Лицей «Физико-техническая школа»
Санкт-Петербургского Академического университета им. Ж. И. Алферова

Отчет о практике

Миграция кода на *Java*, использующего *int*, на *BigInteger*

Работу выполнил:
ученик 11а класса Сапожников Аркадий

Научный руководитель:
Суворов Егор Федорович

Место прохождения практики:
НИУ ВШЭ - Санкт-Петербург

Санкт-Петербург
2020

Аннотация

Моя задача – для данного кода, с использованием переменных типа *int*, заменить данные переменные и все операции, с использованием этих переменных, на вид *BigInteger*. Оба этих типа позволяют пользователю хранить числовые переменные, то есть компилятор, к котором написана программа запрашивает у компьютера “ячейку” для этого числа. Отличие *int* от *BigInteger* – размер ячейки, (то есть количество отведенной памяти) а также внешний вид: *BigInteger* – это класс, в отличие от int, поэтому одни и те же операции будут записаны по-разному.

Содержание

Аннотация **2**

Содержание **3**

Введение **4**

Постановка задачи **4**

Методика **5**

Java-parser и AST 5

Разбор AST 5

Начальная вершина 5

Методы в AST и Statements 6

Expressions 7

Итог строения AST 8

Общий план 8

Общая схема преобразования 9

Visit 9

Поиск 10

Замена 10

Проблема с выражением 10

Функция intValue 10

Алгоритм замены 11

Метод updateIntsToBigInt 11

Метод isUpdateIntsToBigInt 11

Массивы типа BigInteger 12

Инструментарий для разработки 12

GitHub и git 12

Travis 12

Результаты **13**

Примеры кода, с которыми работает замена 14

Выводы **15**

Список литературы **15**

Введение

Спортивное и олимпиадное программирование - очень популярные соревнования в наше время. Как правило, некоторые этапы проходят заочно, то есть у участника есть возможность пользоваться всем: своими программами, интернетом или книгами.
Составители задач иногда устанавливают такие ограничения на числа такие, чтобы в использовании переменных типа *int* ($|int| < 2^{31} - 1$), а иногда и *long* ($|long| < 2^{63} - 1$) может не хватить допустимого размера. В таком случае *Java* - программистам необходимо использовать класс, который называется *BigInteger*.
Код с использованием *BigInteger* визуально очень сильно отличается от кода, в котором присутствуют только примитивные типы, так как, например, арифметические операторы +, -... в классе реализованы, как методы:

int	**BigInteger**
`a + b`	`a.add(b);`
`a - b`	`a.subtract(b);`
`a * b`	`a.multiply(b);`
`a / b`	`a.divide(b);`
`a % b`	`a.remainder(b);`

А если выражение будет какое-нибудь сложное, то код с *BigInteger* сложно писать и читать:

int	**BigInteger**
`a + (b * c) - a * (b - c) - (a + b / c) / (b + c % d)`	`a.add((b.multiply(c))).subtract(a.multiply((b.subtract(c)))).subtract((a.add(b.divide(c))).divide((b.add(c.remainder(d)))));`

Чтобы упростить себе задачу, программист может написать более простой код, после чего преобразовать его с помощью моего приложения и получить рабочий код. Также часто проблема с размерами чисел осознается в течение или уже после написания программы, например, при умножении или возведении в степень, и тогда эту проблему можно быстро решить, используя мою программу.

Постановка задачи

Задача - написать приложение, которое автоматически преобразовывает *Java*-код использующего переменные типа *int* на аналогичный код с использованием *BigInteger*.
Основной спектр принимаемых кодов - решения олимпиадных задач, например: на сайте *Codeforces* (https://codeforces.com/). Необязательно поддерживать вообще все случаи, необходимо уметь превращать только основные, такие как, например: арифметические операции, массивы, функции, циклы, ифы...

Методика

Java-parser и AST

Java-parser - библиотека, с помощью которого можно получить *Abstract Syntax Tree (AST)* из *Java*-кода:

```
CompilationUnit compilationUnit = StaticJavaParser.parse("class A {
                                    }");
```

AST - визуализация написанного кода, который можно просматривать, анализировать, частично удалять и как-либо изменять его. Разберем пример *AST* алгоритма Евклида:

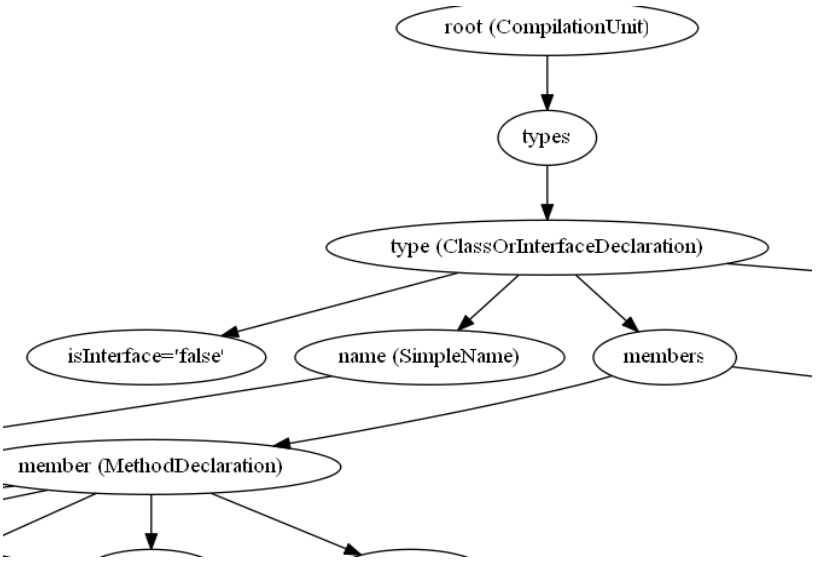
```
public class EuclideanAlgorithm {

    public static void main(String[] args) {
        System.out.println(gcd(a, b));
    }

    public static int gcd(int a, int b) {
        while (b != 0) {
            int tmp = a % b;
            a = b;
            b = tmp;
        }
        return a;
    }
}
```

Разбор AST

Рассмотрим *AST* этого кода по частям:

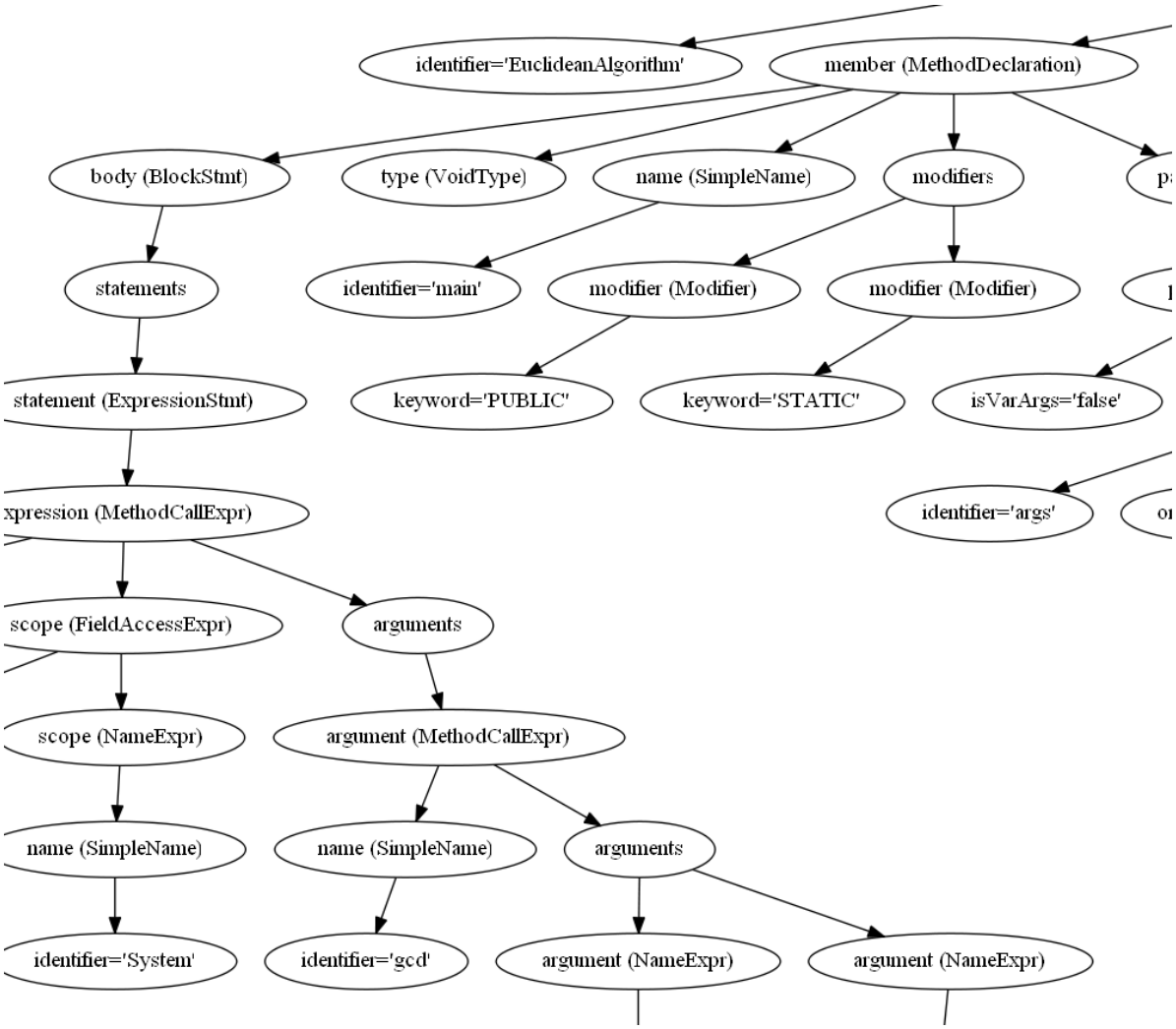
Начальная вершина

Построение дерева начинается с вершины *ComponetnUnit*, от которой происходят первые ответвления - классы.
На рисунке видно, что в коде один класс (*ClassOrInterfaceDeclaration*), у которого есть параметр имени: *EuclideanAlgorithm*; параметры, которые говорят, о том, что этот класс действительно является классом, то есть то, что он не *Interface*, и то, что он публичный; а также параметр *members* - методы этого класса, которые мы рассмотрим подробнее.



Методы в AST и Statements

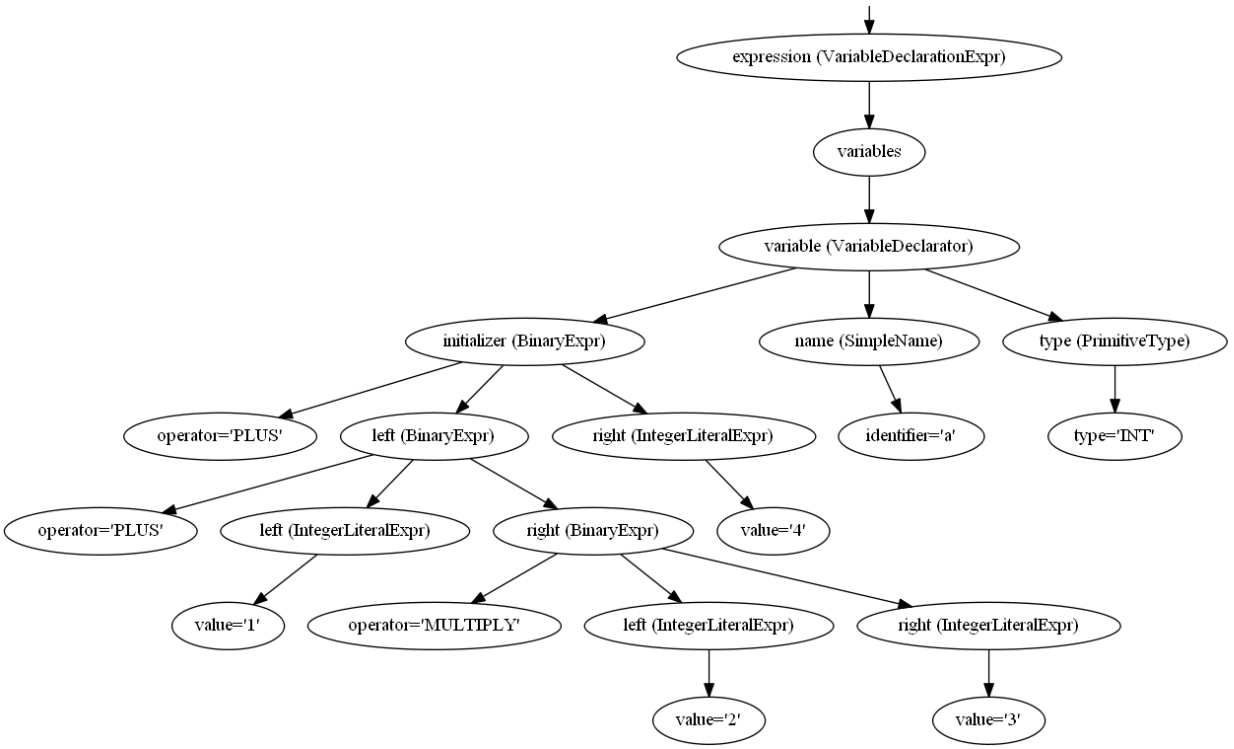
Из вершины выходит два ребра, которые соответствуют двум методам: *main* и *gcd*. Ответвления показывают свойства этих методов: публичность, статичность, тип, имя, список принимаемых параметров, а также *body*, в котором содержится весь код внутри рассматриваемого метода. Переход во внутреннюю часть метода осуществляется через вершину *BodyStmt*. *BodyStmt* - это *Statement*, *Java Interface,* который отображается в *AST*, перед началом какой-либо существенной части кода. Примеры таких частей: тело метода, тело цикла, тело *if* или *else*, содержание *return*, просто новая строчка кода...



На рисунке изображена часть *AST* метода *main*

Expressions

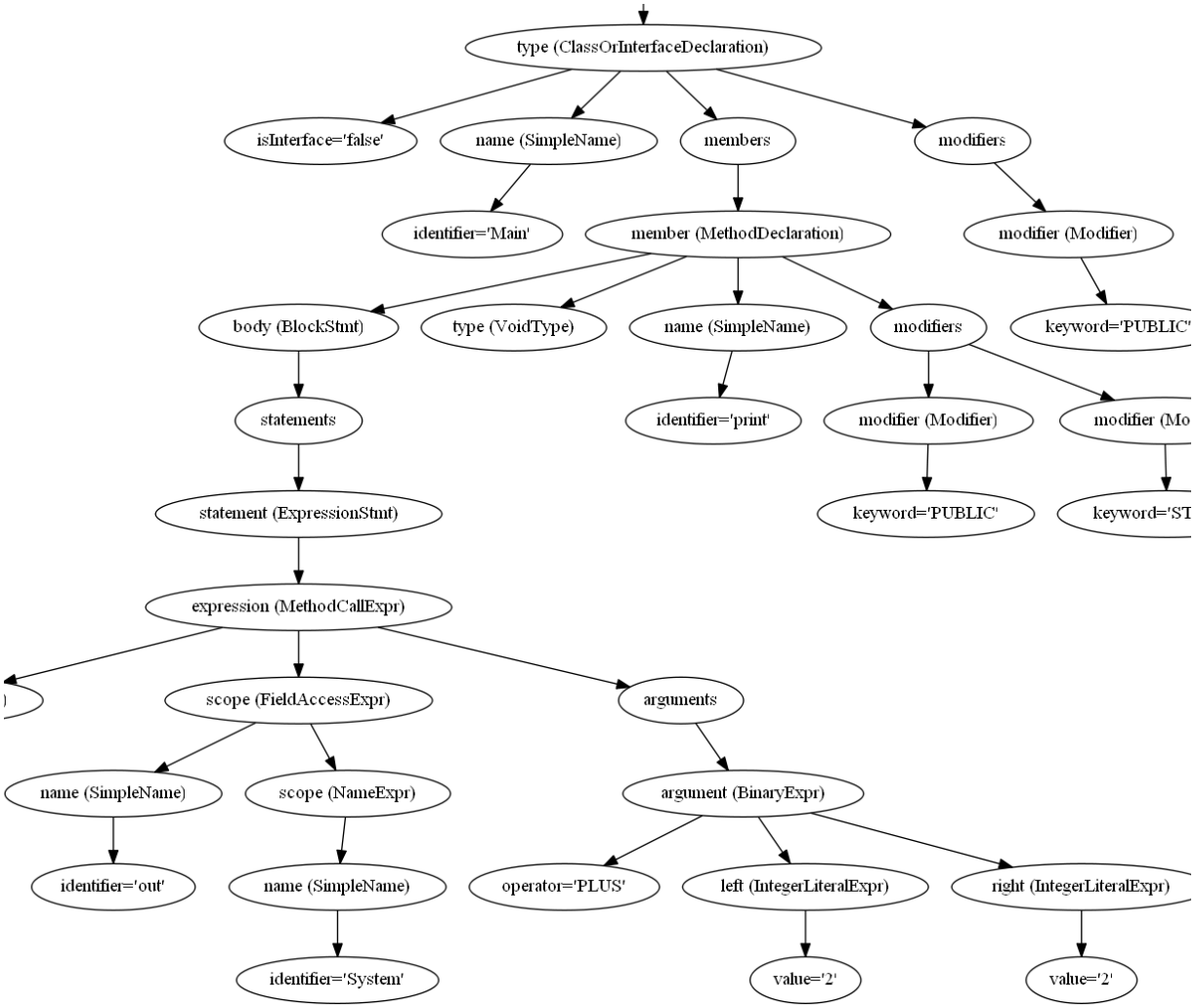
Можно заметить, что из *Statements* выходят различные *Expressions*, аналогия *Statements*, но для менее существенных вещей. Например: *VariableDeclarationExpr* (используемые в коде переменные), *BinaryExpr* (арифметическое действие), *IntegerLiteralExpr* (числа)...
У *Expressions* есть параметры, по которым можно все сказать про этот объект, напоминающие параметры методов.



На рисунке изображено часть *AST* кода *1 + 2 * 3 + 4*

Итог строения *AST*

Код преобразуется в *ComponentUnit*, из которого все начинается. Дальше идет разветвление на *ClassOrInterfaceDeclaration* (классы и интерфейсы), потом на *members* (методы), затем на *Statements* (глобальные части), а после этого на *Expressions* (небольшие части). Действительно, эти разветвления полностью соответствует логике кода.



На рисунке изображено часть *AST* кода *2 + 2*

Общий план

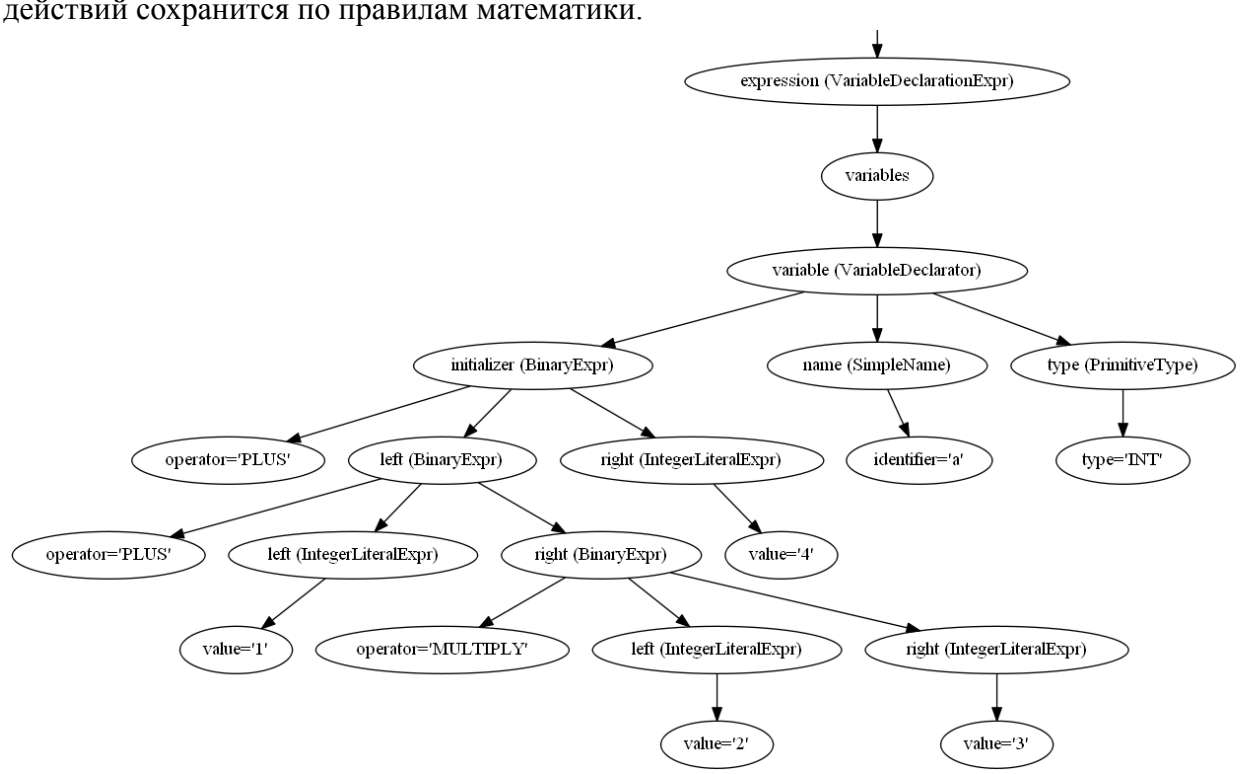
Моя программа принимает код пользователя, в котором нужные переменные отмечены комментарием: `/* BigInteger */`. Пример: `int /* BigInteger */ a = 1;`. После этого переводим полученный код в *AST*, пробегаемся по нему, находим нужные переменные и меняем их на *BigInteger*. Вместе с этими переменными мы вынуждены менять и соседние *Expressions*, чтобы код компилировался. Пример такого случая: *a + b*, где *a* - помеченная переменная. Замена такого выражения необходима, потому что нельзя складывать переменные типов *BigInteger* и *int* при помощи арифметических операторов.

Общая схема преобразования

Мое преобразование делится на две части: поиск нужных переменных и их замена.
У такого решения есть несколько проблем. Одна из них, одна из самый существенных, - проблема с заменой во время пробега. Переменные заменяются последовательно в моем алгоритме, поэтому будет какой-то момент, когда в том же примере *a* + *b*, где *a* и *b* - превращаемые переменные, будет заменена только одна и код перестанет быть компилируемым и дальнейшее перемещение по дереву будет невозможным, и будут выпадать ошибки из-за вызова функции *resolve*, которую я часто использую в решении. *Resolve* - функция, которая позволяет узнать информацию про то или иной выражение (например: его тип, связь с предыдущим…).
Решение этой проблемы очень простое: хранить все изменения в массиве *change* типа *Runnable*, встроенного класса, в котором только лишь один метод - *run*. (`changes.add(() -> replacing code);`). Проходя по *AST* дереву я пополняю массив *changes* новыми заменами. В конце алгоритма, когда мне уже не нужно проходить по дереву, и промежуточная компилируемость уже не так важна, я пробегаюсь по всему массиву *changes*, и запускаю *run*.

Visit

Для удобного перемещения по дереву следует использовать методы *visit*, которые вызываются автоматически, в зависимости от передаваемого *Expression*, *Statements*… (любой вершины *AST*) этому *visit*. То есть если написать метод *visit*, который принимает *VariableExpr*, то этот метод вызовется для всех *VariableExpr* в исходном коде, более того, если указать вначале "*super.visit"*, то тогда *Expression* будут перечисляться снизу вверх, то есть более глубокая вершина в графе вызовется первой. Это бывает очень удобно для арифметики, так как ее можно заменять последовательно и быть уверенным, что порядок арифметических действий сохранится по правилам математики.



В этом примере *visit* будет проходить снизу вверх, сначала заменится умножение, а потом суммы. Дерево строится согласно правилам выполнения операций.

Поиск

Поиск происходит довольно очевидно: пробег по *visits* переменных: *VariableExpr* (обычные), *ArrayCreationExpr* (массивы) и *MethodDeclaration* (методы). Сохраняем *Range* (координаты переменных в коде, то есть номер строки и столбца в файле кода, которые можно узнать с помощью *getRange()*) этих переменных в *HashMap variablesToReplace*. *Range* хранить надежнее и удобнее всего, потому что это свойство точно не повторяется у разных объектов, а имена, например, могут повторятся; и есть у всех объектов. *HashMap* удобнее использовать, так как есть встроенная функция *contains*, которая быстро и в одну строчку позволяет узнать принадлежность списку.

Замена

Замена происходит чуть сложнее, чем поиск. Количество визитов должно быть сильно больше, так как теперь цель - найти все места, в которых присутствуют наши заменяемые переменные, и теперь менять придется большие куски кода, чтобы сохранялось компилируемость. Основной пример - арифметика.

Проблема с выражением

Если хотя бы одна переменная должна быть заменена, то придется заменять все выражение. Это выражение может быть почти где угодно:

1. в присваивании (`int c = a + b`)
2. в аргументе какого-либо метода (`foo(a + b)`)
3. в теле *return* (`return a + b`)

Если в каком-либо выражении есть различный виды *int* (те, которые надо менять и те, которые менять не надо), то тогда у незаменяемых переменных нужно вызвать встроенную функцию *valueOf*, которая превращает *int* в *BigInteger* в выражениях.
Пример:

до	**после**

`int /* BigInteger */ a = 1;`
`int b = 2;`
`System.out.print(a + b);`
	`java.math.BigInteger a =`
	`java.math.BigInteger.ONE;`
	`int b = 2;`
	`a.add(java.math.BigInteger.valueOf(b));`

Функция *intValue*

Также есть несколько ограничений на замены, к примеру: индексы у массивов (они всегда должны быть типа *int*), разные типы *int* у аргумента и параметра метода или при присваивании.... Все эти проблемы решаются с помощью *valueOf* и *intValue* (противоположность *valueOf* - превращает *BigInteger* в *int*).
Проблема с массивом решается переводом всего выражения, после чего вызова функции *intValue*, чтобы не возникало ошибок про некорректный тип длины или индекса.
Чтобы не было проблем из-за разного типа параметра и аргумента, необходимо, аналогично индексу массива, преобразовать аргумент, если, параметр не нужно менять.

Алгоритм замены

Я создаю много методов *visit*, на все основные *Expressions* и *Statements*, в теле которых смотрю на содержимое этих объектов. Изменение кода будет заметно отличаться в зависимости от *Expressions.* Например: если происходит замена переменной, то тогда нужно просто поменять тип переменной. Если это число *0*, *1*, *2* или *10*, то я использую встроенные в класс *BigInteger* статичные числа (`java.math.BigInteger.ZERO`, `java.math.BigInteger.ONE`…), иначе вызываю функцию *valueOf*, если выражение простое.

Метод `updateIntsToBigInt`

К сожалению, часто выражение сложное, в таком случае я вызываю метод `updateIntsToBigInt`. Этот метод принимает *Expression*, который нужно поменять в любом случае. Сначала идет множество проверок на принадлежность какому-либо *Expression*, где для каждого случая идет своя обработка: раскручивание *Expression*, то есть рекурсивно обрабатываем *Expressions*, выходящие из данного.
Например:

1) если *BinaryExpr*, то вызываем `updateIntsToBigInt` от правой и левой части;
2) если *EnclosedExpr*, то есть выражение внутри круглых скобках в коде пользователя, то вызываем этот метод для выражения внутри;
3) если *UnaryExpr*, то есть выражение вида оператор и внутреннее выражение (например: *-1, +5, -(a + b)*), то вызываем `updateIntsToBigInt`, от выражения после оператора, и если вначале был знак -, то добавляем *negate* к преобразованному выражению;...

Если пришли к концу дерева, то добавляем замену в *changes*. Если передаваемый *Expression* тот, которые алгоритм не проверяет, то тогда вызываем функцию *valueOf*.

Метод `isUpdateIntsToBigInt`

Перед тем, как вызывать метод `updateIntsToBigInt`, необходимо убедиться, что менять это выражение действительно нужно. Для этого я написал метод `isUpdateIntsToBigInt`, который работает так же рекурсивно, как и метод `updateIntsToBigInt`, только вместо замен делает проверку, что переменная из помеченных пользователем, то есть такая, что ее *Range* принадлежит *variablesToReplace*; возвращаем информацию, о том, что выражение нужно заменить.

Массивы типа *BigInteger*

При замене *int* массивов на *BigInteger*, возникает небольшая проблема: *int* массивы, в отличие от *BigInteger* изначально заполнен нулями, чем программисты и пользуются. Я вынужден добавлять циклы в код, который заполняет все элементы нулями сразу после создания массива:

```
java.math.BigInteger[][] a = new java.math.BigInteger[b][c];
for (int aFilling1 = 0; aFilling1 < b; aFilling1++) {
     for (int aFilling2 = 0; aFilling2 < c; aFilling2++) {
          a[aFilling1][aFilling2] = java.math.BigInteger.ZERO;
     }
}
```

Инструментарий для разработки

GitHub и *git*

Я храню свой проект на *GitHub* (https://github.com/), самом популярном сайте для хранения программистами своих проектов. *GitHub* основан на *git*, системе управления проектом. Он позволяет разветвления по веткам, где каждая ветка решает какую-либо задачу или часть задачи. Это удобно для тестирования и дебага кода. После решения задачи, ветку можно сложить друг с другом (*git merge)*, превратив в код, который решает уже две задачи. Чтобы проверять корректность своего кода, я для каждой подзадачи писал свои тесты (например: тесты с арифметикой, массивами, циклами, минимумами, максимумами,...), которые запускала *travis* сразу после того, как я отправляю свой код на *GitHub*.

Travis

Travis - система автоматических тестов, которые можно подключить к *GitHub*, она запускает все тесты, находящиеся в проекте (названия которых заканчиваются на “*test*”). У каждой ветки есть текущий вердикт: галочка или крестик, что означает, готова ли это ветка к *merge*. Если тесты не проходят, то делать *merge* не стоит, потому что по предположению тесты в основной ветке (в ветке текущей версии основной задачи, которая по умолчанию называется *master*) должны выполняться.

Результаты

Я написал приложение, которое преобразовывает несложный *Java*-код использующего переменные типа *int* на аналогичный код с использованием *BigInteger*.
Я отправил несколько задач на сайт *Codeforces* (https://codeforces.com/), некоторые не прошли по времени, так как *BigInteger* работает сильно медленнее, некоторые как работали, так и продолжают работать, а некоторые наоборот начали проходить все тесты, чего я и добивался, так как *BigInteger* позволяет оперировать с большими числами. Ссылки на посылки решений:

Код задачи	Задача	Исходная посылка	Вердикт до	С аннотациями	После изменения	Вердикт после
1A	https://codeforces.com/contest/1/problem/A	https://codeforces.com/contest/1/submission/66223527	*WA9*	https://codeforces.com/contest/1/submission/67466330	https://codeforces.com/contest/1/submission/67466388	*OK*
2B	https://codeforces.com/contest/2/problem/B	https://codeforces.com/contest/2/submission/63283489	*OK*	https://codeforces.com/contest/2/submission/66887144	https://codeforces.com/contest/2/submission/66887134	*TL16*
6A	https://codeforces.com/contest/6/problem/A	https://codeforces.com/contest/6/submission/66383423	*OK*	https://codeforces.com/contest/6/submission/67466669	https://codeforces.com/contest/6/submission/67466691	*OK*
9D	https://codeforces.com/contest/9/problem/D	https://codeforces.com/contest/9/submission/10138795	*WA6*	https://codeforces.com/contest/9/submission/67479702	https://codeforces.com/contest/9/submission/67479756	*OK*
521A	https://codeforces.com/contest/521/problem/A	https://codeforces.com/contest/521/submission/57404094	*OK*	https://codeforces.com/contest/521/submission/67480457	https://codeforces.com/contest/521/submission/67480499	*OK*
631B	https://codeforces.com/contest/631/problem/B	https://codeforces.com/contest/631/submission/61297692	*OK*	https://codeforces.com/contest/631/submission/67481062	https://codeforces.com/contest/631/submission/67481147	*OK*
886B	https://codeforces.com/contest/886/problem/B	https://codeforces.com/contest/886/submission/32658722	*OK*	https://codeforces.com/contest/886/submission/67482994	https://codeforces.com/contest/886/submission/67483052	*OK*
929B	https://codeforces.com/contest/929/problem/B	https://codeforces.com/contest/929/submission/35860660	*OK*	https://codeforces.com/contest/929/submission/67484780	https://codeforces.com/contest/929/submission/67484913	*OK*
1011A	https://codeforces.com/contest/1011/problem/A	https://codeforces.com/contest/1011/submission/40788395	*OK*	https://codeforces.com/contest/1011/submission/67490212	https://codeforces.com/contest/1011/submission/67490219	*OK*

Примеры частей кода, с которыми работает замена

I пример:

До:

```
int /* BigInteger */ a[]=new int[4];
for(int /* BigInteger */ i=0;i<4;i++) {
    a[i]=scan.nextInt();
}
```

После:

```
java.math.BigInteger[] a = new java.math.BigInteger[4];
for (int aFilling1 = 0; aFilling1 < 4; aFilling1++) {
    a[aFilling1] = java.math.BigInteger.ZERO;
}
for (java.math.BigInteger i = java.math.BigInteger.ZERO;
i.compareTo(java.math.BigInteger.valueOf(4)) < 0; i =
i.add(java.math.BigInteger.ONE)) {
    a[i.intValue()] = scan.nextBigInteger();
}
```

II пример:

До:

```
for (int /* BigInteger */ j = 0; j < 200; j++) {
    System.out.println(-1);
}
int j = 0;
while(j < 100) {
    System.out.println(-1);
    j++;
}
int /* BigInteger */ i = 0;
while(i < 100) {
    System.out.println(-1);
    i++;
}
```

После:

```
for (java.math.BigInteger j = java.math.BigInteger.ZERO;
j.compareTo(java.math.BigInteger.valueOf(200)) < 0; j =
j.add(java.math.BigInteger.ONE)) {
     System.out.println(-1);
}
int j = 0;
while (j < 100) {
    System.out.println(-1);
    j++;
}
java.math.BigInteger i = java.math.BigInteger.ZERO;
while (i.compareTo(java.math.BigInteger.valueOf(100)) < 0) {
    System.out.println(-1);
     i = i.add(java.math.BigInteger.ONE);
}
```

III пример:

До:

```
int /* BigInteger */ a = 2 * 3 - 100 * (12 - 11) - (13 + (12
/ 3) + 1) / 3;
int /* BigInteger */ charInt = 'a' + 'c' * 5;
```

После:

```
java.math.BigInteger a =
java.math.BigInteger.TWO.multiply(java.math.BigInteger.valueOf(3))
.subtract(java.math.BigInteger.valueOf(100).multiply((java.math.Bi
gInteger.valueOf(12).subtract(java.math.BigInteger.valueOf(11)))))
.subtract((java.math.BigInteger.valueOf(13).add((java.math.BigInte
ger.valueOf(12).divide(java.math.BigInteger.valueOf(3)))).add(java
.math.BigInteger.ONE)).divide(java.math.BigInteger.valueOf(3)));
java.math.BigInteger charInt =
java.math.BigInteger.valueOf('a').add(java.math.BigInteger.valueOf
('c').multiply(java.math.BigInteger.valueOf(5)));
```

По этой ссылке можно посмотреть на код: https://github.com/arksap2002/int-increase.

Выводы

Я достаточно долго писал на языке *Java*, и до этого года не знал, как распознается мой код, как он парситься и запускается. Изучение *java-parser* оказалось очень интересным и совсем не таким простым, как казалось вначале. В процессе решения возникло большое количество непредвиденных проблем, которые нужно было решать, иногда полностью преобразуя написанный мною код.

Также я научился работать с *git*, *GitHub* и *Travis*, что мне точно пригодиться при работе над следующим большим проектом.

Список литературы

1) https://javaparser.org
2) https://git-scm.com/
3) https://learngitbranching.js.org/
4) https://graphviz.gitlab.io/
5) http://mydebianblog.blogspot.com/2010/01/graphviz.html
6) https://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html
7) https://docs.oracle.com/javase/7/docs/api/java/lang/Integer.html
8) https://docs.oracle.com/javase/tutorial/java/nutsandbolts/expressions.html
9) https://habr.com/ru/company/jugru/blog/348710/
10) https://git-scm.com/book/en/v2
11) https://help.github.com/en/github/getting-started-with-github/create-a-repo
12) https://help.github.com/en/github/creating-cloning-and-archiving-repositories/cloning-a-repository
13) https://maven.apache.org/guides/introduction/introduction-to-the-standard-directory-layout.html
14) https://mvnrepository.com/artifact/junit/junit/4.12
15) https://docs.travis-ci.com/user/tutorial/
16) https://www.baeldung.com/java-initialize-hashmap
17) https://www.javadoc.io/doc/com.github.javaparser/javaparser-core/3.13.9/index.html