Санкт-Петербургский Государственный Университет

Математико-механический факультет

Кафедра системного программирования

Поиск шаблонов в программном коде

Курсовая работа студента 445 группы

Куделевского Евгения Валерьевича

Научный руководитель Максим Мосиенко

к.ф.-м.н., ст. разработчик компании JetBrains

Санкт-Петербург

2010

Оглавление

Введение

В области программной инженерии часто возникает задача поиска в большом объеме программного кода. Поиск может быть как точным, когда ищутся фрагменты кода, совпадающие с заданным шаблоном на уровне текстового представления, так и неточным, когда поиск ведется по каким-либо критериям, приблизительно описывающим искомый фрагмент. Для нас интерес представляет именно 2-ой случай. Итак, под **шаблоном** будем понимать фрагмент кода на каком-либо языке программирования, в котором также могут присутствовать специальные конструкции – **шаблонные переменные**. Шаблонные переменные – это параметры, вместо которых могут подставляться различные сущности языка, такие как выражения, идентификаторы, и т.п. При этом, как на значения шаблонных переменных, так и на фрагмент в целом могут накладываться самые различные ограничения, начиная от простых регулярных выражений, и заканчивая скриптами-предикатами.

В некоторых случаях поиск является первой частью процесса изменения кода, например процесса замены всех фрагментов, соответствующих заданному шаблону на другой шаблон.

Механизм поиска по шаблону может иметь множество применений, вот некоторые из них:

- Извлечение знаний из кода (reverse engineering)
- Инспекция кода (например, поиск «нежелательных» фрагментов кода, «антипаттернов» и замена их «правильными фрагментами»)
- Нахождение дубликатов

Чтобы формализовать задачу поиска по шаблону, необходимо ввести отношение соответствия между множеством фрагментов кода и множеством шаблонов. Пусть это отношение состоит из 2-х компонент: множества классов синтаксических элементов языка *var_elements* (множества значений шаблонных переменных) и отношения эквивалентности ~, заданного на множестве фрагментов кода. Итак, фрагмент кода *code*, соответствует шаблону *template* тогда и только тогда, когда существуют такие значения шаблонных переменных класса *var_elements*, при подстановке которых в шаблон, полученный фрагмент кода будет эквивалентен фрагменту *code* в смысле отношения ~, а также выполнены дополнительные ограничения на фрагмент и на значения переменных. Во множество классов *var_elements*, например, могут входить такие классы, как выражения или идентификаторы. В качестве отношения ~, вообще говоря, естественно взять семантическую эквивалентность фрагментов, но в общем случае установить эту эквивалентность практически невозможно. Простой пример отношения ~ - лексическое соответствие: фрагменты кода считаются

эквивалентными, если после лексического разбора и удаления некоторых типов лексем, например пробелов и комментариев, их лексические последовательности совпадают.

Тем не менее, простейшую семантическую эквивалентность при поиске по шаблону крайне желательно учитывать. Например, в языках JavaScript и ActionScript, как и во всех C-подобных языках, необязательно ставить фигурные скобки в условиях и циклах, если в блоке лишь один оператор. То есть следующие фрагменты кода будут эквивалентны:

```
if (i == n) break;
if (i == n) {break;}
```

Таким образом, если фрагмент отличается от шаблона наличием или отсутствием этих необязательных фигурных скобок, он все равно соответствует данному шаблону.

Цель курсовой работы

Цель моей курсовой работы – разработать метод поиска шаблонов программного кода и на основе него реализовать механизм поиска по шаблону для языков JavaScript и ActionScript, работающий внутри среды разработки IntelliJ IDEA. Метод должен быть ориентирован на поиск по запросу, то есть время ожидания пользователем результата должно быть минимизировано. В то же время должно быть возможно учитывать простейшую семантику языковых конструкций и допускать в качестве шаблонных переменных различные синтаксические единицы языка. Метод должен допускать расширяемость реализации, то есть объем работ по поддержке нового языка программирования должен быть минимальным.

1. Связь с задачей поиска дубликатов

Помимо задачи поиска по шаблону, существует более известная задача - задача поиска **дубликатов**, которая также имеет множество применений. Под дубликатами здесь понимаются «похожие» фрагменты программного кода. Чтобы формализовать эту задачу, необходимо ввести некоторое отношение эквивалентности ≈ на множестве фрагментов. Пусть задано отношение соответствия →, при этом в нем отсутствуют какие-либо дополнительные ограничения, и отношение полностью определяется упорядоченной парой *(var_elements, ~)*. Определим отношение ≈ следующим образом: фрагменты считаются эквивалентными, если после «уравнивания» всех синтаксических элементов класса *var_elements* полученные фрагменты кода эквивалентны в смысле отношения ~. Под «уравниванием» понимается следующая процедура: если 2 элемента e1 и e2 имеют один класс, например выражения, и этот класс принадлежит *var_elements*, то они должны быть заменены на одинаковые элементы этого же класса. То есть, если *var_elements* = «выражения и идентификаторы», то все выражения будут заменены одинаковыми выражениями, а все идентификаторы – одинаковыми идентификаторами. Нетрудно заметить, что 2 фрагмента эквивалентны в смысле ≈ тогда и только тогда, когда существует шаблон, которому соответствуют они оба в смысле →, а значения одинаковых шаблонных переменных имеют одинаковые классы. Таким образом, существует явная связь между задачей поиска по шаблону и задачей поиска дубликатов. Метод решения задачи поиска по шаблону может быть адаптирован для решения задачи поиска дубликатов и наоборот. В частности, разработанный мной метод может быть применен для реализации инструмента поиска дубликатов.

2. Обзор средств и подходов

Как было показано выше, задача поиска шаблонов тесно связана с задачей поиска дубликатов, метод решения 1-ой задачи может быть применен для решения 2-ой и наоборот. По причине малой популярности темы поиска по шаблону, в этом разделе будут в основном рассматриваться методы, которые так или иначе были применены именно для решения задачи поиска дубликатов.

Итак, существует 3 подхода к решению подобного рода задач – текстовый, синтаксический и лексический. Последний значительно выигрывает в производительности, в то время как первый предоставляет большую гибкость.

2.1. Текстовый подход

Исходный код программы рассматривается как обычный текст. Шаблоны при таком подходе, по сути, представляют собой регулярные выражения. Этот подход не представляет большого интереса, так как является менее гибким, чем лексический подход, и неэффективным для неточного поиска. Главное его достоинство в том, что он никак не привязан к конкретному языку программирования, и инструмент, использующий данный подход, может быть применен к абсолютно любому коду. Далее этот подход обсуждаться не будет.

2.2. Синтаксический подход

Основа синтаксического подхода – абстрактное синтаксическое дерево программы. Дерево строится как для исходного кода, так и для шаблона. Следует заметить, что шаблон, состоящий, например, из нескольких подряд идущих операторов, рассматривается не как одно, а как последовательность поддеревьев. Таким образом, возникает задача поиска эквивалентного поддерева (или последовательности поддеревьев) в дереве. При этом необходимо уметь определять, является ли узел дерева шаблонной переменной. Если да, он считается эквивалентным определенному классу узлов.

Главное преимущество такого способа – гибкость. На уровне синтаксического дерева мы располагаем всей информацией о синтаксисе программы, можем различать языковые конструкции, и, следовательно, можем учитывать их семантику. Например, как показано на Рис.1, мы можем считать эквивалентными блок с единственным внутренним оператором и сам этот оператор.

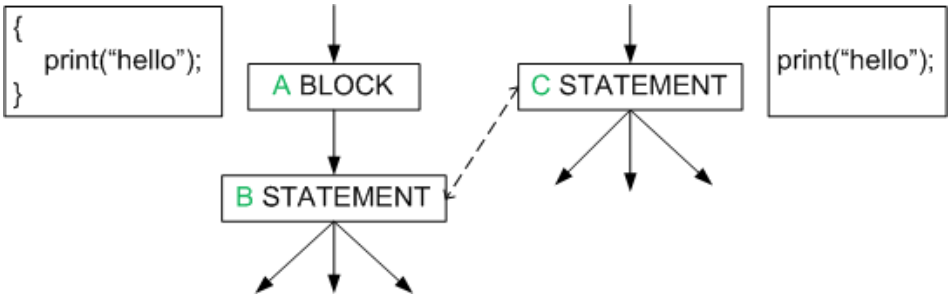

Рис. 1: Если поддеревья B и C – эквивалентны, значит A и C – также эквивалентны

Гибкость синтаксического подхода проявляется также в том, что вместо шаблонных переменных могут быть подставлены любые синтаксические единицы (то есть поддеревья синтаксического дерева). Разумеется, возможно ограничить классы синтаксических единиц, которые могут выступать в роли значений шаблонных переменных.

Несмотря на неоспоримые достоинства данного метода, он не слишком эффективен, так в общем случае происходит полный обход синтаксического дерева и сопоставление с каждым его поддеревом дерева шаблона. Таким образом, очень многие узлы синтаксического дерева будут просмотрены несколько раз.

Синтаксический подход был применен для создания инструмента поиска дубликатов еще в 1998 году [6].

2.3. Лексический подход

При лексическом подходе исходный текст рассматривается как обобщенная последовательность лексем. Обобщенные лексемы могут быть 3-х типов: конкретные, абстрактные и маркеры. Конкретная лексема – обычная лексическая единица языка программирования, а маркеры представляют конец файла и содержат информацию о нем. Кроме того, некоторые типы лексем, например идентификаторы и литералы, заменяются на абстрактные. Например, имея исходный код *if (a > b) println("hello");* получим последовательность лексем {'if', '(', ID, '>', ID, ')', ID, '(', STRING, ')', ';'} , где ID и STRING – это абстрактные лексемы. Абстрактные лексемы, как нетрудно догадаться, нужны для неточного поиска, они могут подставляться вместо шаблонных переменных.

Таким образом, задача поиска шаблона сводится к поиску подстроки в строке, где символами являются лексемы, подстрокой – шаблон, а строкой – обобщенное лексическое представление исходного кода. Поиск дубликатов, в свою очередь, сводится к поиску одинаковых подстрок.

Несложно понять, что при использовании быстрых алгоритмов поиска подстроки в строке, лексический подход гораздо более эффективен, нежели синтаксический. Например, суффиксное дерево, которое обычно применяется при реализации лексического подхода,

позволяет добиться времени работы поиска $O(m \cdot occ)$, где m – длина шаблона, а occ – количество найденных соответствий. Другое достоинство лексического подхода – меньшая зависимость от конкретного языка программирования, так как нам не нужно производить синтаксический разбор кода, а лишь разбить его на лексемы. Однако такой подход предоставляет меньшую гибкость:

- В роли значений шаблонных переменных могут выступать только лексемы, но не более сложные синтаксические единицы языка, такие как выражения.
- Невозможно учитывать простейшую семантику языковых конструкций, в то время как синтаксический подход это позволяет.

2.3.1. Использование суффиксного дерева

Существует несколько методов решения подобных задач на лексической основе [4, 8], практически во всех предполагается построение суффиксного дерева или аналогичной структуры.

Суффиксное дерево – это дерево, где каждый путь от корня к листу соответствует некоторому суффиксу строки, причем каждый его внутренний узел имеет по крайней мере 2 потомка. Пример суффиксного дерева для строки BANANA показа на Рис. 2

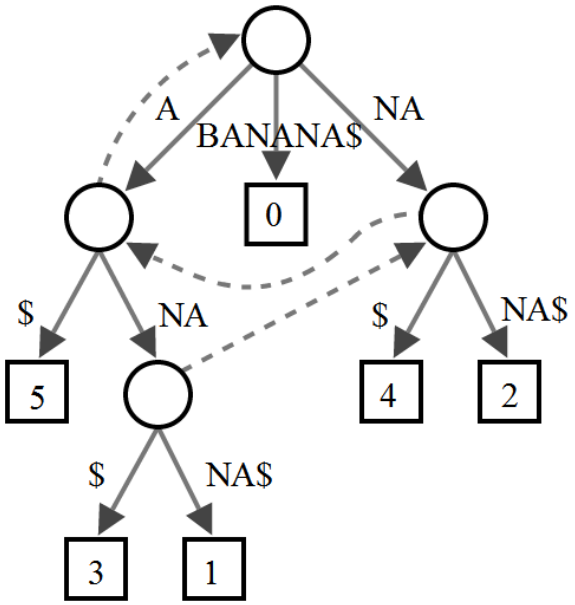

Рис. 2 Суффиксное дерево для строки BANANA. Символ $ обозначает конец строки, метки на ребрах – некоторые подстроки исходной строки, метка на каждом листе – позиция суффикса, соответствующего пути от корня к данному листу.

Суффиксное дерево, как известно, позволяет очень быстро выполнять поиск подстроки в строке ($O(m \cdot occ)$, где m – длина подстроки, а occ – количество найденных вхождений). Кроме того, суффиксное дерево возможно построить за линейное время от длины строки с помощью алгоритма Укконена [3] или алгоритма Мак-Крейта [2]. В [8] для

реализации инструмента поиска дубликатов предлагается строить суффиксное дерево для обобщенного лексического представления кода, подобного тому, что было описано выше, и обновлять его по мере изменения кода.

Суффиксное дерево – довольно объемная структура. В худшем случае она имеет 2n узлов. Для каждого ребра требуется хранить метку – пару индексов, идентифицирующих подстроку. Для каждого узла – указатель на предка. Также требуется хранить хэш-таблицу «узел * символ -> исходящее ребро». Она-то и занимает основную часть памяти. Тем не менее, если хранить дерево на жестком диске, большой расход памяти вполне допустим. Кроме того, в 2007 году S.Wong предложил метод наиболее эффективного хранения суффиксного дерева на диске [10].

2.3.2. Использование суффиксного массива

Для экономии памяти вместо суффиксного дерева можно использовать альтернативную структуру – суффиксный массив. Суффиксный массив – это массив суффиксов строки, отсортированный в лексикографическом порядке. Пример суффиксного массива показан на Рис. 3.

Суффикс	Индекс в строке
$	6
A$	5
ANA$	3
ANANA$	1
BANANA$	0
NA$	4
NANA$	2

Рис. 3. Суффиксный массив для строки “BANANA”. Символ $ обозначает конец строки.

С помощью бинарного поиска в суффиксном массиве можно находить подстроки почти также эффективно, как и в суффиксном дереве: $O(m \cdot \log n)$, где m – размер подстроки, а n – размер строки. Нетрудно заметить, что суффиксный массив – структура гораздо более компактная, чем суффиксное дерево: каждый суффикс можно хранить просто как индекс, а суффиксов всего n. В 2003 году Pang Ko предложил алгоритм построения суффиксного массива за линейное время [9]. В 2007 году суффиксный массив был применен для реализации инструмента поиска дубликатов на лексической основе [4].

Главный недостаток суффиксного массива в том, что он не приспособлен к динамическому изменению текста. Обновление же суффиксного дерева возможно выполнить

с помощью упрощенного алгоритма Мак-Крейта [1] – он выполняет обновление за время $O(k)$, где k – размер изменившегося файла, а n – размер всего исходного кода. Однако в 2004 году был предложен метод индексации строки использующий специфический способ хранения суффиксного массива [5]. Построение индекса происходит за время $O(n \cdot \log n)$, затем индекс позволяет выполнять операции удаления и вставки за время $O(t \cdot \log n)$, где t – размер удаленного/вставленного текста. Также, в конце 2009 года M. Salson предложил другой индекс [7], также на основе суффиксного массива, обновляющийся достаточно эффективно на практике, в среднем быстрее чем $O(t \cdot \log n)$, но в худшем случае на очень специфическом тексте время будет порядка $O(n \cdot \log n)$.

Такие «динамические» суффиксные массивы имеют несколько недостатков. Во-первых, они требуют памяти значительно больше, чем обычный суффиксный массив: $O(S \cdot n)$, где S – размер алфавита. Во-вторых, они плохо приспособлены к хранению на жестком диске, так как отсутствуют эффективные методы локализации, и количество обращений к жесткому диску при выполнении операций удаления/вставки будет очень большим, что значительно увеличит время работы этих операций на практике.

2.4. Моя задача

В рамках данной курсовой работы я попытался разработать метод, который был бы достаточно гибким, то есть допускал бы некоторые абстрактные языковые конструкции в качестве значений шаблонных переменных, а также позволял бы учитывать простейшую семантику конструкций языка. Кроме того, метод должен быть ориентирован на работу внутри интегрированной среды разработки, то есть в условиях постоянно изменяющегося кода, позволять достаточно быстро производить поиск «по запросу», а также позволять лекго добавлять поддержку новых языков в существующую реализацию. Также в мою задачу входит применить данный метод для реализации механизма поиска по шаблону для языков JavaScript и ActionScript, работающего внутри среды разработки IntelliJ IDEA. Следует заметить, что предлагаемый мною метод, может быть с некоторыми изменениями, применен при создании инструмента быстрого поиска дубликатов.

3. Описание метода

Мой метод представляет собой некоторую комбинацию лексического и синтаксического методов. Как и синтаксический метод, он допускает синтаксические единицы в качестве значений переменных, а также позволяет учитывать семантику элементарных языковых конструкций. С другой стороны, он более эффективен, нежели синтаксический.

Итак, по исходному коду программы строится обобщенное лексическое представление, подобное тому, что было описано в разделе 2.3, но абстрактные лексемы могут представлять не только лексические сущности языка, но и более сложные синтаксические сущности, например выражения или параметры. Например, имея код *if (a > b) System.out.println("hello");*, получим последовательность лексем {'if', '(', EXPRESSION, ')', EXPRESSION}, где EXPRESSION – абстрактная лексема. Также при построении такого представления кода не будем различать некоторые структуры, идентичные с семантической точки зрения, например *if (a > b) println("hello");* и *if (a > b) { println("hello"); }*, то есть не будем генерировать лексемы { и }.

По шаблону тоже строится синтаксическое дерево и обобщенная последовательность лексем ровно таким же способом. Шаблонные переменные при этом рассматриваются как идентификаторы и в итоге должны замениться на какую-либо абстрактную лексему.

Сам поиск происходит в 2 фазы:

1. *Лексический поиск.* Поиск в обобщенной последовательности лексем программы подпоследовательности-шаблона. Под подпоследовательностью здесь понимается именно последовательность подряд идущих элементов.
2. *Синтаксический поиск.* Сопоставление синтаксического дерева шаблона с синтаксическими поддеревьями программы, располагающимися в области, где был получен положительный результат в 1-ой фазе. На этой стадии мы располагаем информацией, которая была потеряна вследствие использования абстрактных лексем, и можем отсеять лишние вхождения.

3.1. Обобщенное лексическое представление

Назначение фазы лексического поиска – фильтрация, то есть лексический поиск должен находить все «настоящие» вхождения шаблона, но также может находить и ложные вхождения, которые впоследствии будут отсечены на этапе синтаксического поиска. Из этого следует, что для любого шаблона и любого фрагмента кода, который ему соответствует, их обобщенные лексические представления должны совпадать. Простейший метод построения обобщенной лексической последовательности был описан в разделе 2.3,

однако, этот метод «работает», только для отношения соответствия, когда во множество значений шаблонных переменных входят только классы лексических единиц языка, например идентификаторы и литералы, а в качестве отношения ~ используется обычное лексическое соответствие. Чтобы допускать в качестве значений шаблонных переменных синтаксические единицы, будем строить обобщенную лексическую последовательность не с помощью лексера, а из синтаксического дерева программы, обходя его в глубину и заменяя поддеревья, корни которых могут являться значениями шаблонных переменных, абстрактными лексемами. Процесс построения последовательности из синтаксического дерева показан на Рис. 4.

Также, чтобы учитывать простейшую семантическую эквивалентность некоторых языковых конструкций, в процессе обхода для некоторых типов узлов будем применять особую стратегию построения лексической последовательности. Например, в условных операторах и операторах цикла C-подобных языков не будем генерировать фигурные скобки, ограничивающие их тело.

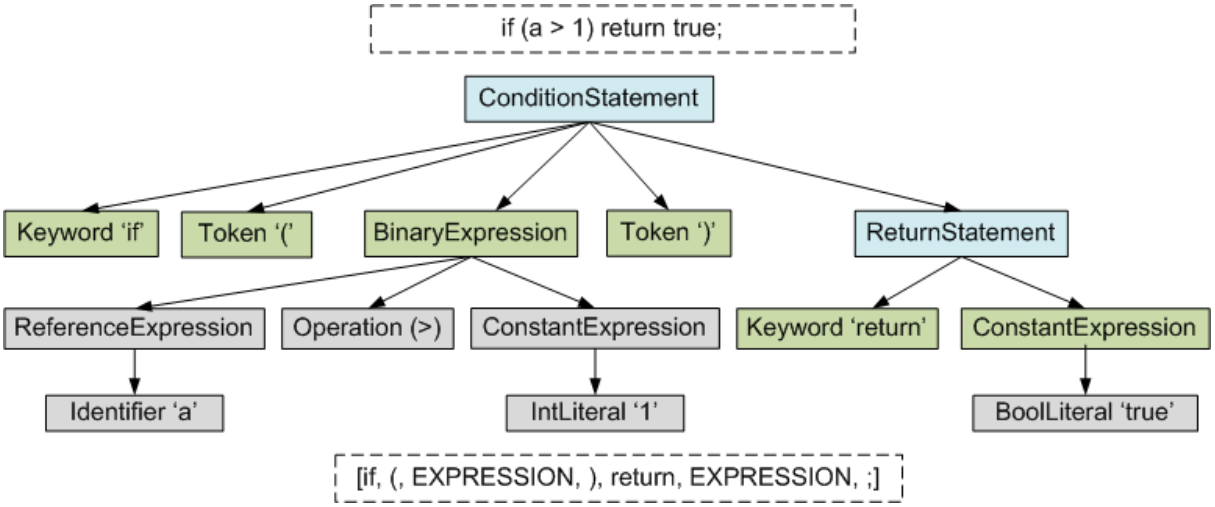

Рис. 4 Голубым отмечены посещенные промежуточные узлы, зеленым - узлы, соответствующие абстрактным лексемам, серым - непосещенные узлы. В качестве значений шаблонных переменных здесь могут выступать выражения.

3.1.1. Лексический индекс

В некоторых публикациях, посвященных задаче поиска дубликатов на лексической основе, предлагается хранить индекс исходного кода на основе суффиксного дерева или суффиксного массива, построенных по лексическому представлению кода. Как уже было рассказано в разделе 2.3, эти структуры хороши тем, что позволяют впоследствии быстро выполнять поиск подстроки в строке, но для нашей задачи необходимо еще и динамическое обновление индекса, так как инструмент работает внутри IDE и полностью перестраивать индекс при малейшем изменении кода неэффективно.

Проведенные мной тесты показали, что на практике динамически обновлять индексы на основе суффиксных структур очень непросто, так как для больших проектов его необходимо хранить на жестком диске, а при изменении одного файла может потребоваться множество обращений в различные области диска, поэтому крайне сложно добиться приемлемой производительности.

Вместо сложных индексов, основанных на суффиксных структурах, данный метод использует простой лексический индекс: массив подряд идущих обобщенных лексем. При хранении такого индекса на жестком диске, лексемы, относящиеся к одному файлу, находятся рядом, и поэтому его просто обновлять при каждом изменении кода. Кроме того этот индекс очень компактен: тесты показали, что для JavaScript кода размер индекса в байтах приблизительно в 5 раз меньше размера самого исходного кода.

3.2. Лексический поиск

Первый этап процесса поиска по шаблону – лексический поиск. Для шаблона строится обобщенная последовательность лексем и выполняется поиск в обобщенном лексическом представлении кода. По сути, этот процесс является поиском подстроки в строке, где символами являются лексемы.

Для поиска подстроки в строке вместо суффиксных структур данный метод использует алгоритм Кнута-Морриса-Пратта (КМП) [11]. Время его работы – $O(m+n)$, где m – длина подстроки, а n – длина строки. Как показали проведенные мной измерения, если в процессе построения лексической последовательности заменять абстрактными лексемами все выражения и идентификаторы, то количество лексем в коде на языках JavaScript и ActionScript будет примерно в 50 раз меньше, чем размер самого кода. То есть при объеме кода 200 мб., обобщенных лексем будет немногим больше 4 млн., поэтому лексический поиск будет происходить достаточно быстро. Достоинство алгоритма КМП в том, что он сканирует строку последовательно, что позволяет не загружать лексическое представление исходного кода с жесткого диска полностью, а загружать инкрементально, небольшими блоками, чтобы потреблять меньший ресурс оперативной памяти.

3.3. Синтаксический поиск

После завершения работы фазы лексического поиска начинается фаза синтаксического поиска. Назначение данного процесса – на основе предварительной информации о возможных вхождениях шаблона, полученной на этапе лексического поиска, найти все настоящие вхождения.

На этапе синтаксического поиска происходит обход в глубину синтаксического дерева программы, а точнее тех его поддеревьев, которые находятся в области, где были обнаружены соответствия в фазе лексического поиска. В процессе обхода для каждого рассматриваемого узла, находящееся под ним поддерево сравнивается с синтаксическим деревом шаблона и, если результат сравнения положительный, то обрабатывается очередное найденное соответствие, иначе поиск продолжается в этом поддереве.

Заметим, что фаза синтаксического поиска не только выполняет «отсев» ложных соответствий, полученных в 1-ой фазе, но и может обнаружить новые, так как внутри одной абстрактной лексемы может оказаться несколько вхождений шаблона.

3.4. Баланс между двумя фазами

Если говорить кратко, то, чем больше синтаксических элементов заменяется абстрактными лексемами на этапе лексического поиска (то есть допускается в качестве шаблонных переменных), тем больше эта фаза порождает «ложных» соответствий, и, следовательно, тем больше работы нам придется делать на этапе синтаксического поиска. Таким образом, необходимо найти идеальный баланс между гибкостью и производительностью. В разработанном мной инструменте, для языка JavaScript, я включил в множество значений шаблонных переменных идентификаторы, литералы и параметры.

3.5. Зависимость от конкретного языка

В некотором смысле предложенный метод не привязан к конкретному языку программирования. Действительно, литералы и идентификаторы есть практически в любом языке, и, если добавить во множество значений шаблонных переменных только их, то мы получим поиск по шаблону для любого языка, достаточно лишь располагать его парсером. Фаза синтаксического поиска в этом случае может вообще отсутствовать. Если нужен более гибкий поиск, появляется часть поисковой системы, ориентированная на конкретный язык.

3.6. Возможное применение к задаче поиска дубликатов

Описанный метод с некоторыми изменениями может быть применен для решения задачи поиска дубликатов кода. Для поиска дубликатов на этапе лексического поиска вместо алгоритма КМП можно строить суффиксный массив, а в нем производить поиск одинаковых подстрок. Возникает проблема, что построение суффиксного массива происходит в оперативной памяти, иначе, как уже говорилось в разделе 2.3.2, значительно снижается производительность. В 2005 году Yuta Mori предложил «легковесный» алгоритм построения суффиксного массива. Исходный код этого алгоритма на языке C++ под лицензией MIT можно найти на странице проекта http://code.google.com/p/libdivsufsort/. Алгоритм работает за время $O(n \cdot \log n)$ и требует всего $O(1)$ дополнительной памяти, то есть память используется практически только для хранения суффиксного массива.

На этапе синтаксического поиска мы имеем область исходного кода, в которой, с определенной долью вероятности, могут быть дубликаты. Чтобы не сравнивать попарно все последовательности поддеревьев в этой области, применяется хэширование синтаксического дерева: сравниваются только те деревья, хэши которых совпали.

4. Некоторые детали реализации

В этом разделе будут описаны некоторые алгоритмы и решения, которые я применил в реализации механизма поиска по шаблону для языков ActionScript и JavaScript, основаной на вышеизложенном методе поиска.

4.1. Форма записи шаблона

Ранее уже говорилось, что шаблон – это фрагмент кода на некотором языке программирования, в котором могут присутствовать шаблонные переменные. Все шаблонные переменные имеют имена, состоящие из букв, цифр и символов подчеркивания. Внутри шаблона переменная обозначается, как *$name$*, где *name* – ее имя. Простой пример шаблона: *int $variable$ = 0;*

4.2. Компиляция шаблона

При реализации метода удобно работать с шаблоном как с обычным фрагментом кода, но шаблон не является таковым из-за присутствия шаблонных переменных. Чтобы преобразовать шаблон в обычный код, выполним замену всех шаблонных переменных на некоторые идентификаторы языка программирования. Для JavaScript/ActionScript шаблонов, будем заменять переменную *$name$* на идентификатор *___$___name*. В дальнейшем нам потребуется возможность определять, является ли данный идентификатор преобразованной шаблонной переменной, и восстанавливать ее имя, поэтому в шаблонах для JavaScript и ActionScript нельзя допускать идентификаторы вида ___$__name, так как впоследствии их нельзя будет отличить от шаблонных переменных. Будем называть преобразование шаблона в обычный фрагмент кода его компиляцией.

4.3. Синтаксический разбор шаблона

При реализации алгоритма для шаблона требуется строить синтаксическое дерево. Любой шаблон, которому может соответствовать некоторый синтаксически корректный код, считается корректным и необходимо уметь производить его синтаксический разбор. Но из описания алгоритма компиляции не следует, что из любого корректного шаблона будет получен синтаксически правильный код, так как нет гарантии, что там, где согласно грамматике языка программирования допускается какой-либо элемент класса *var_elements*, также допускается идентификатор. Возникает ограничение на класс *var_elements*: любая синтаксическая единица этого класса может являться идентификатором. В частности, это означает, что в этом классе не могут присутствовать, такие сущности, как функции, классы,

различные виды разделителей (например, скобки), но могут присутствовать выражения, идентификаторы.

4.4. Построение обобщенного лексического представления

Далее изложен алгоритм построения обобщенного лексического представления кода из синтаксического дерева программы. Обозначим класс элементов, которые необходимо заменить абстрактными лексемами *anonym_elements*. Обходя синтаксическое дерево в глубину, будем последовательно генерировать результирующую последовательность следующим способом:

- Если просматриваемый узел – лист, то он соответствует обычной лексической единице, и мы добавляем в результирующую последовательность конкретную лексему.
- Если просматриваемый узел – класса *anonym_elements*, добавляем абстрактную лексему в результирующую последовательность.
- Если ни одно из перечисленных не верно, продолжаем обход в глубину.

Таким образом, имея исходный код *if (a > 1) return true;,* получим последовательность лексем {‘if’, ‘(‘, EXPRESSION, ‘)’, return, EXPRESSION, ‘;’}, где EXPRESSION – это абстрактная лексема. Для корректной работы алгоритма, необходимо, чтобы класс *anonym_elements* полностью включал в себя множество значений шаблонных переменных (то есть класс *var_elements*). В некоторых случаях классы *var_elements* и *anonym_elements* могут совпадать, но как будет показано в следующем разделе, это не всегда удобно. Еще одно требование к классу *anonym_elements*: все шаблонные переменные должны быть частью какой-либо синтаксической конструкции из класса *anonym_elements*, чтобы потом преобразоваться в абстрактные лексемы. Это требование, однако, не очень существенно, так как, чтобы его удовлетворить, достаточно включить в *anonym_elements* класс идентификаторов.

4.4.1. Представление лексем в памяти

Конкретную лексему будем представлять в памяти как структуру, содержащую пару целочисленных индексов start и end, представляющих ее местоположение в файле с исходным кодом, а также хэш ее текстового представления. Структура абстрактной лексемы также будет содержать пару индексов и еще переменную типа byte, представляющую тип абстрактной лексемы (выражения, идентификатор, и т.п.). Наконец лексему-маркер будем представлять в памяти как структуру, содержащую информацию о файле (например, путь к

файлу). Информация о местоположении лексемы в исходном файле используется при обработке результатов поиска. Таким образом, получаем, что для хранения одной конкретной лексемы необходимо 12 байт, для абстрактной – 9 байт, а для маркера – $p \cdot 4$ байт, где p – длина пути к файлу.

Чтобы рассматривать лексемы как символы строки и выполнять в таких строках поиск, необходимо определить понятие равенства лексем: лексема может быть равна только лексеме такого же типа, а 2 лексемы одного типа равны, если все их поля, кроме, возможно, индексов start и end, совпадают.

4.5. Алгоритм работы лексического поиска

Алгоритм работы лексического поиска, как уже было сказано в разделе 3.2, основан на алгоритме поиска подстроки в строке Кнута-Морриса-Пратта. В процессе работы алгоритма КМП, для очередного найденного соответствия выполняется следующее:

- Из самой левой лексемы соответствия считывается индекс start
- Из самой правой лексемы, не являющейся маркером, считывается индекс end
- Из ближайшего к соответствию справа маркера считывается путь к файлу file_path
- Структура (start, end, file_path) записывается в результирующий список

Этот список и есть результат работы фазы лексического поиска, он передается синтаксической фазе.

4.6. Алгоритм работы синтаксического поиска

Этот алгоритм получает на вход список структур (start, end, file_path), и генерирует на выход список такого же вида. При этом в результирующем списке могут быть не только элементы, которые уже были в исходном, но и новые элементы. Например, имея код *var n = a – (b + c);* и шаблон *$var1$ + $var2$*, на этапе лексического поиска мы произведем поиск обобщенной лексемы EXPRESSION в последовательности var IDENTIFIER = EXPRESSION;. В результате мы найдем единственный интервал (start, end), ограничивающий текст *a – (b + c)*, но результатом должен быть *b + c*. В этом случае входной и выходной списки вообще не будут иметь общих элементов.

Обозначим *pattern_roots* список корневых элементов синтаксического представления шаблона (ранее уже было замечено, что их может быть несколько), список имеет длину m. Алгоритм синтаксического поиска состоит в следующем:

1. Определяем множество корневых узлов, то есть таких узлов, что объединение их поддеревьев полностью покрывает все интервалы из входного списка.
2. Соединяем корневые узлы в единое дерево, добавив дополнительный элемент, потомками которого будут являться все корневые узлы.
3. Рекурсивно обходим полученное дерево, используя функцию *findMatches(node)*, работающую следующим образом:
    a. Если текущий узел node имеет меньше *m* потомков, то рекурсивно запускаем функцию *findMatches* для всех потомков узла.
    b. Если количество потомков текущего узла ≥ *m*, то производим сопоставление каждой последовательности подряд идущих потомков узла *node* длины *m* и списка *pattern_roots*. Алгоритм сопоставления поддеревьев будет описан далее.
        i. Для каждого сопоставления, которое дало положительный результат, вычисляем интервал, которому соответствует данная последовательность узлов в тексте и файл, где находится данный код, а затем добавляем результаты в выходной список.
        ii. Рекурсивно запускаем функцию *findMatches* для всех потомков, которые не вошли ни в какое соответствие.

4.6.1. Алгоритм сопоставления поддеревьев

В процессе работы алгоритма синтаксического поиска используется операция сопоставления некоторого поддерева синтаксического дерева и дерева шаблона. Опишем алгоритм, производящий такое сопоставление. Алгоритм работает рекурсивно, начиная проверку с корней 2-х деревьев. Будем обозначать текущий узел дерева кода *node*, а текущий узел дерева шаблона – *pattern_node*. Кроме того, в процессе работы алгоритма будем строить хэш-таблицу *values* = «имя переменной → узел-значение». Итак, введем функцию *match(node, pattern_node)*, производящую сопоставление 2-х узлов *node* и *pattern_node* и возвращающую результат в виде boolean значения:

1. Если поддерево, с корнем *pattern_node* имеет единственный лист – идентификатор, причем имя идентификатора говорит о том, что он был сгенерирован из шаблонной переменной с именем *var* (например, идентификатор имеет имя *___$___name*), то проверяется, имеет ли узел *node* класс из множества *var_elements*. Если нет, то результат сопоставления объявляется отрицательным. В противном случае существует 2 варианта:

a. Хэш-таблица *values* содержит для переменной *name* некоторое значение *value_node*. Тогда *match(node, pattern_node) = match(node, value_node)*.
b. Хэш-таблица *values* еще не содержит значения для переменной *name*. Тогда проверяются дополнительные ограничения, накладываемые на значение переменной, а затем узел *node* записывается в хэш-таблицу *values* как значение переменной *name*.

2. Узел *pattern_node* сам является листом и этот лист – не идентификатор, порожденный шаблонной переменной. Тогда проверяется, является ли листом узел *node*. Если нет – результат отрицательный, иначе сравниваются текстовые представления узлов.
3. Поддерево с корнем *pattern_node* имеет больше, чем один лист, либо этот лист не является идентификатором, порожденным шаблонной переменной, а сам *pattern_node* листом не является. Тогда результат и процесс его вычисления, вообще говоря, могут зависеть от класса конкретного узла, но в общем случае проверяется равенство классов узлов *node* и *pattern_node*, а затем проверяются на соответствие все пары потомков, без учета пробелов и комментариев. Если узлы имеют разное количество потомков, не считая пробелов и комментариев, то результат сопоставления – отрицательный. Эту стратегию можно изменить для некоторых типов узлов, чтобы учесть семантику соответствующих языковых конструкций. Подробнее об этом будет рассказано в разделе 4.7.

Расширим область определения функции *match*, допустив, что она может принимать 2 списка узлов: *nodes* и *pattern_nodes*, и последовательно производить последовательное сопоставление пар. Если списки имеют разную длину, результат работы функции *matches* – отрицательный.

4.7. Семантическая эквивалентность

Чтобы учитывать семантику некоторых языковых конструкций, необходимо, во-первых сделать так, чтобы обобщенные лексические представления семантически эквивалентных конструкций не различались, а во-вторых, применить особую стратегию в алгоритме сопоставления поддеревьев для некоторых типов узлов. Я приведу несколько примеров, как именно можно учитывать семантику различных синтаксических конструкций на примере языков JavaScript и ActionScript:

- **Необязательные фигурные скобки.** В JavaScript, как и во всех C-подобных языках, необязательно заключать его в фигурные скобки, если в условных операторах и операторах цикла тело имеет лишь один оператор. Чтобы учесть эту эквивалентность на этапе лексического поиска достаточно в процессе построения лексического представления не добавлять в последовательность конкретные лексемы, соответствующие фигурным скобкам вокруг блоков, содержащих только 1 оператор, если их предок – цикл, либо условие. На этапе синтаксического поиска необходимо применить особую стратегию сопоставления поддеревьев, если один из узлов *node* и *pattern_node* – узел типа STATEMENT, а другой – узел типа BLOCK, имеющий единственного потомка типа STATEMENT, причем родители узлов – условные операторы, либо операторы цикла. Тогда производится сопоставление узла типа STATEMENT и единственного потомка узла BLOCK того же типа. То есть, алгоритм как бы «пропускает» узел BLOCK в случае необходимости.
- **Различный порядок методов класса в ActionScript.** В ActionScript, как и во многих других языках, не имеет значения, в каком порядке указаны методы внутри класса. В таком случае, на этапе лексического поиска можно обходить поддеревья методов в каком-то фиксированном порядке, например в лексикографическом порядке их имен. На этапе синтаксического поиска нужно в процессе работы функции *match* вызывать ее рекурсивно для методов класса в таком же порядке. Заметим, что такой способ применим ко всем случаям, когда порядок элементов не важен, например, для списка реализуемых классом интерфейсов. Также заметим, что такие изменения в алгоритме поиска ограничивают множество допустимых шаблонов. Например, невозможно будет найти 2 подряд идущих метода с заданными именами, поскольку в обобщенном лексическом представлении они могут идти не подряд.

- **Обращение к элементам массива в JavaScript.** В JavaScript к элементам массива можно обращаться как к свойствам, то есть даже если *a* – обычный массив с целочисленными ключами, то при обращении к его элементу индекс можно указывать

в кавычках. Например, код *a["3"]* будет эквивалентен коду *a[3]*. Чтобы учитывать семантику этой конструкции на этапе лексического поиска, нужно генерировать последовательность всегда одним способом, например, всегда заменять индекс на абстрактную лексему STRING_LITERAL, соответствующую строковому литералу: {IDENTIFIER, '[', STRING_LITERAL, ']', IDENTIFIER, ';'}, здесь IDENTIFIER и STRING_LITERAL – абстрактные лексемы. На этапе синтаксического поиска, если узлы *node* и *pattern_node* имеют тип, соответствующий конструкции обращения к элементу массива (например ARRAY_ACCESS_EXPRESSION), то функция *match* не учитывает кавычки при сравнении поддеревьев индексов.

Заключение

Для некоторых видов шаблонов предложенный метод поиска гораздо более эффективен, нежели полностью основанный на синтаксическом дереве. В особенности это касается «больших» шаблонов, состоящих из нескольких операторов. В этом случае, если не включать тип «оператор» в класс *anonym_elements* качество фильтрации в первой фазе будет достаточно высоким. Для «маленьких» шаблонов, обобщенное лексическое представление которых состоит из 1-2 лексем, фильтрация на этапе лексического поиска практически отсутствует. Разумеется, проблему можно решить, исключив из класса *anonym_elements* некоторые типы элементов, но тогда невозможно будет использовать их в качестве значений шаблонных переменных, а также придется жертвовать гибкостью поиска.

С другой стороны, предложенный метод лучше, чем полностью основанный на лексическом представлении кода, так как он позволяет сделать поиск более гибким и использовать в качестве шаблонных переменных синтаксические конструкции.

Результат

В рамках данной курсовой работы мною был разработан метод поиска шаблонов программного кода, который работает достаточно быстро для большинства шаблонов, в особенности для «больших», например, состоящих из нескольких операторов, или представляющих функцию. Также он допускает в качестве значений переменных различные синтаксические единицы языка и способен учитывать простейшую семантику языковых конструкций. Метод в некотором смысле не привязан к конкретному языку и допускает простое расширение реализации, так как для минимальной поддержки нового языка программирования необходимо лишь располагать его парсером, строящим синтаксическое дерево, а также указать множество значений шаблонных переменных.

На основе этого метода мною был реализован механизм поиска по шаблону для языков JavaScript и ActionScript в среде разработки IntelliJ IDEA.

Также мною было получено, что индексы кода на основе суффиксных структур плохо применимы в условиях постоянно изменяющегося кода, например, внутри IDE, так как они требуют частого обращения в различные области жесткого диска из-за отсутствия эффективных методов локализации.

Некоторые возможные направления дальнейшей работы

- Поддержка многих языков программирования.
- Адаптация предложенного метода к задаче поиска дубликатов и разработка соответствующего инструмента.
- Совершенствование метода для более быстрого поиска «маленьких» шаблонов и возможности допускать в качестве значений шаблонных переменных «большие» синтаксические конструкции (например, операторы) без значительной потери производительности.

Список литературы

1. A. Amir, M. Farach, Z. Galil, R. Giancarlo, K. Park, “Dynamic dictionary matching”. - // Journal of Computer and System Sciences, 49 (2), 1993, p.208-222. - : Elsevier Science Publishers B. V.
2. E. M. McCreight, “A Space-Economical Suffix Tree Construction Algorithm”. - // Journal of the ACM 23 (2), 1976, p.262-272. - : Association for Computing Machinery
3. E. Ukkonen, “On-line construction of suffix trees”. - // Algorithmica 14(3), 1995, p. 249-260
4. H. A. Basit, S. J. Puglisi, W. F. Smyth, A. Turpin, S. Jarzabek, “Efficient token based clone detection with flexible tokenization”. - // The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering, 2007, p.513-516. - : Association for Computing Machinery
5. H.-L. Chan, W.-K. Hon, T.-W. Lam, K. Sadakane, “Compressed indexes for dynamic text collections”. - // ACM Translations on Algorithms 3(2**),** 2007, p. \21. - : Association for Computing Machinery
6. I. D. Baxter, A. Yahin, L. Moura, M. Anna, L. Bier, “Clone Detection Using Abstract Syntax Trees”. - // Proceedings of the International Conference on Software Maintenance, March 1998, p.368. - : IEEE Computer Society
7. M. Salson, T. Lecroq, M. Leonard, L. Mouchard, “Dynamic extended suffix arrays”. - Journal of Discrete Algorithms 8(2), June 2010, p.241-257. - : Elsevier Science Publishers B. V.
8. N. Gode, R. Koschke , “Incremental Clone Detection”. - // Software Maintenance and Reengineering, 2009, CSMR '09, 13th European Conference, March 2009, p.219-228. - : IEEE Computer Society
9. Pang Ko and Srinivas Aluru, "Space efficient linear time construction of suffix arrays". - // Combinatorial Pattern Matching, 2003, p.203-210. - : LNCS 2676, Springer
10. S. Wong , W. Sung , L. Wong, “CPS-tree: A Compact Partitioned Suffix Tree for Disk-based Indexing on Large Genome Sequences”. - // IEEE 23rd International Conference on Data Enginering, April 2007, p.1350-1354. - : IEEE Computer Society
11. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, "Section 32.4: The Knuth-Morris-Pratt algorithm". - // Introduction to Algorithms (Second ed.), 2001, p. 923–931. - : MIT Press and McGraw-Hill.