
بسم الله الرحمن الرحيم

دانشگاه صنعتی شریف
دانشکده مهندسی کامپیوتر

آزمون کامپایلر تک گذره MninJava

سید سجاد میرزابابایی ـ امیرحسین کارگران خوزانی ـ رامتین باقری
(۹۹۳۰۱۹۳۸ـ۹۹۲۰۱۱۱۹ـ۹۹۲۱۰۱۴۲)

استاد درس

دکتر حسن میریان

تحت نظارت

علیرضا آقا محمدی

نیمسال دوم تحصیلی ۱۴۰۰ـ۱۳۹۹

فهرست مطالب

عنوان **صفحه**

فصل اول

مقدمه

۱ـ۱ معرفی برنامه مورد آزمون و هدف پروژه

برنامه مورد ازمون یک پروژه به زبان جاوا می‌باشد که وظیفه آن ترجمه برنامه ورودی به یک کد میانی ۳ آدرسه است. این کامپایلر تک گذره برای زبان MiniJava طراحی شده است. که این زبان قابلیت‌های کمتری نسبت به زبان جاوا دارد و زبان کوچکتری نسبت به آن محسوب می‌شود. این پروژه شامل ۵ قسمت اصلی می‌باشد: ۱. تحلیل گر لغوی ۲. تحلیل گر نحوی ۳. تحلیل گر معنایی ۴. مولد کد میانی ۵. خطا پرداز

این پروژه یک متن حاوی برنامه ای به زبان MiniJava را به عنوان ورودی قبول می‌کند و آن را ترجمه می‌کند و کدهای میانی ۳ آدرسه به شرح تصویر ۱ـ۱ را تولید می‌کند.

حال به عنوان کسانی که آزمون نرم‌افزار می‌دانند از ما خواسته شده که این پروژه را آزمون کنیم با این هدف که اگر خطایی در این برنامه وجود دارد ان خطا را به تیم برنامه نویس پروژه اعلام کنیم و همچنین توضیح دهیم که چرا این آزمون خوب است و چقدر خوب است و چگونه طراحی شده است. از انجا که هدف این آزمون رفع‌یابی خطاهای پروژه توسط خود تیم آزمون نیست تنها موارد آزمون مناسب باید طراحی شود. همچنین گفته شده است که قرار نیست که برای تک تک توابع در این پروژه آزمون نوشته شود و یک راه حل مناسب و منطقی ارائه شود.

توضیح	قالب کد سه‌آدرسه
عملوند اول و دوم جمع می‌شوند و در مقصد قرار می‌گیرند.	(ADD, S1, S2, D)
عملوند اول و دوم AND می‌شوند و در مقصد قرار می‌گیرند.	(AND, S1, S2, D)
محتوای مبدأ در مقصد قرار می‌گیرد.	(ASSIGN, S, D)
اگر S1 و S2 مساوی باشند در D مقدار true و در غیر این صورت false ذخیره می‌شود.	(EQ, S1, S2, D)
حاصل S بررسی می‌شود و در صورتی که false باشد به L جهش می‌کند.	(JPF, S, L)
پرش به L انجام می‌شود.	(JP, L)
اگر S1 از S2 کوچکتر باشد مقدار D برابر true و در غیر این صورت مقدار false می‌گیرد.	(LT, S1, S2, D)
عملوند اول در عملوند دوم ضرب می‌شود و در مقصد قرار می‌گیرد.	(MULT, S1, S2, D)
عملوند نقیض می‌شود.	(NOT, S, D)
محتوا بر روی صفحه چاپ می‌شود.	(PRINT, S)
عملوند دوم از عملوند اول کم می‌شود و در مقصد قرار می‌گیرد.	(SUB, S1, S2, D)

شکل ۱ ـ ۱ : کدهای ۳ آدرسه

۱ ـ ۲ فرض‌ها و محدودیت‌های درنظر گرفته شده

کد برنامه کامپایلر MiniJava خود بر اساس گرامر MninJava نوشته شده است که این گرامر به عنوان توصیف صحیح ما از کاری است که قرار است این برنامه انجام دهد، البته دقت شود که همه برنامه‌هایی که با این گرامر ساخته می‌شود، برنامه مورد قبول ما نیست و باید شروط دیگری که در ضمن کد نیز گذاشته شده است نیز رعایت شود. برای مثال این که این گرامر تک گذره است و اشاره‌ای رو به جلو رخ نباید بدهد یا identifier نباید جزو کلمات کلیدی ای مانند int یا class باشد یا اگر در ادامه identifier ای استفاده شد باید قبل از آن تعریف شده باشد یا به شیوه صحیح call شود یا دستور چاپ تنها محتوی عدد صحیح را چاپ کند. به همین منظور ازمون‌های مورد نیاز ورودی به برنامه ابتدا از طریق گرامر MiniJava ساخته می‌شوند و در حین ساخت به این موارد نیز توجه می‌شود (که البته تمامی specification برنامه دست ما نیست و این موارد جزو فرضیات اضافه برای شروع آزمون است).

همچنین فرض دیگر نیز آن است که برای اعداد int رقم‌های ۰ تا ۹ و برای literal ها ترکیب حروف کوچک و بزرگ انگلیسی در نظر گرفته شده است.

۱ ـ ۳ روش آزمون انتخابی و دلایل انتخاب آن

متدهای مختلفی برای تست این برنامه وجود دارد؛ برای مثال برای قسمتی از برنامه که دارای قواعد شروط اما و اگر زیاد است می‌توان از آزمون منطق یا برای مثال در قسمت‌هایی که شامل حلقه است و برنامه پیچیدگی‌های

الگوریتمی دارد از روش‌های مبتنی بر گراف و اگر برای مثال اگر تابع مورد بررسی حالت واسط دارد از روش‌های واسط می‌توان استفاده کرد. در ادامه خواهیم دید که چه روشی انتخاب شده و دلیل انتخاب این روش چه بوده است.

اما همانجور که قبلا گفته شد از آنجا که بهترین توصیفی که از برنامه داریم توسط گرامر به ما داده شده است، بهترین آزمون آن است که بر اساس این گرامر و فرضیات اضافه مساله ورودی معتبر را تولید کنیم و سپس آن را به به عنوان مورد آزمون به برنامه بدهیم با توجه به توصیف برنامه از کدهای میانی نیز می‌توانیم خروجی مورد انتظار معتبر را تولید کنیم.

به طور کلی از دلایل انتخاب می‌توان به موارد زیر اشاره کرد:

- از آنجا که گرامر به عنوان توصیف صحیح از برنامه حضور دارد بهترین راه برای تست استفاده مستقیم از خود گرامر برای تولید ورودی‌های معتبر است، از انجا که خروجی مورد معتبر نیز با توجه به شکل ۱ـ۱ قابل دستیابی است این روش را ممکن می‌کند.
- کد بخش های مختلفی دارد و پیچیدگی بسیاری در برخی از بخش‌ها دارد، همچنین به ازای هر تابع مجزا توصیفی از کار تابع، ورودی ها و خروجی‌های مورد انتظار وجود ندارد و این موضوع موجب می‌شود که تست تابع به تابع امکان پذیر نباشد. در صورتی که روش پیشنهاد شده نیاز ندارد تا به صورت مجزا توصیف تابع‌ها را بداند و با هر مقدار پیچیدگی در بالاترین سطح ممکن می‌تواند کد را بیازماید و اگر رفتار آن متفاوت با خروجی مورد انتظار بود آن را گزارش دهد.

۱ـ۴ معیارهای پوشش انتخابی

از انجا که در این آزمون از گرامر استفاده می‌کنیم از معیار پوشش production استفاده می‌کنیم. در این پوشش باید هر یک از قوانین گرامر حداقل یکبار trigger شوند، حداکثر تعداد موارد ازمون طراحی شده با این روش برابر تعداد قوانین هر گرامر است.

۱ـ۵ چگونگی استفاده از روش‌های انتخابی

ابتدا گرامر به فرم استانداردی که که در آن غیرترمینال‌ها با "" مشخص شده است در می‌آوریم. همچنین به هر یک از قوانین شماره‌ای داده شده است تا شمارش و رفرنس‌دهی راحت تر باشد. برای معادل کردن گرامر دو قانون ۵۳ و ۵۴ را به فرمی که در شکل ۱ـ۲ (یا فایل grammar.bnf به پیوست) مشاهده می کنید بازنویسی شده است و در نهایت به ۱۲۰ قانون مجزا دست پیدا شده است.

```
[1] Goal -> Source EOF
[2] Source -> ClassDeclarations MainClass
[3] MainClass -> "class" Identifier "{" "public" "static" "void" "main()" "{" VarDeclarations Statements "}" "}"
[4-5] ClassDeclarations -> ClassDeclaration ClassDeclarations | eps
[6] ClassDeclaration -> "class" Identifier Extension "{" FieldDeclarations MethodDeclarations "}"
[7-8] Extension -> "extends" Identifier | eps
[9-10] FieldDeclarations -> FieldDeclaration FieldDeclarations | eps
[11] FieldDeclaration -> "static" Type Identifier ";"
[12-13] VarDeclarations -> VarDeclaration VarDeclarations | eps
[14] VarDeclaration -> Type Identifier ";"
[15-16] MethodDeclarations -> MethodDeclaration MethodDeclarations | eps
[17] MethodDeclaration -> "public" "static" Type Identifier "(" Parameters ")" "{" VarDeclarations Statements "return" GenExpression ";" "}"
[18-19] Parameters -> Type Identifier Parameter | eps
[20-21] Parameter -> "," Type Identifier Parameter | eps
[22-23] Type -> "boolean" | "int"
[24-25] Statements -> Statements Statement | eps
[26-30] Statement -> "{" Statements "}" | "if" "(" GenExpression ")" Statement "else" Statement | "while" "(" GenExpression ")" Statement
| "System.out.println" "(" GenExpression ")" ";" | Identifier "=" GenExpression ";"
[31-32] GenExpression -> Expression | RelExpression
[33-35] Expression -> Expression "+" Term | Expression "-" Term | Term
[36-37] Term -> Term "*" Factor | Factor
[38-44] Factor -> "(" Expression ")" | Identifier | Identifier "." Identifier | Identifier "." Identifier "(" Arguments ")" | "true" | "false" | Integer
[45-46] RelExpression -> RelExpression "&&" RelTerm | RelTerm
[47-48] RelTerm -> Expression "==" Expression | Expression "<" Expression
[49-50] Arguments -> GenExpression Argument | eps
[51-52] Argument -> "," GenExpression Argument | eps
[53-54] Identifier -> Consonant Identifier| eps
[55-56] Integer -> Digit Integer | eps
[57-67] Digit -> ["0"-"9"]
[68-120] Consonant -> ["a-"Z"]
```

شکل ۱ـ۲: گرامر bnf

در ادامه با استفاده از این گرامر سعی شده است که یک یا چند کد تولید شود که تا حد ممکن از تمامی قوانین در مجموع کدهای تولیدی استفاده شده باشد. پکیج‌هایی نظیر nltk وجود دارند که می‌توان گرامر را با همین فرمت دریافت کنند و از قوانین آن رشته‌های مختلف با عمق متفاوت بسازند، اما از آنجا که این پکیج‌ها عموما محدودیت‌هایی دارند که ملزم به cfg بودن گرامر و نداشتن recursion می‌کند به همین منظور ترجیح داده شد با توجه به این نوع محدودیت‌های بیان شده در بخش ۱ـ۲ مورد آزمون به صورت دستی نوشته شود، در نهایت یک مورد آزمون نوشته شد که از تمامی قوانین در آن استفاده شده است، در بسط دادن تا جای ممکن از اشتقاق سمت چپ استفاده شده است. بدلیل طولانی بودن بسط دادن تنها چند مرحله اول و خروجی نهایی نشان داده شده است.

شماره قانون	رشته بسط داده شده توسط قانون
	Goal
1	Source EOF
2	ClassDeclarations MainClass
4	ClassDeclaration ClassDeclarations
6	class Identifier Extension { FieldDeclarations MethodDeclarations }
53,54,68-94	class abcdefghijklmnopqrstuvwxyz Extension { FieldDeclarations MethodDeclarations }
...	....

شکل ۱ـ۳: بسط گرامر توسط قوانین

۱ـ۶ ابزارهای مورد استفاده جهت تست

- Junit: برای اجرای تست‌ها
- PIT : برای ایجاد موتاسیون و اندازه‌گیری پوشش

```
class abcdefghijklmnopqrstuvwxyz {}

class ABCDEFGHIJKLMNOPQRSTUVWXYZ extends abcdefghijklmnopqrstuvwxyz {
    public static boolean diffComparison ( int x , int y )  {
        boolean ans ;
        if ( x == y && x < 0)
            ans = true ;
        else {
            while ( x < 0 )
                x = x + 1 ;
            ans = false ;
        }
        return ans ;
    }

    public static boolean check ( ) {
        boolean b ;
        b = ABCDEFGHIJKLMNOPQRSTUVWXYZ . diffComparison ( 8 , 7 ) ;
        return b ;
    }
}
class myClIn {
    static int a ;
    static int b ;
    public static int methodIn ( int x , int y ) {
        return myClIn . a - myClIn . b + ( 2 * x ) + y;
    }

}

public class myMain {
    public static void main() {
        boolean b;
        int i;

        b = false;
        i = ( 12 - 34) + ( 56 * 68 ) - 90 + 1 * 0 ;

        i = myClIn . methodIn ( 9 , 0 ) ;
        b = ABCDEFGHIJKLMNOPQRSTUVWXYZ . check ( ) ;
        System.out.println ( 0 ) ;
    }
}
```

شکل ۱ـ۴: خروجی بسط گرامر با تریگر کردن تمام قانون ها حداقل یکبار

۱ـ۷ مراحل مختلف آزمون

همانگونه که در بخش ۱ـ۳ گفتیم از روش گرامر برای تولید آزمون استفاده خواهیم کرد و همانطور که در بخش ۱ـ۴ گفتیم پوشش مد نظر پوشش production خواهد بود که از جمله پوشش‌های قوی برای گرامرهاست و همانگونه که در بخش ۱ـ۵ گفتیم مورد آزمون همانند شکل ۱ـ۳ بسط می‌یابد و از هر قانون حداقل یکبار استفاده می‌شود و خروجی نهایی آن بدست می‌اید که در شکل ۱ـ۴ نمایش داده شده است. نهایتا آزمون با استفاده از ابزار Junit اجرا می‌شود و diff خروجی و خروجی مورد انتظار گرفته و بررسی می‌شود. همچنین از ابزار PIT برای ایجاد موتاسیون استفاده می‌شود تا مشخص شود که چه قسمت‌هایی از کد پوشش داده شده است و تست طراحی شده چقدر مناسب است.

۱ـ۷ـ۱ نیازمندی‌های آزمون

با توجه به انتخاب پوشش production نیازمندی‌های آزمون P۱ تا P۱۲۰ می‌باشد که متناظر هر یک از قوانین داخل گرامر است.

۱ـ۷ـ۲ مورد آزمون

مورد آزمون رشته تولید شده توسط گرامر و پوشش production است که در شکل ۱ـ۴ نمایش داده شده است.

در طراحی مورد آزمون باید خروجی مورد انتظار نیز نمایش داده شود که خروجی مورد انتظار برای این رشته در شکل ۱ـ۵ نمایش داده شده است.

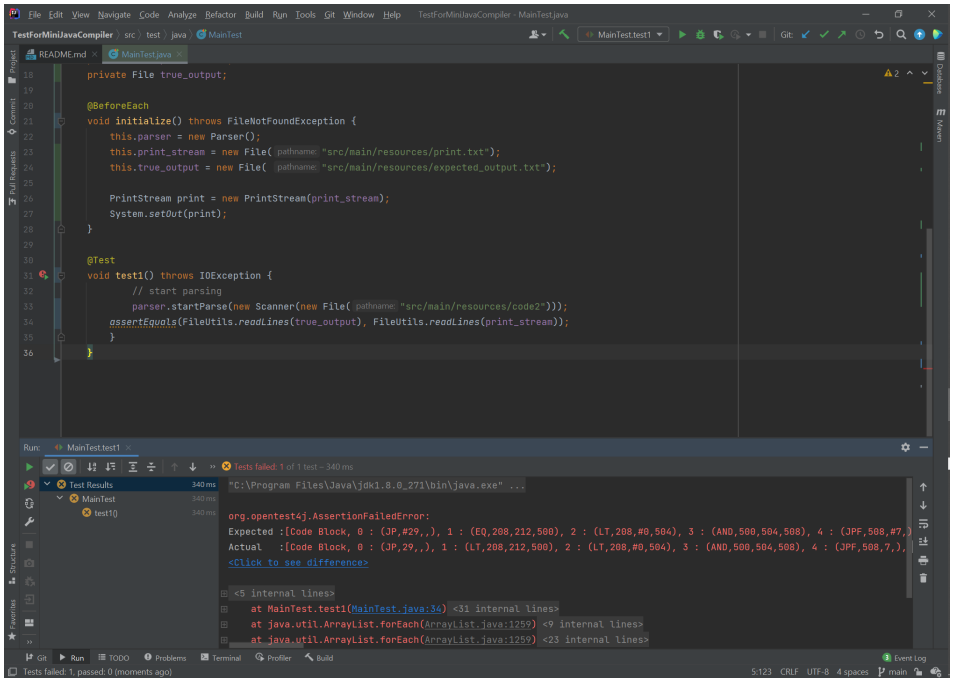
```
Code Block
0 : (JP,#29,,)
1 : (EQ,208,212,500)
2 : (LT,208,#0,504)
3 : (AND,500,504,508)
4 : (JPF,508,#7,)
5 : (ASSIGN,#1,216,)
6 : (JP,#13,,)
7 : (LT,208,#0,512)
8 : (JPF,512,#12,)
9 : (ADD,208,#1,516)
10 : (ASSIGN,516,208,)
11 : (JP,#7,,)
12 : (ASSIGN,#0,216,)
13 : (ASSIGN,216,@200,)
14 : (JP,#204,,)
15 : (ASSIGN,#8,208,)
16 : (ASSIGN,#7,212,)
17 : (ASSIGN,#520,200,)
18 : (ASSIGN,#20,204,)
19 : (JP,#1,,)
20 : (ASSIGN,520,228,)
21 : (ASSIGN,228,@220,)
22 : (JP,#224,,)
23 : (SUB,232,236,524)
24 : (MULT,#2,248,528)
25 : (ADD,524,528,532)
26 : (MULT,532,252,536)
27 : (ASSIGN,536,@240,)
28 : (JP,#244,,)
29 : (ASSIGN,#0,264,)
30 : (SUB,#12,#34,540)
31 : (MULT,#56,#68,544)
32 : (ADD,540,544,548)
33 : (SUB,548,#90,552)
34 : (ADD,552,#1,556)
35 : (MULT,556,#0,560)
36 : (ASSIGN,560,268,)
37 : (ASSIGN,#9,248,)
38 : (ASSIGN,#0,252,)
39 : (ASSIGN,#564,240,)
40 : (ASSIGN,#42,244,)
41 : (JP,#23,,)
42 : (ASSIGN,564,268,)
43 : (ASSIGN,#568,220,)
44 : (ASSIGN,#46,224,)
45 : (JP,#15,,)
46 : (ASSIGN,568,264,)
47 : (PRINT,#0,,)
```

شکل ۱ـ۵: خروجی مورد انتظار مورد آزمون طراحی شده

۱ـ۷ـ۳ انجام آزمون Junit:

با استفاده از junit و assertion مساوی خروجی کد رو ابتدا در فایل print.txt ریخته و سپس با آنچه که به عنوان خروجی موردنظر طراحی شده است مقایسه می‌کند. همانگونه که در شکل ۱ـ۶ مشاهده می‌شود، تست انجام شده و جواب خروجی و جواب مدنظر مقایسه شده و خطا raise شده است، در کنسول خطا تصویر ۱ـ۶ کاملا مشخص است.

اگر دو فایل را با یکدیگر مقایسه کنیم متوجه خطاها به صورت خط به خط می‌شویم، مقایسه این دو را در



شکل ۱ـ۶: انجام آزمون

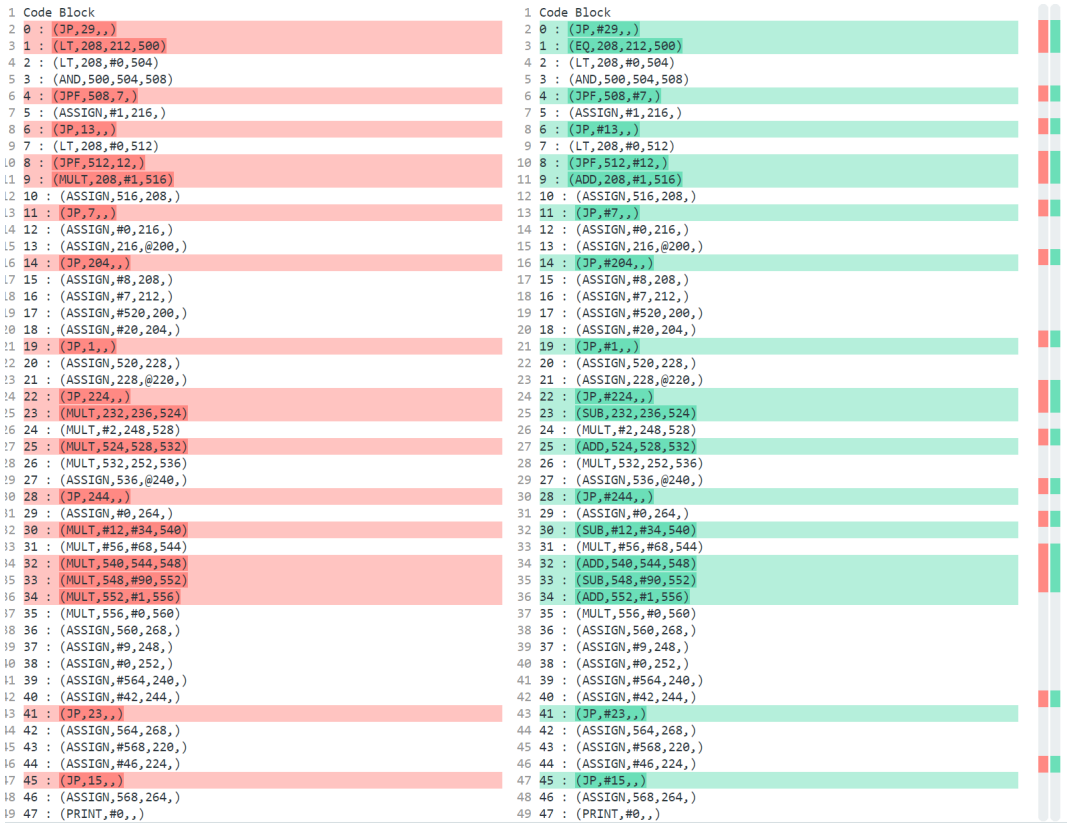
شکل ۱ـ۷ مشاهده می‌کنید.

:PIT

بدون رفع کردن خطاهای موجود در برنامه از ابزار PIT برای آزمون موتاسیون استفاده می‌کنیم. این ابزار نیاز دارد که یک سری mutator را به عنوان کانفیگ دریافت کند، که تعدادی پیش‌فرض آن را انتخاب کردیم که فعال‌های آن در این پروژه در شکل ۱ـ۱۰ مشاهده می‌کنید، همچنین کلاس‌های هدف رو نیز تعیین کرد که در شکل ۱ـ۸ کلاس‌های هدف تعیین شده را مشاهده می‌کنید.

توسط این ابزار می‌توان ریپورت جامعی را با پسوند html تولید کرد که نشان می‌دهد چه تعداد موتاسیون تولید شده و چه تعداد از آن‌ها توسط تست پیشنهاد شده پوشش داده شده است. در شکل زیر نمای کلی این گزارش را مشاهده می‌کنید.

در برخی از موارد پوشش کامل نیست و به این منظور به داخل ریپورت‌ها مراجعه می‌کنیم و خطوط قرمز رنگ را بررسی و وضعیت انها نسبت به خطوط سبز را می‌سنجیم. ماژول errorHandler که طبیعی است چرا کمترین پوشش را دارد، چون که در ازای اجرای این تست ما به خطا نمی‌خوریم و بیشتر خطوط این ماژول برای مدیریت خطاهاست. همچنین وقتی به داخل ماژول semantic.symbol می‌رویم تنها خطوط قرمز مربوط به مدیریت خطاهای مربوط به آن مانند خطاهای مروبط به دوباره تعریف کردن نام یک کلاس متد یا متغیر است و بقیه خطوط پوشش داده شده است. در parser هم که فقط فایل Action.java کامل پوشش داده نشده که



```
Code Block
0 : (JP,29,,)
1 : (LT,208,212,500)
2 : (LT,208,#0,504)
3 : (AND,500,504,508)
4 : (JPF,508,7,)
5 : (ASSIGN,#1,216,)
6 : (JP,13,,)
7 : (LT,208,#0,512)
8 : (JPF,512,12,)
9 : (MULT,208,#1,516)
10 : (ASSIGN,516,208,)
11 : (JP,7,,)
12 : (ASSIGN,#0,216,)
13 : (ASSIGN,216,@200,)
14 : (JP,204,,)
15 : (ASSIGN,#8,208,)
16 : (ASSIGN,#7,212,)
17 : (ASSIGN,#520,200,)
18 : (ASSIGN,#20,204,)
19 : (JP,1,,)
20 : (ASSIGN,520,228,)
21 : (ASSIGN,228,@220,)
22 : (JP,224,,)
23 : (MULT,232,236,524)
24 : (MULT,#2,248,528)
25 : (MULT,524,528,532)
26 : (MULT,532,252,536)
27 : (ASSIGN,536,@240,)
28 : (JP,244,,)
29 : (ASSIGN,#0,264,)
30 : (MULT,#12,#34,540)
31 : (MULT,#56,#68,544)
32 : (MULT,540,544,548)
33 : (MULT,548,#90,552)
34 : (MULT,552,#1,556)
35 : (MULT,556,#0,560)
36 : (ASSIGN,560,268,)
37 : (ASSIGN,#9,248,)
38 : (ASSIGN,#0,252,)
39 : (ASSIGN,#564,240,)
40 : (ASSIGN,#42,244,)
41 : (JP,23,,)
42 : (ASSIGN,564,268,)
43 : (ASSIGN,#568,220,)
44 : (ASSIGN,#46,224,)
45 : (JP,15,,)
46 : (ASSIGN,568,264,)
47 : (PRINT,#0,,)
```

```
Code Block
0 : (JP,#29,,)
1 : (EQ,208,212,500)
2 : (LT,208,#0,504)
3 : (AND,500,504,508)
4 : (JPF,508,#7,)
5 : (ASSIGN,#1,216,)
6 : (JP,#13,,)
7 : (LT,208,#0,512)
8 : (JPF,512,#12,)
9 : (ADD,208,#1,516)
10 : (ASSIGN,516,208,)
11 : (JP,#7,,)
12 : (ASSIGN,#0,216,)
13 : (ASSIGN,216,@200,)
14 : (JP,#204,,)
15 : (ASSIGN,#8,208,)
16 : (ASSIGN,#7,212,)
17 : (ASSIGN,#520,200,)
18 : (ASSIGN,#20,204,)
19 : (JP,#1,,)
20 : (ASSIGN,520,228,)
21 : (ASSIGN,228,@220,)
22 : (JP,#224,,)
23 : (SUB,232,236,524)
24 : (MULT,#2,248,528)
25 : (ADD,524,528,532)
26 : (MULT,532,252,536)
27 : (ASSIGN,536,@240,)
28 : (JP,#244,,)
29 : (ASSIGN,#0,264,)
30 : (SUB,#12,#34,540)
31 : (MULT,#56,#68,544)
32 : (ADD,540,544,548)
33 : (SUB,548,#90,552)
34 : (ADD,552,#1,556)
35 : (MULT,556,#0,560)
36 : (ASSIGN,560,268,)
37 : (ASSIGN,#9,248,)
38 : (ASSIGN,#0,252,)
39 : (ASSIGN,#564,240,)
40 : (ASSIGN,#42,244,)
41 : (JP,#23,,)
42 : (ASSIGN,564,268,)
43 : (ASSIGN,#568,220,)
44 : (ASSIGN,#46,224,)
45 : (JP,#15,,)
46 : (ASSIGN,568,264,)
47 : (PRINT,#0,,)
```

شکل ۱ـ۷: مقایسه فایل خروجی و فایل مورد انتظار

یه switch بوده که مشخصا خطا از آنجا نیز نیست. تنها کلاس باقی مانده CodeGenerator است که وقتی داخل آن می رویم به وضوح دیده می‌شود که در زیر فایل codegenerator.java یک سری از توابع که مربوط به کدهای میانی ADD ، SUB و EQ است که هیچ وقت صدا زده نشده‌اند حال آن که در مورد ازمون این موارد وجود داشتند که مشخصا نشان می‌دهد این کلاس به درستی کار نمی‌کند، جزییات را در شکل ۱ـ۱۱ مشاهده می‌کنید.

۱ـ۸ نتیجه‌گیری

همانگونه که گفته شد مقایسه دو خروجی مورد آزمون و خروجی مورد انتظار در شکل ۱ـ۷ انجام شده است. از مقایسه چندین تفاوت بین خطوط انها متوجه می‌شویم که خروجی کامپایلر در این خطوط با یک دیگر فرق دارد:

- به جای EQ خروجی برابر LT می‌گذارد.
- به جای ADD و SUB خروجی برابر MULT می‌گذارد.
- در پرش‌ها علامت شارپ را فراموش می‌کند.

```
<plugin>
    <groupId>org.pitest</groupId>
    <artifactId>pitest-maven</artifactId>
    <version>1.6.7</version>
    <configuration>
        <targetClasses>
            <param>codeGenerator.CodeGenerator</param>
            <param>codeGenerator.Address</param>
            <param>codeGenerator.Memory</param>
            <param>codeGenerator._3AddressCode</param>
            <param>codeGenerator.Operation</param>
            <param>codeGenerator.TypeAddress</param>
            <param>codeGenerator.varType</param>
            <param>errorHandler.ErrorHandler</param>
            <param>semantic.symbol.Symbol</param>
            <param>semantic.symbol.SymbolTable</param>
            <param>parser.Action</param>
            <param>parser.Rule</param>
            <param>parser.GrammarSymbol</param>
        </targetClasses>
        <targetTests>
            <param>AmirTestMiniJava</param>
        </targetTests>
    </configuration>
</plugin>
```

شکل ۱ـ۸: کلاس‌های تعیین شده در PIT

Active mutators

- BOOLEAN_FALSE_RETURN
- BOOLEAN_TRUE_RETURN
- CONDITIONALS_BOUNDARY_MUTATOR
- EMPTY_RETURN_VALUES
- INCREMENTS_MUTATOR
- INVERT_NEGS_MUTATOR
- MATH_MUTATOR
- NEGATE_CONDITIONALS_MUTATOR
- NULL_RETURN_VALUES
- PRIMITIVE_RETURN_VALS_MUTATOR
- VOID_METHOD_CALL_MUTATOR

شکل ۱ـ۹: mutator های فعال

همچنین با استفاده از آزمون موتاسیون دیدیم که توابع متناظر همین خطاها در فایل codegenerator.java اصلا اجرا نمی‌شوند و پوشش داده نمی‌شوند.

برای انجام کارهای بیشتر بهتر بود که خطاها درگام اول که شناخته می‌شدند برطرف می‌شدند و سپس آزمون موتاسیون اجرا می‌گردید و باز بررسی می‌گشت و دوباره به طراحی موارد آزمون می‌پرداختیم تا mutant بیشتری kill شود و با این طراحی مجددا خطاها را میافتیم و باز رفع می‌کردیم . دوباره آزمون موتاسیون را با تعداد بیشتری mutant اجرا می‌کردیم و این روند را به صورت چرخه تکرار می‌کردیم تا تعداد mutant هایی که kill می‌شوند افزایش یابد و اگر نسبت آن دیگر ثابت ماند پروسه رو متوقف می کردیم. با توجه به انتخاب این روش و عدم موجود بودن هر یک از توصیف تابع‌ها تصحیح خطاها سخت می‌انجامید. لذا به پیدا کردن این

Pit Test Coverage Report

Project Summary

Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
7	83%	369/442	73%	111/153	97%	111/115

Breakdown by Package

Name	Number of Classes	Line Coverage		Mutation Coverage		Test Strength	
codeGenerator	3	84%	302/360	76%	93/123	99%	93/94
errorHandler	1	8%	1/12	0%	0/5	0%	0/0
parser	2	97%	28/29	50%	4/8	57%	4/7
semantic.symbol	1	93%	38/41	82%	14/17	100%	14/14

Report generated by PIT 1.6.7

شکل ۱ـ۱۰: آزمون موتاسیون

```
            case 9:
1               assign();
                break;
            case 10:
1               add();
                break;
            case 11:
1               sub();
                break;
            case 12:
1               mult();
                break;
            case 13:
1               label();
                break;
            case 14:
1               save();
                break;
            case 15:
1               _while();
                break;
            case 16:
1               jpf_save();
                break;
            case 17:
1               jpHere();
                break;
            case 18:
1               print();
                break;
            case 19:
1               equal();
                break;
            case 20:
1               less_than();
                break;
            case 21:
1               and();
                break;
            case 22:
1               not();
                break;
            case 23:
```

شکل ۱ـ۱۱: آزمون موتاسیون فایل codegenerator.java

خطاها تا همین جا با استفاده از گرامر و پوشش production و انجام یک تست جامع با استفاده از Junit و PIT و گزارش آن بسنده کردیم.