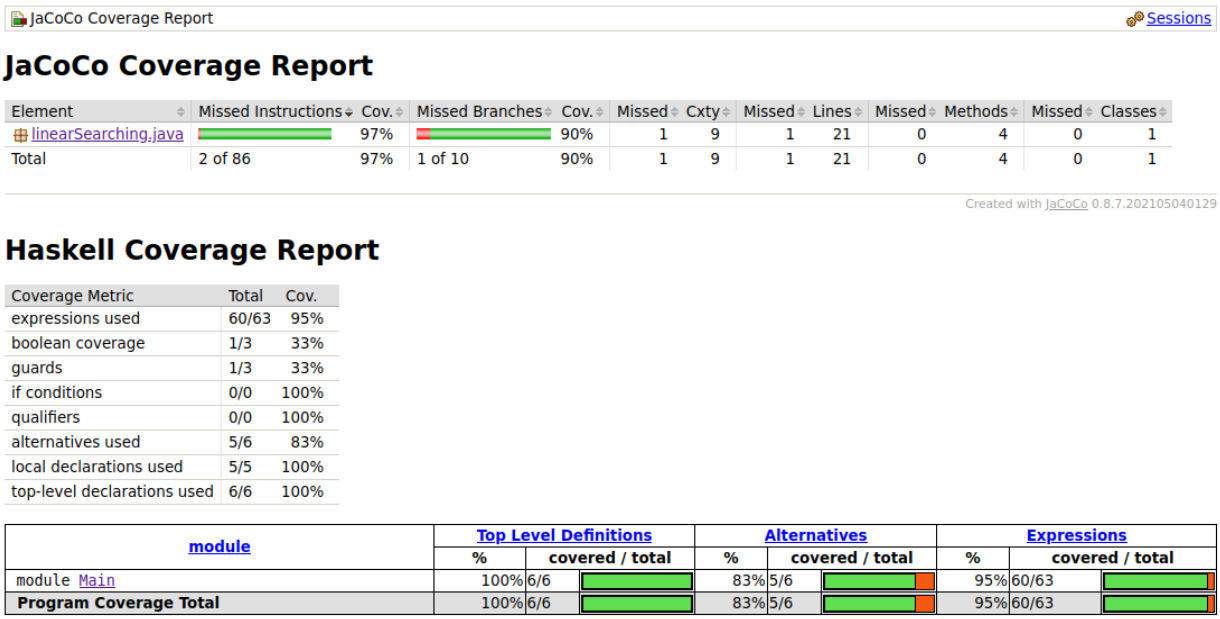
module	Top Level Definitions		Alternatives		Expressions	
	%	covered / total	%	covered / total	%	covered / total
module Main	100%	6/6	100%	6/6	98%	62/63
Program Coverage Total	100%	6/6	100%	6/6	98%	62/63

شکل (3–4) معیارهای پوشش بر روی تابع linear search با افزودن حالت های خاص به تست ها

```haskell
linearSearch :: (Eq a) => a -> [a] -> Int
linearSearch _ [] = -1
linearSearch a (x:xs)
    | x == a = 0
    | indexOfTailSearch == -1 = -1
    | otherwise = indexOfTailSearch + 1
      where
          indexOfTailSearch = linearSearch a xs
```

شکل (3–5) تابع پیاده سازی شده ی linear search در هسکل

حال بررسی یک مورد دیگر هم الزامی است، حذف تست‌کیس هایی که در آنها مقدار مورد نظر وجود ندارد. میخواهیم تغییر معیار ها را در حالتی که بخشی از عملکرد برنامه سنجیده نمی شود بررسی کنیم:



JaCoCo Coverage Report — Sessions

JaCoCo Coverage Report

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
linearSearching.java		97%		90%	1	9	1	21	0	4	0	1
Total	2 of 86	97%	1 of 10	90%	1	9	1	21	0	4	0	1

Created with JaCoCo 0.8.7.202105040129

Haskell Coverage Report

Coverage Metric	Total	Cov.
expressions used	60/63	95%
boolean coverage	1/3	33%
guards	1/3	33%
if conditions	0/0	100%
qualifiers	0/0	100%
alternatives used	5/6	83%
local declarations used	5/5	100%
top-level declarations used	6/6	100%

module	Top Level Definitions		Alternatives		Expressions	
	%	covered / total	%	covered / total	%	covered / total
module Main	100%	6/6	83%	5/6	95%	60/63
Program Coverage Total	100%	6/6	83%	5/6	95%	60/63

شکل (3-6) معیارهای پوشش بر روی تابع linear search با حذف حالت های خاص از تست

به نظر میرسد که با وجود اینکه یکی از دو حالت کلی برنامه حذف شده اند تغییر چندانی روی بسیاری از معیار ها صورت نگرفته است. همه ی توابع مانند گذشته استفاده شده اند، درصد instruction ها و expression های کد های جاوا و هسکل نیز تفاوت چندانی نکرده است. دلیل این موضوع به این برمیگردد که کد های دیگری نیز برای خواندن و تبدیل ورودی ها وجود دارند. که آنها همچنان اجرا می شوند و باعث میشوند بسیاری از معیار ها با وجود حذف کد افت چشمگیری پیدا نکنند. حتی با وجود اینکه این کد، یک کد کوچک است ولی معیار expression در گزارش hpc و معیار مترادف آن در jacoco، یعنی instructions هر کدام تنها دو واحد نسبت به حالت قبل افت داشته اند. به عبارت دیگر با در نظر گرفتن این دو مقیاس، چنانچه فردی ندلند که چه تست کیس هایی برای برنامه ها ایجاد شده است، حدس خواهد زد که تست‌کیس ها مناسب و پوشش به نحو احسن صورت گرفته، در حالی که چنین نیست. از طرف دیگر اما، دو معیار دیگر ما یعنی پوشش شاخه ای و پوشش مسیر هر یک تحت تاثیر قرار گرفته اند و درصد قابل توجهی از پوشش خود را از دست داده اند.

بنابراین تا اینجا این دو نکته در مسائل کوچک به دست آمد: در نظر گرفتن حالات خاص، حتی بدون وارد شدن به کد می‌تواند پوشش مناسبی را روی هر دو کد ایجاد کند، از بین معیار ها در کد

های کوچک معیار پوشش شاخه ای و مسیری از دو معیار دیگر ارتباط بهتری با یکدیگر دارند و با وضوح بیشتری می‌توانند نبود پوشش مناسب را بر روی کد نشان دهند.

در بقیه ی پیاده سازی ها سعی شده است، که روش های دیگر موجود در کد زدن زبان های تابعی و یا ترکیب آنها استفاده شود، روش به کار رفته در هر الگوریتم به شرح زیر است:

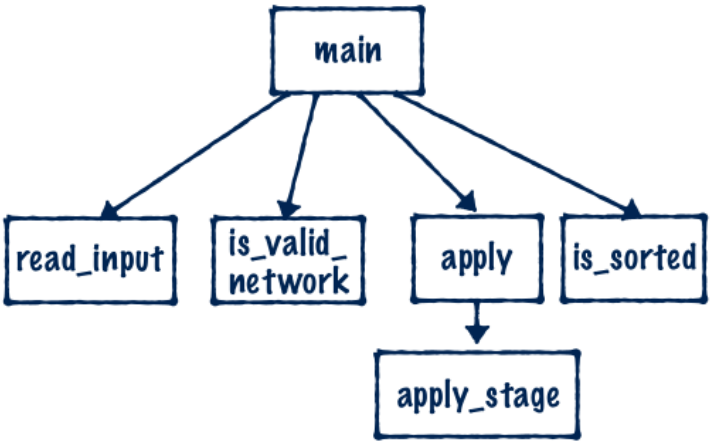
- **جستجوی خطی: پیاده سازی به روش بازگشتی**
- **جستجوی باینری: پیاده سازی به روش بازگشتی**
- **مرتب سازی سریع: پیاده سازی به روش بازگشتی**
- **مرتب سازی ادغامی: پیاده سازی به روش بازگشتی**
- **مرتب سازی انتخابی: پیاده سازی به روش** higher order functions
- **مرتب سازی درجی: ترکیب روش بازگشتی و** higher order functions
- **مرتب سازی حبابی: ترکیب روش بازگشتی و** higher order functions

نتایج کامل این تست ها به همراه در نظر گرفتن حالات خاص آن در جدول۱ بر اساس هر معیار آورده شده است:

معیارها	معیار پوشش تابعی		معیار پوشش عبارت		معیار پوشش شاخه ای		معیار پوشش مسیر	
زبان	هسکل	جاوا	هسکل	جاوا	هسکل	جاوا	هسکل	جاوا
الگوریتم جستجوی خطی	100	100	98	100	100	100	100	100
الگوریتم جستجوی باینری	100	100	98	100	100	100	100	100
الگوریتم مرتب سازی سریع	100	87	100	95	100	91	100	85
الگوریتم مرتب سازی حبابی	100	100	100	97	100	100	100	92
الگوریتم مرتب سازی انتخابی	100	85	100	97	100	100	100	92
الگوریتم مرتب سازی درجی	100	100	100	97	100	100	100	92
الگوریتم مرتب سازی ادغامی	100	100	100	98	100	100	100	93

جدول (3-1) معیار پوشش های الگوریتم های پیاده سازی شده ی ساده در فاز نخست

پیش از آنکه به سراغ فاز بعد برویم قصد داریم الگوریتم مرتب سازی شبکه ای را به عنوان مثال کاملتری از الگوریتم های فاز یک بیشتر بررسی کنیم. تعریف این الگوریتم و یا به تعبیر درست تر روش مرتب سازی شبکه ای در کتاب Introduction to Algorithms[1] آمده است. این الگوریتم شامل سه مرحله است که به ترتیب شامل چک کردن درستی ساختار شبکه، فرآیند مرتب سازی و در نهایت بررسی موفقیت فرآیند در مرتب سازی است.



شکل (3-7) جریان پردازه ی روش مرتب سازی شبکه ای

این مثال، نسبت به سایر الگوریتم های موجود در این فاز پیچیده تر است و میتواند نتایج دقیق تری را در مورد معیار های ما ارائه دهد.

مجددا مانند قبل قصد داریم به صورت black box تست کیس هایی برای اجرای برنامه مشخص کنیم. به نظر میرسد، که با توجه به تعریف مسئله، موارد

- نادرست بودن شبکه ی ورودی
- درست بودن شبکه و عدم مرتب شدن شبکه پس از اجرای فرآیند

[1] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, “Introduction to Algorithms”, 2nd edition, MIT Press, 2001

- درست بودن شبکه و مرتب شدن شبکه پس از اجرای فرآیند
- میتوانند حالت های اصلی تست های ما باشند. به علاوه همانطور که از مرحله ی قبل به آن پی بردیم نیاز داریم که حالاتی خاص را هم به این قسمت ها اضافه کنیم، این حالات خاص با توجه به تعریف مسئله بدین صورت در نظر گرفته شده است:
- شبکه ی از پیش مرتب شده
- طول زوج یا فرد شبکه

در این مسئله حالت خالی بودن شبکه جزو حالت های امکان پذیر نیست و در صورت ورود شبکه ای با ابعاد صفر برنامه خاتمه میابد. در test generator پیاده سازی شده امکان تنظیم تعداد شبکه های معتبر، تعداد شبکه های غیر معتبر، ایجاد تست های شبکه های از پیش مرتب شده و اندازه ی سایز پکت های شبکه وجود دارد. حال میخواهیم ببینیم با تغییر هر یک از این تست ها معیار های ما چه تغییری پیدا میکنند:

معیارها در حالتی که برنامه ها را با ۱۰۰ شبکه معتبر، ۱۰ شبکه ی غیر معتبر و یک شبکه ی از پیش مرتب شده با سایز های ۶ و ۷ به این ترتیب می باشند (نیمی از شبکه ها در نهایت مرتب شده و نیمی هم غیر مرتب):

JaCoCo Coverage Report — Sessions

JaCoCo Coverage Report

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
networkSorting.java		100%		100%	0	29	0	50	0	9	0	1
Total	0 of 266	100%	0 of 40	100%	0	29	0	50	0	9	0	1

Created with JaCoCo 0.8.7.202105040129

Haskell Coverage Report

Coverage Metric	Total	Cov.
expressions used	183/183	100%
boolean coverage	2/3	66%
guards	2/3	66%
if conditions	0/0	100%
qualifiers	0/0	100%
alternatives used	3/3	100%
local declarations used	15/15	100%
top-level declarations used	12/12	100%

module	Top Level Definitions		Alternatives		Expressions	
	%	covered / total	%	covered / total	%	covered / total
module Main	100%	12/12	100%	3/3	100%	183/183
Program Coverage Total	100%	12/12	100%	3/3	100%	183/183

شکل (3-8) معیارهای پوشش بر روی پیاده سازی های مرتب سازی شبکه ای

همانگونه که مشخص است، بر روی کد هسکل توانسته ایم پوشش کاملی بر روی همه ی معیار ها بگیریم. در جاوا هم پوشش کامل است. بنابراین به نظر میرسد که با در نظر گرفتن همه ی حالت ها میتوان به پوشش کاملی بر روی دو برنامه رسید. به کار گیری پیاده سازی با استفاده از higher order functions در کد هسکل باعث شده است که به جز خروجی نهایی تابع process که نتیجه را مشخص میکند هیچ، شاخه ی دیگری نداشته باشیم، با این وجود با در نظر گرفتن همه ی حالت های ورودی و خروجی، رسیدن به مجموعه تست‌کیس هایی که هر دو زبان را به طور کامل پوشش دهند، امکان‌پذیر شده است. این نکته وقتی قابل درک تر میشود که در کد جاوا ۹ حلقه ی for و ۱۱ شاخه ی if .. else وجود دارد. و بدون در نظر گرفتن و وارد شدن به جزئیات این برنامه توانسته ایم به صورت black box تست‌کیس هایی با پوشش مناسب تولید کنیم. در ادامه در جدول ۲ معیار های پوشش برای اجرای حالت های مختلف تولید مجموعه تست‌کیس آورده شده است.

	معیار پوشش تابعی		معیار پوشش عبارت		معیار پوشش شاخه ای		معیار پوشش مسیر	
	هسکل	جاوا	هسکل	جاوا	هسکل	جاوا	هسکل	جاوا
حالت تست کامل	100	100	100	100	100	100	100	100
بدون تست های حالت مرتب شده	100	100	100	100	100	100	100	100
بدون تست های حالت نامرتب	100	100	100	100	100	100	100	100
بدون تست های حالت های شبکه ی نادرست	100	100	99	97	100	95	66	93
بدون تست شبکه های با طول زوج	100	100	100	100	100	100	100	100
بدون تست شبکه های با طول فرد	100	100	96	99	100	95	100	93

جدول (2-3) معیار پوشش های مختلف الگوریتم مرتب سازی شبکه ای در حالت های مختلف تست کیس ها

2-3- پیاده سازی چند مسئله از online contests

گرفتن پوشش کامل لزوما به معنای درست بودن کد نیست. به عنوان مثال ممکن است با

فراموش کردن حالتی خاص در کد و یابا وجود انواع خطا های منطقی و معنایی،باز هم پوشش کاملی بر روی کد خود بگیریم در حالی که هدف کد که در نهایت تولید دیتا یا پردازش صحیح است، محقق نشده. وجود ورودی و خروجی استاندارد به ما امکان می دهد برنامه ی جاوا و هسکل را با تولید یک تست کیس اجرا کرده و به راحتی خروجی آن ها را نیز با یکدیگر مقایسه کنیم. به علاوه نیاز داریم که پیاده سازی زبان تابعی را نه تنها با یک پیاده سازی از زبان امری، بلکه با چندین پیاده سازی مقایسه کنیم. همه ی این موضوعات باعث شدند که در فاز دوم پروژه مسائلی را از سایت های hackerrank و codeforces انتخاب کنیم، پیاده سازی هسکل و جاوای آن ها را انجام دادهبا انتخاب چند پیاده سازی جاوای موجود که توسط افراد مختلف به عنوان پاسخ این مسائل در این سایت ها submit شده اند، به بررسی معیار های مطرح شده بپردازیم. هدف از انجام این کار این است که معیار های پوشش را در برنامه ی هسکل و سبک های متفاوت کدنویسی در جاوا توسط افراد مختلف بسنجیم.به علاوه در این قسمت تصمیم گرفتیم که در انتخاب کد های جاوابه کدهایی که نمره ی کاملی از online judge گرفته اند اکتفا نکرده و کد هایی را که دارای باگ بوده اند و نمره ی کامل نگرفته اند هم در آزمون خود شرکت دهیم.

در ادامه یکی از این مسائل آورده شده که مسئله ای از نوع scheduling[2] است. شرح مختصر آن بدین صورت است که تعدادی تسک داریم که هر یک ددلاین و زمانی برای انجام تسک دارند که باید آنها را با شروع از ثانیه ی صفر انجام داد، میتوان در حین انجام هر یک از تسک ها به دیگری سوییچ کرد و هدف این است که بیشترین زمانی که از ددلاین یک تسک بدون انجام آن می گذرد را مینیمایز کنیم. ورودی این مسئله تعداد تسک ها و ددلاین و زمان لازم برای هر تسک است، تسک ها ترتیب خاصی دارند و می‌بایست به ازای تسک های شماره ی یک تا هر تسک هم این مقدار مینیمایز را حساب کنیم. این مسئله به این دلیل انتخاب شد که یک مسئله ی الگوریتمی پرکاربرد در حوزه ی کامپیوتر است و به علاوه مسئله ایست که در برای حل آن باید پیاده‌سازی ای مرکب از higher order functions و recursion را به کار گیریم.

[2] https://www.hackerrank.com/challenges/task-scheduling/problem

پس از پیاده سازی کد هسکل و کد جاوای این زبان و انتخاب چند پاسخ دیگر از پاسخ های ارسالی کاربران برای این مسئله، test generator ای هم برای تولید تست‌کیس های رندوم آن طراحی شد. برای طراحی تست های این مسئله شرط خاصی در نظر گرفته نشد و به تولید داده هایی رندوم اکتفا شد. در واقع میتوان با تولید تعداد کافی عدد و تست رندوم همه ی حالت هایی که حالت های خاص این مسئله اند را به دست آورد، از طرفی چون خروجی های مسئله در کلاس های متفاوتی قرارندارند و خروجی تنها آرلیه ای از اعداد است، بنابراین می توان از حللت بندی خروجی ها و ورودی های مسئله چشم پوشید (میتوان گفت که شاید تنها حالت خاصی که باید مورد توجه قرار دهیم صفر بودن پاسخ در آغاز تسک هاست که از آن چشم پوشی میکنیم) و در این مسئله تنها به این نکته توجه داشت که هر یک از معیار های مختلف در این مسئله به چه نحوی پوشش داده شده اند.

خروجی این پوشش با اجرای تست بر روی تسک هایی که ددلاین آنها به صورت رندوم بین ۱ تا ۵۰ زمان مورد نیاز آنها بین ۱ تا ۱۰ است، تولید شده است، همچنین تعداد تسک ها را ده تسک در نظر گرفته ایم:

JaCoCo Coverage Report — Sessions

JaCoCo Coverage Report

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
BatchScheduling.java		92%		95%	17	141	16	395	8	46	0	10
Total	156 of 2,220	92%	9 of 190	95%	17	141	16	395	8	46	0	10

Created with JaCoCo 0.8.7.202105040129

Haskell Coverage Report

Coverage Metric	Total	Cov.
expressions used	156/156	100%
boolean coverage	2/2	100%
guards	0/0	100%
if conditions	2/2	100%
qualifiers	0/0	100%
alternatives used	6/6	100%
local declarations used	11/11	100%
top-level declarations used	13/13	100%

module	Top Level Definitions		Alternatives		Expressions	
	%	covered / total	%	covered / total	%	covered / total
module Main	100%	13/13	100%	6/6	100%	156/156
Program Coverage Total	100%	13/13	100%	6/6	100%	156/156

شکل (3-9) معیارهای پوشش بر روی پیاده سازی های مسئله ی scheduling

مانند قبل، معیار ها روی زبان تابعی همگی به طور کامل پوشش داده شده اند. در واقع این امر دور از انتظار هم نیست چرا که در یک زبان تابعی و استفاده از higher order functions و استفاده ی محدود از دستورات شرطی باعث خواهد شد که تعداد شاخه های کمتری در کد به وجود بیاید و معیار های مسیر، پوشش شاخه ای و البته تمامی دستورات آن برنامه به طور کامل اجرا شوند. نکته ی مهم در این فاز پوشش زبان امری است، در واقع در این قسمت مجموعه ای پیاده سازی ها را داریم که روش تفکر پیاده سازی آنها متفاوت از یک دیگر است:

BatchScheduling.java

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Main2		73%		75%	6	19	7	49	0	7	0	1
Main3		91%		92%	3	17	2	36	1	4	0	1
Main5		96%		92%	2	11	1	30	1	4	0	1
Main10		98%		100%	1	19	1	63	1	9	0	1
Main8		98%		100%	1	13	1	42	1	3	0	1
Main7		98%		100%	1	11	1	39	1	3	0	1
Main9		98%		100%	1	17	1	37	1	4	0	1
Main4		98%		100%	1	11	1	27	1	2	0	1
Main1		97%		100%	1	12	1	38	1	5	0	1
Main6		100%		100%	0	11	0	34	0	5	0	1
Total	156 of 2,220	92%	9 of 190	95%	17	141	16	395	8	46	0	10

شکل (10-3) معیارهای پوشش بر روی پیاده سازی های جاوا در مسئله ی scheduling

لازم به توضیح است که هر یک از این کلاس ها که با نام Main و اندیس ترتیبی جلوی آنها یک پیاده سازی اند که در سایت hackerrank و برای این مسئله submit شده اند (غیر از Main1 که پیاده سازی آن در راستای این پروژه انجام شد)

برای معیار پوشش تابعی همه به طور کامل این پوشش را داشته اند. کلاس هایی که در آنها یک تابع missed وجود دارد در واقع کلاس هایی اند که از خود کلاس استفاده نکرده اند. در واقع این عدم استفاده همه ی معیار ها را تحت تأثیر قرار داده است.

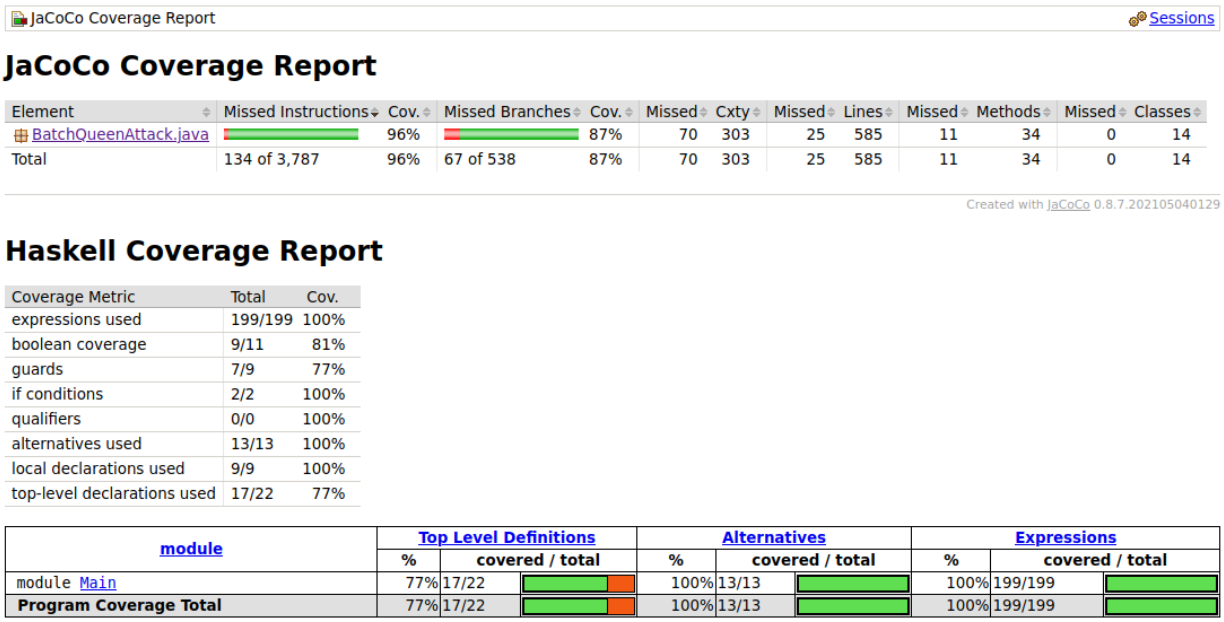
سایر معیار ها یعنی پوشش دستوری - پوشش شاخه ای و پوشش مسیر، به جز دو کلاس Main2 و Main3 پوشش کاملی دارند. این دو کد در واقع در خود قسمت هایی دارند که به طور منطقی هیچوقت قابل دسترس نیستند (این unreachable بودن نحوی نیست بلکه الگوریتم و پیاده سازی این دو کد به طریقی است که این دو عبارت هیچوقت قابل حصول نیستند.

فارغ از این موضوع، نکته ی مهم دیگر این است که در بین این نمونه کد ها، Main9 و Main10 مواردی اند که کد آنها مشکل دارد و خروجی صحیحی تولید نمیکند. با وجود این که در این کد ها باگ وجود دارد ولی تفاوتی میان پوشش آنها در هیچ معیاری با سایر کد های صحیح دیده نمیشود.

غیر از این مسئله سه مسئله ی دیگر نیز به طور مشابه بررسی شدند که یکی دیگر از آنها را در این قسمت بسط میدهیم، مسئله ای ساده که در عین ساده بودن راه حل آن نیاز به بررسی شرایط مختلف دارد. در این مسئله[3] می‌خواهیم تمام خانه های زیر حمله ی مهره ی وزیر را در یک صفحه ی شطرنج بسنجیم، ضمن اینکه مانع هایی هم همراه وزیر در صفحه حضور دارند. با نگاهی به راه حل های جاوای انتخاب شده به خوبی مشخص است که تا چه میزان برنامه نویسان در نوشتن راه حل برای این مسئله شرط های مختلف و مسیر ها و شاخه های زیادی را در کد خود ایجاد میکنند.

در این مسئله به دلیل پیاده سازی شرطی قصد داریم که حالت های خاص را هم در طراحی تست های خود لحاظ کنیم.به نظر میرسد که شرایطی مانند نبودن تعداد موانع، قرار گرفتن وزیر در گوشه ها برای ایجاد تست کیس ها مناسب و کافی باشند. بنابراین تست های خود را بر این اساس ایجاد میکنیم. شکل 3-11 معیار های پوشش را پس از اجرای تست ها بر روی کد ها نشان میدهد:

[3] https://www.hackerrank.com/challenges/queens-attack-2/problem



JaCoCo Coverage Report — Sessions

JaCoCo Coverage Report

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
BatchQueenAttack.java		96%		87%	70	303	25	585	11	34	0	14
Total	134 of 3,787	96%	67 of 538	87%	70	303	25	585	11	34	0	14

Created with JaCoCo 0.8.7.202105040129

Haskell Coverage Report

Coverage Metric	Total	Cov.
expressions used	199/199	100%
boolean coverage	9/11	81%
guards	7/9	77%
if conditions	2/2	100%
qualifiers	0/0	100%
alternatives used	13/13	100%
local declarations used	9/9	100%
top-level declarations used	17/22	77%

module	Top Level Definitions		Alternatives		Expressions	
	%	covered / total	%	covered / total	%	covered / total
module Main	77%	17/22	100%	13/13	100%	199/199
Program Coverage Total	77%	17/22	100%	13/13	100%	199/199

شکل (3-11) معیارهای پوشش بر روی پیاده سازی های مسئله ی queen attack

در خروجی هسکل مانند همیشه معیار های پوشش کامل به نظر میرسند. پوشش کامل بر روی معیار های پوشش دستوری و مسیر و شاخه ای داریم (همانطور که پیشتر توضیح داده شد در guard ها به این علت که عموما حالت نهایی مقدار otherwise است همیشه درست تلقی شده و حالت bool coverage و guard آن به صورت غیر قابل پوشش بیان میشود) به علاوه در مورد پوشش تابعی نیز باید گفته شود که پوشش کامل است، دلیل تفاوت ادعای ذکر شده با گزارش این است که Top Level Declaration تنها شامل توابعی که برنامه نویس به طور صریح تعریف میکند نیست. بلکه هنگامی که با استفاده از deriving یکی از class های زبان هسکل را extend میکنیم توابع موجود آنها نیز همراه با تایپ جدید تعریف میشوند. که ممکن است از همه ی آنها استفاده نکنیم. در این قسمت ما تایپ جدید Direction را تعریف کرده ایم که سه کلاس Eq - Show - Ord را extend میکند، با این وجود از همه ی تابع هایی که در این کلاس ها تعریف شده اند در کد استفاده نشده است. به هر حال hpc برای مشخص کردن این توابع دستوری را در نظر گرفته است:

```
→ haskell git:(main) ✗ hpc report Main --decl-list
100% expressions used (199/199)
 81% boolean coverage (9/11)
      77% guards (7/9), 2 always True
     100% 'if' conditions (2/2)
     100% qualifiers (0/0)
100% alternatives used (13/13)
100% local declarations used (9/9)
 77% top-level declarations used (17/22)
unused declarations:
     Main./=
     Main.<=
     Main.>=
     Main.showList
     Main.showsPrec
```

شکل (3-12) **موارد** unused declaration **در پیاده سازی هسکل در مسئله ی** queen attack

این مسئله ماهیتا حالت های مختلفی را در بردارد که باعث میشود در کد هسکل نیز آن ها را در نظر بگیریم. حالت هایی مانند حالت قرار گیری یک خانه نسبت به مهره ی وزیر و یا فاصله ی بین دو خانه در شطرنج و ... هر چند همه ی این ها بستگی به راه حل دارند اما آنچه که از مسئله بر می آید چنین است.

حال میخواهیم این معیار ها را بر روی زبان امری نیز ببینیم. پیش از آنکه وارد معیار بر روی هر یک از کد ها شویم هم مشخص است که معیار های پوشش شاخه ای و مسیر نسبت به حالت های قبل کاهش قابل توجهی داشته اند. این نکته بسیار مهم است. پرسش این است که آیا میتوان با بیشتر کردن تست ها این موارد را پوشش داد یا خیر. در واقع قصد داریم این نکته را بسنجیم که آیا فقدان تست کافی و بررسی شدن حالت های خاص بوده است که باعث شده معیار های پوشش مسیر و شاخه ای برخلاف دو معیار دیگر که ثابت ملنده لند، کاهش یابند یا خیر به همین منظور تست اتوماتیک خود را با کانفیگ هایی که موجب میشوند تست های بیشتری تولید شوند، تنظیم میکنیم.

نتایج معیار ها در حالت عادی در شکل 3-13 **آورده شده است:**

BatchQueenAttack.java

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Main1		100%		100%	0	11	0	35	0	6	0	1
Main1.Direction		100%		n/a	0	1	0	3	0	1	0	1
Main1.Point		100%		100%	0	13	0	17	0	3	0	1
Main10		91%		78%	15	47	12	111	1	2	0	1
Main11		98%		88%	3	11	1	29	1	2	0	1
Main12		93%		91%	6	38	4	78	1	3	0	1
Main2		99%		88%	9	36	1	47	1	2	0	1
Main3		98%		100%	1	10	1	25	1	2	0	1
Main4		98%		86%	7	24	1	34	1	2	0	1
Main5		99%		77%	12	26	1	57	1	2	0	1
Main6		98%		88%	8	32	1	44	1	2	0	1
Main7		98%		100%	1	14	1	41	1	3	0	1
Main8		95%		100%	1	5	1	16	1	2	0	1
Main9		92%		87%	7	35	1	48	1	2	0	1
Total	134 of 3,787	96%	67 of 538	87%	70	303	25	585	11	34	0	14

شکل (3-13) معیارهای پوشش بر روی پیاده سازی های جاوا در مسئله ی queen attack

پس از افزایش تعداد تست های اتوماتیک به میزان صد در صد (دو برابر حالت قبل) بسیاری از معیار ها به حالت کامل خود نزدیک شدند. و می توان گفت به طور تقریبی برابر مقدار میانگین مسئله ی قبلی یعنی مسئله ی scheduling شده اند:

BatchQueenAttack.java

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Main1		100%		100%	0	11	0	35	0	6	0	1
Main1.Direction		100%		n/a	0	1	0	3	0	1	0	1
Main1.Point		100%		100%	0	13	0	17	0	3	0	1
Main10		99%		96%	4	47	1	111	1	2	0	1
Main11		98%		88%	3	11	1	29	1	2	0	1
Main12		93%		92%	5	38	4	78	1	3	0	1
Main2		99%		95%	4	36	1	47	1	2	0	1
Main3		98%		100%	1	10	1	25	1	2	0	1
Main4		98%		97%	2	24	1	34	1	2	0	1
Main5		99%		87%	7	26	1	57	1	2	0	1
Main6		98%		98%	2	32	1	44	1	2	0	1
Main7		98%		100%	1	14	1	41	1	3	0	1
Main8		95%		100%	1	5	1	16	1	2	0	1
Main9		99%		98%	2	35	1	48	1	2	0	1
Total	59 of 3,787	98%	22 of 538	95%	32	303	14	585	11	34	0	14

شکل (3-14) معیارهای پوشش بر روی پیاده سازی های جاوا در مسئله ی queen attack پس از افزایش مجموعه تست

این معیار ها را برای دو مسئله ی دیگر نیز به دست آورده و در مجموع دیتای این بخش را در جدول شماره ی 3-3 آورده ایم (در قسمت Non-Divisible subset در واقع هیچ شاخه و مسیری وجود نداشته است و به همین دلیل پوشش را کامل محسوب کرده ایم)

معیار	معیار پوشش تابعی		معیار پوشش عبارت		معیار پوشش شاخه ای		معیار پوشش مسیر	
زبان/مسئله	هسکل	جاوا	هسکل	جاوا	هسکل	جاوا	هسکل	جاوا
scheduling	100	82	100	92	100	95	100	88
Non divisible subset[4]	100	63	100	96	100	96	100	95
Web of lies[5]	100	80	100	90	100	76	100	64
Queen attack	100	67	100	98	100	96	100	90

جدول (3-3) معیار های پوشش بر روی چهار مسئله ی پیاده سازی شده از مسائل online contests

3-3- پیاده سازی پروژه ها

در قسمت های قبل به پیاده سازی کد های ساده و مسائل تعریف شده در online contest **ها که پیچیده تر بودند، و بررسی معیار های پوشش در آنها پرداختیم. چنانچه مشاهده شد معیار های پوشش مقدار قابل قبولی داشتند و به نظر میرسد که رابطه ی نزدیکی نیز با یکدیگر دارند و در مسائل فاز دوم و به خصوص مثال آخر متوجه شدیم که با بالا رفتن یکی از آنها سایرین نیز افزایش می یابند. حال میخواهیم که برای بخش نهایی پروژه دو پروژه ی درس مهندسی اینترنت را هم پیاده سازی کرده تا بتوانیم این معیار ها را نه تنها در الگوریتم ها و مسائل کوچک بلکه در دو پروژه کوچک مقیاس نیز بررسی کنیم.**

[4] https://www.hackerrank.com/challenges/non-divisible-subset/problem

[5] https://codeforces.com/contest/1548/problem/A

1-3-3- بلبلستان

همانطور که پیشتر نیز گفته شد، این پروژه برای درس مهندسی اینترنت دانشکده ی کامپیوتر تهران تعریف شده و جنبه ی آموزشی داشته است. با این وجود با داشتن معیار های ما یعنی یک پروژه با مقیاس مناسب در زبان جاوا همخوانی دارد و از طرفی میتوان با داشتن پیاده سازی های متفاوت دانشجویان میانگینی از معیار ها را به دست آورد که دیتای آن می تواند معیار بهتری برای سنجش باشد. بلبلستان پروژه ای برای طراحی سامانه ی انتخاب واحد است که فاز نخست آن به صورت کامند لاین طراحی شده و به همین منظور دارای استاندارد ورودی برای دریافت دستورات و دیتاهای مربوط به هر دستور است، به علاوه خروجی ها نیز دارای استاندارد خاص تعریف شده در پروژه اند. همچنین پیاده سازی این پروژه با استفاده از ابزار maven انجام شده است که باعث شده گرفتن خروجی از پروژه ها به کاری سهل تبدیل شود. در ادامه قصد داریم که پیاده سازی این پروژه را در زبان هسکل شرح دهیم و با توضیح و تولید تست کیس های مورد نظر معیار های پوشش را برای هسکل و سایر پروژه ها که به زبان جاوا و توسط دانشجویان دیگر پیاده سازی شده اند به دست آوریم.

همانطور که میدانیم هسکل زبانی declarative است و دیتاها در آن تغییر نخواهند کرد. بنابراین برای طراحی این زبان نیاز داریم که state ای از برنامه را نگه داریم که نام آن را در پروژه SystemState گذاشته ایم. تعریف SystemState به صورت زیر است:

```
type SystemState = (DataBase, [Response])
```

DataBase تایپی است که دیتای مورد نظر برنامه را در خود نگه میدارد و پس از اجرای هر دستور ممکن است مقدار جدیدی به خود بگیرد. [Response] هم آرایه ای از پاسخ هاست که پاسخ به یک command ورودی است.

نحوه ی کار کل برنامه که در تابع runProgram پیاده سازی شده است، به این صورت است که در ابتدا همه ی دستورات را دریافت کرده. سپس یک initial state که برابر یک جفت از یک دیتابیس خالی و یک آرایه ی خالی از Response ها است ایجاد می‌کند. حال به ازای هر کامند database را

عوض کرده و پاسخ آن دستور را هم به آرایه ی responses اضافه میکند، در نهایت همه ی پاسخ ها را به پاسخ استاندارد مطرح شده در صورت پروژه تبدیل کرده و نشان میدهد.

تابع exec وظیفه ی اجرای هر دستور را دارد بنابراین یک دیتابیس که بیانگر حللت فعلی سیستم است را به همراه یک دستور گرفته و در خروجی خود یک دیتابیس که بیانگر حالت جدید است را به همراه یک پاسخ که پاسخ دستور متناظر است برمی‌گرداند. در این تابع با توجه به اینکه Command از چه نوعی است، تابع نوشته شده برای آن دستور که در ماژول Application پیاده سازی شده، اجرا می‌گردد، این تابع نیز می‌بایست در ورودی یک دیتابیس و یک دستور بگیرد و در خروجی یک database و یک response برگرداند.

در نهلیت‌بالاترین لول این ماژول،تابع ران قرار گرفته که وظیفه ی خولندن همه ی مقادیر ورودی و پایپ‌لاین آن به parser های تعریف شده و اجرای runProgram را به همراه نمایش همه ی پاسخ ها به عهده دارد .

```haskell
type Scanner = State [C.ByteString]
type SystemState = (DataBase, [Response])

runScanner :: Scanner a -> C.ByteString -> a
runScanner = runScannerWith C.lines

runScannerWith :: (C.ByteString -> [C.ByteString]) -> Scanner a -> C.ByteString -> a
runScannerWith t s = evalState s . t

changeSystemState :: SystemState -> (DataBase, Response) -> SystemState
changeSystemState systemState (database, response) = (database, snd systemState ++ [response])

runProgram :: Scanner String
runProgram = do
    commands <- parseCommands
    let database = getEmptyDataBase
    let systemState = foldl (\ss command -> changeSystemState ss (exec command (fst ss))) (database, []) commands
    let responses = unlines $ map (toDTO) (snd systemState)
    return responses


exec :: Command -> DataBase -> (DataBase, Response)
exec command database = case command of
    AddCourse args -> Application.addCourse database args
    AddStudent args -> Application.addStudent database args
    GetCourses args -> Application.getCourses database args
    GetCourse args -> Application.getCourse database args
    AddToWeeklySchedule args -> Application.addToWeeklySchedule database args
    RemoveFromWeeklySchedule args -> Application.removeFromWeeklySchedule database args
    GetWeeklySchedule args -> Application.getStudentWeeklySchedule database args
    FinalizeSchedule args -> Application.finalizeWeeklySchedule database args
    Command.BadCommand -> (database, Response.BadCommand)


run :: IO ()
run = C.interact $ runScanner (C.pack <$> runProgram)
```

شکل (15-3) پیاده سازی کد هسکل handler در پروژه ی بلبلستان

دیتابیس خود یک record type است که شامل سه آرایه از تایپ های Student، Course و StudentScheduleCourse است. برای تغییر این دیتابیس، توابعی در ماژول DataAccess نوشته شده اند که با گرفتن یک database و یک مقدار، این مقدار را یا به دیتابیس اضافه یا از آن حذف و یا آن را آپدیت و یا با گرفتن کلید یکی از این Entity ها آن را جستجو میکند. در پیاده سازی دستوراتی مانند AddStudent به این توابع DataAccess نیاز پیدا خواهیم پیدا کرد. همچنین ساختار Command و CommandArgument نیز به ترتیب در ماژول های همنام آنها آورده شده. در واقع هر Command یک تایپ از نوع CommandName CommandArgument است (. در هر تابع موجود در ماژول Application، یکی از این Command ها رابه همراه args آن گرفته و پردازش لازم را انجام میدهد. نکته دیگر این است که Entity های Student و Course و StudentSchedulCourseبه این صورت تعریف شده اند که دیتای مورد نیاز این مفاهیم را در خود نگه داشته و مختصر functionality هم برای آنها تعریف شده است. همچنین این موارد را از data structure ورودی مستقل کرده ایم تا وابستگی ای بین این دو نداشته باشیم. سایر ساختار ها و کد ها پیاده سازی است و از توضیح آنها اجتناب میکنیم.

با همه ی این تفاسیر میخواهیم که برای این پروژه تست طراحی کنیم. تست های خود را بر این اساس طراحی می‌کنیم که مجموعه ورودی ها بتوانند به این منجر شوند که همه ی خروجی های ممکن در کد دیده شوند. در واقع اگر سیستم را به صورت شکل ۱۰ متصور شویم قصد داریم که همه ی گره های انتهایی این گراف را بپیماییم. معیار سنجش آن هم دشوار نیست، کافی است که در فایل Application معیار پوشش مسیر کامل باشد. در واقع قصد داریم Node Coverage ای روی گراف شکل ۱۰ بر روی همه ی نقاط نهایی گراف داشته باشیم. پرسشی که لازم به پاسخ دارد این است که در این صورت آیا از اصل خود مبنی بر طراحی تست ها به صورت black box دور شده ایم؟ نخست باید گفت که این تصمیم را به خروجی های ممکن محدود ساخته ایم. به علاوه چنانچه هر یک از این گره های انتهایی رابایک‌تایپ از Response تناظر برقرار کنیم، می‌توان گفت فقط حالت های مختلف پاسخ به دستورات برنامه را در طراحی خود تأثیر داده‌ایم، که در واقع جزو پارامتر های

بیزینسی است و با خواندن صورت پروژه هم میتوان به آن رسید.

با این فرض با بررسی تک تک دستورات و حالت های خاصی که ایجاد میکنند تست های خود را طراحی میکنیم.به عنوان مثالی از حللت های خاص در دستور addToWeeklyScheduling حللت هایی که برای آن می‌توان غیر از حالت موفقیت آمیز بودن اضافه شدن درس متصور شد این است که درس یا دانشجو یافت نشوند. ساختار دستورات به این شکل است که ابتدا تعدادی کامند addCourse و addStudent برای ایجاد دروس و دانشجویان اضافه شده، سپس تعدادی دستور getOffering برای برخی از دانشجویان تولید شده، سپس برای تعدادی از دانشجویان تعدادی واحد انتخاب کرده و با دستور addWeeklySchedule آنها را به برنامه های هر یک اضافه میکنیم. بخشی از این دانشجویان برنامه هایی دارند که با در زمان کلاس، امتحان یا .. مشترک لند، بخشی کف و بخشی هم سقف تعداد مجاز را رعلیت نکرده لند. سپس برای همه ی دانشجویان دستور getWeeklySchedule و برای بخشی از آن ها removeWeeklySchedule را برای بعضی از درس ها انتخاب کرده و در نهلیت برای دانشجویان دستور finalize کردن برنامه را صادر میکنیم. طبق گزارش مربوط به ماژول Application میتوان گفت که دستورات ما همه ی حللت های خروجی را پوشش داده اند. اما نتیجه ی معیار ها بر روی کل برنامه و کد به صورت زیر است:

Haskell Coverage Report

Coverage Metric	Total	Cov.
expressions used	1497/1615	92%
boolean coverage	36/63	57%
guards	33/52	63%
if conditions	3/11	27%
qualifiers	0/0	100%
alternatives used	110/143	76%
local declarations used	96/99	96%
top-level declarations used	125/177	70%

module	Top Level Definitions		Alternatives		Expressions	
	%	covered / total	%	covered / total	%	covered / total
module Application	100%	9/9	100%	28/28	100%	417/417
module ClassTime	50%	2/4	100%	2/2	91%	22/24
module Command	-	0/0	-	0/0	-	0/0
module CommandArguments	65%	26/40	-	0/0	93%	122/130
module CommandParser	100%	11/11	64%	16/25	93%	135/145
module Course	90%	9/10	52%	10/19	73%	46/63
module DTO	40%	2/5	-	0/0	84%	16/19
module DataAccess	100%	7/7	100%	6/6	100%	78/78
module DataBase	100%	4/4	-	0/0	100%	7/7
module DataMapper	100%	4/4	-	0/0	91%	68/74
module DateTime	81%	9/11	63%	7/11	89%	157/176
module Day	30%	4/13	-	0/0	-	0/0
module ExamTime	50%	2/4	-	0/0	92%	25/27
module Handler	100%	6/6	88%	8/9	95%	68/71
module Main	100%	1/1	-	0/0	100%	2/2
module RangeTime	50%	3/6	-	0/0	80%	36/45
module ReaderLib	100%	3/3	100%	2/2	100%	28/28
module Response	100%	1/1	92%	13/14	93%	136/146
module Student	50%	2/4	33%	3/9	21%	3/14
module StudentScheduleCourse	100%	8/8	66%	6/9	86%	45/52
module TermCourse	50%	4/8	-	0/0	87%	28/32
module TermCourseStatus	18%	2/11	100%	5/5	100%	5/5
module Time	83%	5/6	100%	2/2	86%	43/50
module Util	100%	1/1	100%	2/2	100%	10/10
Program Coverage Total	70%	125/177	76%	110/143	92%	1497/1615

شکل (3-16) معیارهای پوشش بر روی پیاده سازی هسکل در پروژه ی بلبلستان

همانطور که مشخص است پارامتر های پوشش ما بر روی زبان هسکل، نسبت به فاز های قبل، به مقدار قابل توجهی کاهش داشته اند. در پوشش تابعی حدود ۳۰ درصد، در پوشش دستوری حدود ۸ درصد، در معیار پوشش شاخه ای و مسیر هم به همین ترتیب حدود ۵۰ و ۲۵ درصد کاهش پوشش داشته ایم. علت اصلی این کاهش این است که برخلاف حالات قبل که ما به تک تک ورودی ها احاطه داشتیم و میتوانستیم با تغییر آنها به راحتی پوشش کاملی بگیریم، در یک پروژه ی با دیتا های واقعی چنین امکانی نخواهیم داشت، درواقع بسیاری از کد ها شامل حالات و عباراتی می شوند که برای کامل بودن کد لازم است و ممکن است هیچ گاه به کار نروند. این بخش ها مربوط به هندل کردن ارور ها، ورودی های ناخواسته و ... است. در واقع نباید انتظار پوشش کامل هیچ معیاری را در یک پروژه ی حتی کوچک داشت. نکته ای که اما هدف ما از طراحی این تست بود رسیدن به پوشش کامل همه ی خروجی های ماژول Application بود که پوشش ۱۰۰ درصد را شامل میشود.

یعنی همه ی پاسخ های خروجی ممکن تولید شده اند. در واقع میتوان به تعبیری این خروجی ها را سناریو های پروژه نیز دانست و از این رو معیار مهمی هم محسوب میشوند. در ادامه معیارهای پوشش چهار پروژه ی پیاده سازی شده ی دیگر که به زبان جاوا همین پروژه را پیاده سازی کرده اند آورده شده:

JaCoCo Coverage Report

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
ir.ut.offering		94%		83%	6	41	7	102	1	23	0	4
ir.ut.student		97%		94%	4	51	2	116	1	25	0	3
ir.ut		98%		88%	3	36	1	101	0	20	0	3
default		57%		n/a	1	2	1	3	1	2	0	1
ir.ut.exception		100%		n/a	0	11	0	22	0	11	0	11
Total	46 of 1,469	96%	12 of 113	89%	14	141	11	344	3	81	0	22

شکل (3-17) معیارهای پوشش بر روی نخستین پیاده سازی در پروژه ی بلبلستان

JaCoCo Coverage Report

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
ie		88%		86%	39	181	71	450	18	93	0	9
Exceptions		80%		n/a	2	12	4	18	2	12	1	9
default		40%		n/a	1	2	1	3	1	2	0	1
Total	216 of 1,803	88%	22 of 160	86%	42	195	76	471	21	107	1	19

شکل (3-18) معیارهای پوشش بر روی دومین پیاده سازی در پروژه ی بلبلستان

JaCoCo Coverage Report

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
ie		94%		83%	19	103	17	263	2	31	0	7
default		40%		n/a	1	2	1	3	1	2	0	1
Total	65 of 1,095	94%	22 of 136	83%	20	105	18	266	3	33	0	8

شکل (3-19) معیارهای پوشش بر روی سومین پیاده سازی در پروژه ی بلبلستان

JaCoCo Coverage Report

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
ie		93%		92%	12	127	37	303	4	71	0	11
default		70%		n/a	1	2	1	4	1	2	0	1
Total	86 of 1,283	93%	8 of 105	92%	13	129	38	307	5	73	0	12

شکل (3-20) معیارهای پوشش بر روی چهارمین پیاده سازی در پروژه ی بلبلستان

همانطور که مشخص است توانسته ایم که با گرفتن پوشش کافی بر روی کد هسکل به حد نصاب خوبی بر روی کد جاوا نیز دست پیدا کنیم. در معیار پوشش تابعی، چنانکه پروژه ی P2 را که شامل setter ها و getter های بدون استفاده است را فاکتور بگیریم، تقریبا همه ی توابع در کلیه ی پروژه ها استفاده شده اند. معیار پوشش دستوری به طور میانگین ۹۲ درصد است. به علاوه معیار های پوشش شاخه ای و دستوری به طور میانگین نتایج قابل قبولی را نشان میدهند. در واقع این معیار خوب بودن با توجه به فاز های قبلی که مسائل بسیار کوچکتر بوده اند عنوان می‌شود. چنانچه اختلاف معناداری بین نتایج این فاز که پروژه ای به مراتب بزرگ تر از قسمت های قبل است با نتایج فاز دو دیده نمیشود. و معیار ها به خوبی این موضوع را نشان داده اند. در ادامه قصد داریم که مانند فاز های قبل با تغییر پوشش روی زبان هسکل، اثر آن را بر روی کد جاوا نیز ببینیم، تا بتوانیم رابطه ی بین این دو زبان و به علاوه اثر آن بر روی هریک از معیار ها را نیز ببینیم. به عنوان مثال اگریک دستور کلیدی را در تست ها انجام ندهیم پوشش ما تا چه میزان تحت تاثیر قرار میگیرد. نتایج این قسمت در جدول 4-3 آمده است. دقت شود که داده های ستون های جاوا در این جدول بر اساس میانگین چهار پروژه به دست آمده اند)

تست	معیار پوشش تابعی		معیار پوشش عبارت		معیار پوشش شاخه ای		معیار پوشش مسیر	
	هسکل	جاوا	هسکل	جاوا	هسکل	جاوا	هسکل	جاوا
حالت تست کامل	70	89	92	92	27	87	66	84
بدون تست های حالت های نبود درس یا دانش آموز	70	88	90	91	27	79	66	81
بدون تست های عدم فاینالایز کردن برنامه	64	81	75	73	18	59	59	66
بدون تست حذف درس از برنامه	68	85	88	86	27	79	61	77

جدول (4-3) معیار های پوشش بر روی پروژه ی بلبلستان با تغییر روی مجموعه تست ها

3-3-2- لقمه

یکی دیگر از پروژه هایی که در این قسمت مورد بررسی قرار گرفت پروژه ی لقمه بود که مانند بلبلستان، این پروژه نیز برای درس مهندسی اینترنت دانشکده ی کامپیوتر تهران تعریف شده بود و فاز نخست آن ساختار مشابهی دارد. موضوع طراحی یک رستوران آنلاین بود که دستوراتی از جمله اضافه کردن رستوارن، اضافه کردن غذا، مشاهده ی رستوران ها یا یک رستوران خاص، سفارش غذا و یا حذف آن و نهایی کردن سفارش و مشاهده ی رستوران های برتر، خواسته های این پروژه بوده اند. برای طراحی این پروژه در زبان هسکل، از ساختار های مشابه مانند پروژه ی قبل استفاده کرده ایم. و در اینجا تنها نتایج خروجی را براساس تست های تولید شده آورده ایم.

معیار	معیار پوشش تابعی		معیار پوشش عبارت		معیار پوشش شاخه ای		معیار پوشش مسیر	
زبان/مسئله	هسکل	جاوا	هسکل	جاوا	هسکل	جاوا	هسکل	جاوا
بلبلستان	70	89	92	92	27	87	66	84
لقمه	90	91	95	95	16	91	66	88

جدول (3-5) معیار های پوشش بر روی دو پروژه ی بلبلستان و لفمه

فصل 4: تحلیل نتایج و تعیین معیار ها

با مشخص شدن نتایج هر یک از فاز ها و مقایسه ی آنها با یکدیگر قصد داریم که در این قسمت به شرح نتایج حاصله از این پروژه برای مشخص کردن معیار پوشش مناسب برای پیاده سازی زبان تابعی برای رسیدن به پوشش مطلوب در زبان امری بپردازیم.

4-1- تحلیل نتایج

با اجرای سه فاز متفاوت در پروژه و بررسی هر یک از معیار ها در این قسمت سعی داریم که تحلیل نهایی ای بر روی هر معیار ارائه داده با توجه به نتایج به دست آمده در قسمت های قبل مناسب و یا نامناسب بودن هر معیار را برای هدف پروژه ی خود ذکر کنیم. پیش از هر چیز لازم است که به جدول 4-1 توجه شود. این جدول شامل correlation[1] به دست آمده از معیار های متناظر در دو زبان است که با توجه به نتایج جدول ۸ که تست های خود را برای بررسی تاثیر پذیری بر روی معیار ها، تغییر می دادیم به دست آمده است. نتایج نهایی بر حسب آمار جداول مراحل قبل و این جدول گزارش شده است (مقدار correlation بین منفی یک تا یک می‌تواند باشد که مقدار یک نشان دهنده ی حداکثر correlation و مقدار منفی یک نشانه ی ;کاملا غیر مرتبط بودن همبستگی است)

	Correlation			
	معیار پوشش تابعی	معیار پوشش عبارت	معیار پوشش شاخه ای	معیار پوشش مسیر
تغییرات تست های بلبلستان	0.975	1	- 0.229	0.894

جدول (4-1) مقادیر correlation بین معیارهای پوشش در تغییر معیار ها به دنبال تغییر مجموعه تست ها در پروژه ی بلبلستان

- **معیار پوشش تابعی: در پروژه های فاز های اول که الگوریتم ها و مسائل کوچک بودند توانستیم پوشش کاملی بر روی این معیار بگیریم، اما در مسائل قسمت آخر که پروژه ها مقیاس بزرگتری پیدا کردند، تا میزان قابل توجهی این معیار بر روی پیاده سازی هسکل افت پیدا کرد که البته علت اصلی این موضوع، تعریف ناخواسته ی بسیاری از توابع به علت deriving ها بود، اما این مسئله نمی‌تواند خللی در نتیجه ایجاد کند چرا که این توابع بیشتر جنبه ی رفتاری یک دیتا در هسکل را مشخص میکنند و نه بیشتر. در زبان جاوا اما این**

[1] Spearman's Correlation

معیار در قسمت های مختلف به میزان قابل انتظاری حفظ شد. معیار پوشش تابعی به علت ماهیت زبان تابعی بسیار مهم تلقی می‌شود، در واقع چنانچه متغیر ها را در یک زبان تابعی یک state در نظر بگیریم که توابع نشان دهنده ی چرخش بین این حالت ها باشند، میتوان گفت که با پوشش تابعی همه ی حالت های عوض شدن این state ها را چک کرده ایم (هر چند همه ی state ها را نبینیم). همچنین در زبان امری نیز توابع به معنی رفتار ها و عملکرد های یک پیاده سازی است و پوشش مناسب بر روی آن به معنی این است که توانسته ایم بخش اعظمی از کد را پوشش دهیم. نتیجه ای که در این معیار به دست آمد این است که با پوشش مناسبی بر روی این معیار در زبان تابعی می‌توان انتظار داشت که عملکرد پیاده سازی زبان امری نیز به میزان قابل توجهی سنجیده میشود. چنانچه هنگامی که تست ها را به طور ناکامل اجرا کردیم، با کاهش این معیار بر روی زبان هسکل، شاهد کاهش آن بر روی زبان جاوا نیز بودیم. این موضوع هر چند بدیهی به نظر میرسد اما نکته ی مهمی است چرا که توابع در دو زبان جزو معدود ساختار های مشترک اند و جزو عناصر مهم هر دو پیاده سازی تلقی می‌شوند. بنابراین با بزرگ شدن پروژه ها اهمیت آنها افزایش بیشتری هم می یابد چرا که میتوان گفت بزرگترین ساختار اجرایی اند که میتوان بر روی آنها پوشش را تعریف کرد.

- معیار پوشش دستوری: پوشش دستوری با وجود اینکه ساختار های دو زبان متفاوت اند و حتی دستورات این دو زبان در برخی از موارد معادل هم‌جنسی ندارند، باز هم ارتباط مستقیمی بین این دو معیار مشاهده شد، این معیار تنها معیاریست که حتی در فاز های ابتدایی هم ارتباط همبستگی مناسبی بین دو زبان در آن دیده شد. فارغ از اینکه علت این همبستگی با وجود ساختار های کاملا متفاوت چه میتواند باشد، میتوان با توجه به داده ها ادعا کرد که با پوشش مناسبی از این معیار بر روی زبان تابعی میتوان به پوشش مناسبی از آن بر روی پوشش زبان امری نیز رسید.
- معیار پوشش شاخه ای: چنانچه به خروجی های فاز دو، یعنی جایی که پیاده سازی های مختلفی از افراد مختلف بررسی شد توجه کنیم، در میابیم که بسیاری از پیاده سازی هایی که در زبان امری نیازمند ساختار های شرطی اند در یک زبان تابعی چنین پیاده سازی نمی‌شوند. در واقع سبک کد زنی و ساختار های زبان تابعی باعث میشود که تعداد شاخه

های موجود در یک پیاده سازی زبان تابعی بسیار کمتر از شاخه های موجود در پیاده سازی زبان امری شود. این موضوع در فاز سوم نیز قابل مشاهده است با این تفاوت که این درصد بر روی زبان تابعی بسیار کم و در پیاده سازی های جاوا پوششی ۹۰ درصدی داشت. با این تفسیر میتوان گفت که در پیاده سازی های کوچک مانند فاز دو و یا پیاده سازی پروژه های بزرگ تر مانند فاز سوم موضوعی که باعث میشود پوشش کاملی بر روی معیار پوشش شاخه ای بر روی زبان امری داشته باشیم، داشتن پوشش کامل بر روی پوشش معیار شاخه ای بر زبان تابعی نیست. چنانچه اگر به مثال queeAttack بازگردیم، به طور میانگین در هر یک از دوازده کد پیاده سازی شده ی جاوا ۴۵ شاخه وجود دارد در حالی که با احتساب if condition ها و افزودن و حتی افزودن کلیه ی boolean coverage در زبان هسکل (که البته شاخه محسوب نمیشود) به نصف این تعداد میرسیم. میتوان گفت که برای این معیار در این دو زبان correlation مناسبی وجود ندارد. و با پوشش کامل این معیار بر روی زبان هسکل نمیتوان انتظار داشت که این معیار بر روی زبان جاوا نیز به خوبی پوشش داده شود.

- معیار پوشش مسیر: این معیار در فاز دوم، به طور تقریبا کامل در هر دو زبان امری و تابعی پوشش داده شد. در فاز سوم نیز همین نتیجه گرفته شد (با توجه به حذف کردن حالت های غیر قابل دسترس در هسکل) و حتی هنگامی که از حالت هایی خاص تست ها و برخی از دستورات آن را در پروژه ی بلبلستان حذف کردیم، باز هم این ارتباط بر خلاف پوشش شاخه ای برقرار ماند. به نظر میرسد که correlation خوبی بین این دو معیار در هر دو زبان وجود دارد. به عبارتی می‌توان گفت از آنجا که پوشش مسیر معیاری است از حالت های رسیدن از entry point ها به exit point ها با پوشش کامل این موضوع در تست های خود توانسته ایم این معیار را ارضا کنیم. به علاوه با استفاده از این معیار و به کار بردن این روش که ترکیب همه ی حللت های ممکن ورودی-خروجی را در type Response خود بیاوریم، میتوانیم خود این پوشش را به طور مستقیم در تست زنی خود بیاوریم.

فصل 5

فصل 5: **جمع‌بندی، نتیجه‌گیری و پیشنهادها**

5-1- جمع‌بندی و نتیجه گیری

در این پروژه در ابتدا به بررسی تفاوت های دو زبان تابعی و امری و سپس معرفی معیار های پوشش گراف و ابزار های به کار رفته در پروژه برای محاسبه ی این معیار ها پرداختیم. در گام دوم در طول سه فاز به بررسی معیار های پوشش تعریف شده بر روی زبان های تابعی و امری بر روی مسائل کوچک و ساده، چهار مسئله از مسائل کانتست های آنلاین برنامه نویسی و در نهایت به دو پروژه در مقیاس کوچک پرداختیم؛ به این ترتیب که با سعی در تولید تست های اتوماتیک با طراحی black-box و رسیدن به پوشش کامل زبان تابعی به پوشش مطلوبی بر زبان امری برسیم. در نهایت با مشخص شدن نتایج و به دست آمدن روابط با تغییر تست ها معیار های خود را مشخص کردیم.

5-2- نوآوری ها

اهمیت توسعه ی سریع نرم افزار در متدولوژی های توسعه ی نرم افزار و همه گیری این روش‌ها و نیاز به تست دوره ای و پیوسته ی محصول، در هر integration و delivery/deploy ، نیاز حرکت به سوی روش های جدید تست اتوماتیک و مخصوصا تست end to end بیش از پیش حس میشود. در این پروژه قصد داشتیم که معیار های پوششی را برای یک زبان تابعی مانند هسکل به نحوی که منجر به پوشش مناسب بر روی زبان امری مانند جاوا شود را به دست آوریم تا بتوانیم در این راستا که محصول خود را با توصیف تابعی تعریف کرده و در فرآیند های CI/CD با تولید تست اتوماتیک، مدل و محصول اصلی که با زبان امری تعریف شده است ارزیابی کنیم. به همین منظور در این پروژه، در سه فاز مختلف و از کدهای ساده تا پروژه های در مقیاس دانشگاه این ایده را دنبال کردیم که هر مسئله را با دو زبان هسکل و جاوا پیاده سازی کرده و با تغییر تست ها و تغییر در معیار های پوشش، به دنبال معیار مناسب برای هدف خود بگردیم. در نهلیت توانستیم که بین دو زبان و در واقع دو فلسفه و تفکر برنامه نویسی پلی ایجاد کنیم و معیارهای مشترکی بین هر دو به دست بیاوریم که

correlation مناسبی با یکدیگر داشته باشند. در انتها نیز با بررسی معیار ها به این نتیجه رسیدیم که سه معیار پوشش تابعی، پوشش دستوری و مسیر میتوانند این اطمینان را به ما بدهند که با پوشش مناسب آنها در توصیف تابعی از یک پروژه یا مدل، میتوان به پوششی مناسب در یک توصیف امری از آنها هم دست یافت. در نهایت این معیار ها در پروژه های فاز آخر بیش از نود درصد یافت شدند که در بین پروژه های صنعتی معیار خوبی است، مخصوصا با توجه به اینکه تست ها به صورت black box طراحی شدند.

5-3- محدودیت ها و پیشنهاد ها

در این پروژه خلا چند موضوع حس شد. نخست، ابزار های jacoco و hpc بودند که در معیار های تعریفی خود، به طور صریح نیازمندی های ما را در بر نداشتند، مانند پوشش تابعی در jacoco و hpc که هر کدام طبق داکیومنت های خود این ابزار ها همراه‌با موارد دیگری در گزارش می آیند که این وظیفه ی ما بود که در تحلیل های خود سهم دیگر موارد را از آنها حذف کنیم. دومین موضوع که می تواند به عنوان یک پروژه ی صنعتی و به عنوان ابزاری برای توسعه قرار بگیرد، ایجاد plugin مناسبی برای انجام مراحل این پروژه بود. در این پروژه برای به دست آوردن این معیار ها و راحت بودن اضافه کردن پروژه های جدید که در هر یک میبایست مراحل کامپایل پروژه ی هر دو زبان (که حتی در فاز دو و سه بیش از یک پروژه وجود داشت) و همچنین اجرای test generator و اجرای برنامه ها‌با مجموعه تست های تولیدی و در نهایت به دست آوردن گزارش و ترکیب آنها با یکدیگر script هایی نوشته شد که بتوانند این بستر را فراهم کنند و ضمن انجام این کار ها امکان مقایسه ی خروجی ها را هم در بر داشته باشند. این قسمت از این پروژه می تواند به عنوان پروژه ای جدید برای این سبک پروژه ها که میخواهیم دو یا چند توصیف از یک برنامه داشته باشیم استفاده شود. فراموش نکنیم در دنیای صنعتی امروزه ی کامپیوتر تا هنگامی که ابزار لازم برای یک ایده وجود نداشته باشد، آن ایده دنبال نخواهد شد.

فصل 6: مراجع

[1] P. Amman and Offutt, “Introduction to Software Testing”, 2 nd Ed., Cambridge University Press, 2017

[2] R. Bird, “Thinking Functionally with Haskell”, Cambridge University Press, 2015

[3] Zakeriyan, A., Khosravi, R., Safari, H., Khamespanah, E. 2021: Towards Automatic Test Case Generation for Industrial Software Systems Based on Functional Specifications. In: Fundamentals of Software Engineering - 9th IPM International Conference, FSEN. Tehran, Iran, 19-21 May

[4] HPC toolkit to get code coverage of haskell language http://wiki.haskell.org/Haskell_program_coverage

[5] Jacoco toolkit to get code coverage of java language https://www.eclemma.org/jacoco/trunk/doc/counters.html