САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Математико-механический факультет

Кафедра системного программирования

Поздин Дмитрий Евгеньевич

Декомпиляция выражений по байт-коду JVM

Курсовая работа

Допущена к защите.
Зав. кафедрой:
д.ф.-м.н., проф. А.Н. Терехов

Научный руководитель:
к.ф.-м.н. Д.Ю. Булычев

Санкт-Петербург

2013

Оглавление

Введение

Компиляция — процесс преобразования программы в одном языке в эквивалентную в другом языке. Однако чаще всего под компиляцией подразумевается преобразование кода из высокоуровневого представления в низкоуровневое. Обратный процесс называется декомпиляцией. Декомпилятор — это программа, которая пытается осуществить процесс, обратный производимому компилятором: по данному исполняемому файлу программы, скомпилированному с высокоуровневого языка, она стремится выдать программу на языке высокого уровня (причем не обязательно это будет язык, на котором программа была написана исходно), которая будет выполнять те же самые функции, что и входная исполняемая программа. Декомпиляция появилась в 60-е годы 20 века сразу же, когда стали широко применяться компиляторы языков высокого уровня, но не утратила своей актуальности и по сей день.

Java Virtual Machine (сокращенно Java VM, JVM) [1] — виртуальная машина Java — основная часть исполняющей системы Java, так называемой Java Runtime Environment (JRE). Виртуальная машина Java интерпретирует байт-код Java, предварительно созданный из исходного текста Java-программы компилятором Java (javac). JVM может также использоваться для выполнения программ, написанных на других языках программирования.

Декомпиляцию для JVM можно рассматривать в контексте классической задачи многоязыковой трансляции. Если бы мы захотели сделать сделать программу, которая транслирует код из M входных языков в N выходных языков, то нам бы пришлось написать $M \times N$ трансляторов. Но если мы будем использовать промежуточный язык, так, что каждый входной язык транслируем сначала в промежуточный, а затем уже в выходной язык, то нам необходимо написать всего $M + N$ трансляторов.

Идея промежуточного представления кода при трансляции с одних языков на другие не нова. В качестве примера можно привести проекты Zephyr[1], SUIF [3], LLVM[2]. В роли такого промежуточного языка удобно взять байт-код JVM, поскольку он является в некотором роде универсальным представлением программ. Существует целый ряд языков разработанных специально для JVM и компилируемых в байт-код JVM. Среди них Scala[3], Kotlin[4], Groovy[5], Clojure[6] и ещё несколько. Для многих других языков существуют JVM-реализации[7].

В рамках курсовой работы решается задача декомпиляции выражений. Выражение — участок кода в теле метода, который оставляет на стеке после себя одно или несколько значений. Другими словами, необходимо реализовать построение промежуточного представления выражения из байт-кода JVM в виде абстрактного JVM-дерева разбора. Это дерево будет потом использовано для создания кода на целевых языках.

[1] Zephyr software, 2010, `http://www.zephyrsoftware.com`

[2] The LLVM Compiler Infrastructure, University of Illinois at Urbana-Champaign, `http://llvm.org/`

[3] Официальная страница языка Scala, `http://www.scala-lang.org/`

[4] Официальная страница языка Kotlin, `http://kotlin.jetbrains.org/`

[5] Официальная страница языка Groovy, `http://groovy.codehaus.org/`

[6] Официальная страница языка Clojure, `http://clojure.org/`

[7] `https://en.wikipedia.org/wiki/List_of_JVM_languages`

1 Общие сведения о JVM

JVM доступна для многих аппаратных и программных платформ. Использование одного байт-кода для всех JVM на всех платформах позволяет сказать о байт-коде JVM: «написано однажды — работает везде» [8].

JVM предназначена для исполнения байт-кода. Единица байт-кода — `class`-файлы (программы, скомпилированные в стандартизированный переносимый двоичный формат). Главной особенностью байт-кода JVM является то, что он объектно-ориентированный. Каждый класс содержит методы, поля, конструкторы, данные о суперклассе, в этом JVM-класс повторяет структуру Java-классов. Тело методов состоит из наборов инструкций. Вторая особенность — JVM стековая машина. Большинство инструкций берут несколько значений с вершины стека и возвращают значение, помещая его на стек.

В коде JVM нельзя работать с реальной архитектурой компьютера напрямую, например, с регистрами или оперативной памятью. В JVM используется виртуальная архитектура: *стек операндов*, *куча*, *массив локальных переменных*, *фреймы* и *пул констант*. Все это создается виртуальной машиной при запуске или выполнении программы.

1.1 Структура JVM

JVM может поддерживать много потоков выполнения одновременно. JVM использует модель выполнения на основе стеков. Каждый поток (thread) в JVM имеет собственный JVM-стек, создающийся одновременно с потоком. JVM-стек аналогичен стеку процедурного языка, такого как *C*: он содержит локальные переменные и частичные результаты и участвует в вызовах методов и возвратах управления.

[8] `http://en.wikipedia.org/wiki/Write_once,_run_anywhere/`

В стеках JVM хранятся *фреймы* (frame). При каждом вызове метода создается новый фрейм. Фрейм уничтожается, когда вызов метода завершается, вне зависимости от того, было ли завершение нормальным или было прервано вызовом необработанного исключения. Фрейм состоит из стека операндов, массива локальных переменных и ссылки на пул констант (runtime constant pool) класса выполняемого метода (рис. 1).

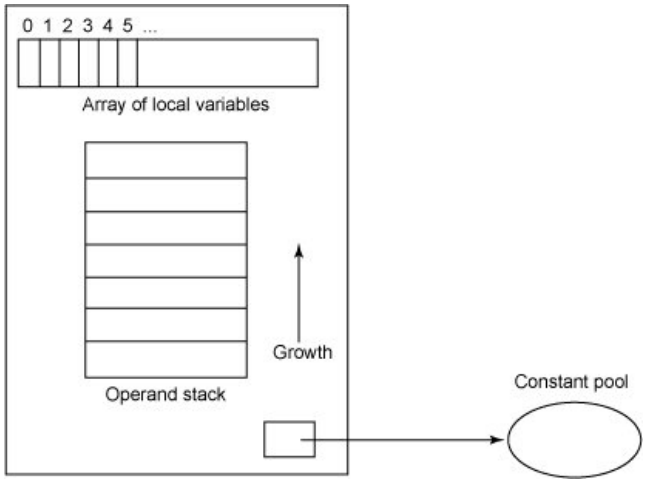

Рис. 1: Фрейм

Размер массива *локальных переменных* определяется во время компиляции в зависимости от количества и размера локальных переменных и параметров метода. *Стек операндов* — LIFO-стек для записи и удаления значений в стеке; размер также определяется во время компиляции. Некоторые инструкции добавляют значения в стек, другие берут из стека операнды, изменяют их состояние и возвращают в стек. Стек операндов также используется для получения значений, возвращаемых методом.

Каждый из потоков имеет свой *регистр PC* (program counter). В любой точке каждый поток JVM выполняет код одного метода, называемого текущим методом этого потока. Если этот метод не нативный (native), то регистр PC содержит адрес JVM-инструкции, выполняющейся в данный момент. Если метод

нативный, то значение регистра PC не определено. Размер регистра достаточно большой, чтобы содержать значение `returnAdress` или машинный указатель (native pointer) на специфической платформе.

В JVM есть *куча*, которая разделена между всеми потоками. Куча — это область данных времени выполнения (runtime data area), в которой расположены объекты классов и массивы. Куча создается при запуске виртуальной машины. Сборщик мусора занимается выделением памяти под объекты, удалением объектов и освобождением памяти.

Локальные переменные могут иметь типы `boolean`, `byte`, `char`, `short`, `int`, `float`, `reference` или `returnAdress`. Пара переменных может содержать значение типа `long` или `double`.

1.2 Инструкции JVM

Инструкции JVM состоят из однобайтового кода инструкции, за которым следуют ноль или более операндов в качестве аргументов. Некоторые данные, используемые в инструкции, передаются неявно через стек операндов. Перед инструкцией может стоять метка, на её можно ссылаться в переходах. Некоторые инструкции не имеют операндов и состоят только из кода инструкции. Многие инструкции имеют префиксы из одной буквы, соответствующие типам их операндов (i для операнда типа `int`, l для `long`, s для `short`, b для `byte`, c для `char`, f для `float`, d для `double`, и a для ссылки). Например, инструкция `iload` загружает содержимое локальной переменной, которая должна быть типа `int`, на стек операндов. В JVM не поддерживаются типы `boolean`, `byte`, `char` и `short`, вместо них используется тип `int`.

Далее будут использоваться следующие обозначения. Знак “`<n>`” означает некоторое натуральное число. “`x`” в начале инструкции — буква соответствую-

щая типу (*l, d, f, i* или *a*). “<x>” — число соответствующего типа. “m1” соответствует “−1”.

Инструкции load и store

Инструкции `load` и `store` перемещают значения между локальными переменными и стеком операндов фрейма. Среди них есть инструкции загрузки локальных переменных на стек операндов (`xload, xload_<n>`), инструкции загрузки значений со стек операндов в локальные переменные (`xstore, xstore_<n>`) и инструкции загрузки констант на стек операндов (`bipush, sipush, ldc, ldc_w, ldc2_w, aconst_null, xconst_m1, xconst_<x>`).

Арифметические операции

Арифметические операции вычисляют значение по нескольким аргументам и кладут результат на стек операндов. Аргументов обычно два, и они заданы неявно через стек операндов. Арифметические операции делятся на два главных типа: оперирующие с целыми числами и оперирующие с числами с плавающей точкой. Любая из них реализована для конкретных численных типов JVM. Есть различия в поведении целочисленных и операций с числами с плавающей точкой при переполнении и делении на ноль.

Существуют следующие арифметические операции: сложение (`xadd`), вычитание (`xsub`), умножение (`xmul`), деление (`xdiv`), взятие остатка (`xrem`), отрицание (`xneg`), двоичный сдвиг (`xshl, xshr, xushr`), поразрядное ИЛИ (`xor`), поразрядное И (`xand`), поразрядное исключительное ИЛИ (`xxor`), инкремент локальной переменной (`iinc`), сравнение (`xcmpg, xcmpl, lcmp`).

Инструкции приведения типов

Для приведения типов в JVM есть следующие инструкции: `i2l`, `i2f`, `i2d`, `l2f`, `l2d`, and `f2d`. Левая буква означает тип, который приводят, правая — тип к которому приводят. “2” означает “to”. Например, инструкция `i2d` приводит значение типа `int` в `double`.

Инструкции по управлению стеком операндов

Набор инструкций для прямого управления стеком операндов:

- снятие верхнего значения со стека: `pop, pop2`;
- дублирование одного или двух значений на вершине стека: `dup, dup2, dup_x1, dup2_x1, dup_x2, dup2_x2`;
- обмен местами двух значений на вершине стека: `swap`.

Инструкции переходов

Среди инструкций переходов бывают:

- Условные переходы
  - по сравнению с нулем: `ifeq, ifne, iflt, ifle, ifgt, ifge`;
  - по сравнению с null: `ifnull, ifnonnull`;
  - по сравнению двух операндов: `if_icmpeq,if_icmpne, if_icmplt, if_icmple, if_icmpgt if_icmpge, if_acmpeq, if_acmpne`.
- Сложные условные переходы: `tableswitch, lookupswitch`.
- Безусловные переходы: `goto, goto_w, jsr, jsr_w, ret`.

Переходы происходят по меткам, находящимися перед началом JVM-инструкции.

Вызов метода и инструкция return

Существуют пять инструкций вызова метода: `invokevirtual`, `invokeinterface`, `invokespecial`, `invokestatic`, `invokedynamic`.

Инструкция возврата управления может быть без типа (`return`) или с типом (`xreturn`)

Инструкции с полями

Существуют две инструкции взятия значения из поля и две инструкции сохраняющие в поле: `getfield`, `getstatic`, `setfield`, `setstatic`.

1.3 Пример кода на JVM

Для примера возьмем простую программу на Java и скомпилированный код на JVM.

Листинг 1: Простой Java-класс

```
public class Example {
    private int f = 1;

    public int foo(int p1, int p2) {
        return p1 + p2 - f;
    }

    public static void main() {
        Example ex = new Example();
        int p1 = 1, p2 = 2;
        int a = 3, c = 4;
        if (a < c)  ex.foo(p1, p2);
    }

    public int getF() {
        return f;
    }
}
```

Следующий код на JVM мы получим из скомпилированного класса Example.class с помощью дизассемблера javap. Так выглядит код, который необходимо декомпилировать (см. листинг 2).

Листинг 2: Представление класса в JVM

```
public class Example {
  public Example();
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: aload_0
       5: iconst_1
       6: putfield      #2                  // Field f:I
       9: return

  public int foo(int, int);
       0: iload_1
       1: iload_2
       2: iadd
       3: aload_0
       4: getfield      #2                  // Field f:I
       7: isub
       8: ireturn

  public static void main();
       0: new           #3                  // class Example
       3: dup
       4: invokespecial #4                  // Method "<init>":()V
       7: astore_0
       8: iconst_1
       9: istore_1
      10: iconst_2
      11: istore_2
      12: iconst_3
      13: istore_3
      14: iconst_4
      15: istore        4
```

```
      17: iload_3
      18: iload         4
      20: if_icmpge     30
      23: aload_0
      24: iload_1
      25: iload_2
      26: invokevirtual #5                  // Method foo:(II)I
      29: pop
      30: return

  public int getF();
       0: aload_0
       1: getfield      #2                  // Field f:I
       4: ireturn
}
```

2 Использованные технологии

В данном разделе описаны два использованных в работе инструмента.

2.1 Библиотека ASM

ASM[9] [2] — многофункциональная библиотека для чтения и модификации JVM-классов. Для этого ASM предоставляет инструменты для чтения, записи и изменения байтовых массивов, которыми представлены классы, используя более высокоуровневые понятия: числовые константы, строки, идентификаторы, типы, классы, используемые в Java и т.д.

Библиотека предоставляет два API для создания и изменения скомпилированных классов: API, дающее представление классов, основанное на событиях, и API, основанное на объектах. В нашей работе мы используем первый.

Модель, основанная на событиях, представлена последовательностью событий, каждое из которых представляет элемент класса. Например, название класса, поле, описание метода, инструкция и т.д. Этот API определяет множество событий и порядок, в котором они должны произойти, и предоставляет парсер класса, создающий одно событие на каждый разобранный элемент.

2.2 Jasmin

Jasmin[10] — ассемблер для JVM. На вход ему подаются описания классов, написанные в синтаксисе JVM. Он транслирует их в бинарные файлы, подходящие для исполнения на JVM.

Jasmin может быть использован для создания `class`-файлов, которые нельзя было бы создать, скомпилировав Java-класс.

[9] Официальная страница библиотеки ASM, `http://asm.ow2.org/`

[10] Официальная страница Jasmin - ассемблера для JVM, `http://jasmin.sourceforge.net/`

3 Декомпиляция выражений

Целью курсовой работы является реализация декомпилятора выражений байт-кода JVM. Для этого сначала нужно понять, каким условиям должен удовлетворять код и, как частный случай, выражения на JVM. На код JVM накладываются синтаксические ограничения и ограничения верификации, т.е. проверки корректности класс-файла JVM. Будем рассматривать только такие выражения, которые удовлетворяют этим требованиям.

3.1 Верификация

Поскольку байт-код может быть создан не только компилятором javac, порождающий корректный байт-код, но и другими средствами (библиотеки (ASM, BCEL[11], SER[12] для создания классов из байт-кода, компиляторы других фирм, Java-ассемблеры), необходима проверка корректности или верификация. JVM проводит проверку байт-кода при загрузке в несколько этапов.

На первом проходе происходит загрузка (loading). Здесь проверяется формат класс-файла (magic number, длина).

На втором проходе происходит связывание (linking). На этом проходе проверяется соблюдение правил использования модификатора final, наличие суперкласса (кроме java.lang.Object), формат пул констант и ссылки внутри него.

Третий проход представляет, собственно, верификацию на этапе связывания. Проверяется формат команд байт-кода, аргументы и индексы, аргументы при вызове метода, входы в обработчики исключений (только через исключение) и семантика команд. В проверку семантики команд входит соответствие типов

[11] Официальная страница BCEL. Инструмент для работы с байт-кодом, `http://commons.apache.org/proper/commons-bcel/`

[12] Официальная страница SERP. Инструмент для работы с байт-кодом, `http://serp.sourceforge.net/`

во время присваивания значений полям класса, доступ к локальным переменным в методе и размер стека операндов метода, типы значений в нем. То же проверяется для локальных переменных.

Четвертый (и последний) проход является виртуальным, проходит на этапе выполнения инструкции. В нем проверяется, что тип переменной или сигнатура вызываемого метода совпадают с теми, которые записаны в вызывающем методе компилятором, и флаги доступа вызываемого методу разрешают доступ из вызывающего метода. То же происходит для переменных. На этом проходе также проверяется, что вызываемый методы или переменная, к которой производится доступ, действительно существует в соответствующем классе.

При декомпиляции выражений важно следующее. При разветвлении в коде в конце каждой ветви на стеке должен остаться одинаковый по типу и количеству набор значений. После любого прохода разветвления на стеке остается определенный набор значений. Эти значения могут быть использованы, например, как аргументы метода.

3.2 Виды выражений

Выражение в байт-коде JVM может быть написано без использования переходов или с ними. В последнем случае в разных ветвях выполнения программы могут быть положены на стек JVM несколько значений, а обработка, т. е. снятие их со стека и выполнение некоторой инструкции, может быть выполнена уже после завершения разветвления кода. Поэтому в выражение может входить часть графа потока управления.

3.3 Выражения, содержащие переходы

В случае выражений с переходами необходимо анализировать часть графа потока управления. Есть несколько естественных условий и определенных требований, которые накладываются на код, соответствующий выражению. Граф должен быть ориентированным, без циклов, иметь одну начальную вершину и одну конечную. Каждый узел — участок линейного кода. Ребро — переход из одной линейной части в другую. Из каждого узла выходит не более двух ребер. Действительно, два ребра будет при условном переходе и одно ребро при безусловном (goto). Главное условие — на параллельных участках стек обрабатывается одинаковым образом. В конечной вершине величина стека может быть любой отличной от нуля (иначе это уже не выражение).

Пример выражения с переходами

Ниже приведен пример корректного кода на JVM, который нельзя получить, скомпилировав Java-класс. Для удобства представления на рис. 2 изображен граф потока управления, соответствующий выражению. На нем показаны и пронумерованы участки линейного кода. Всего в графе девять вершин. Видно, что в коде четыре условных перехода (1, 2, 3, 5) и четыре безусловных перехода (4, 6, 7, 8).

До вершины 1, на стек кладется объект класса graph (`28: aload_0`). Потом перед каждым условным оператором кладутся на стек значение типа int для осуществления инструкции сравнения аргумента с нулем. В участках кода, соответствующих вершинам 4, 5 и 6, на стек кладутся два значения типа `int`. В участке 9 вызывается метод `foo` при трёх значениях на стеке: первое из них — объект класса graph, от которого вызывается метод, остальные два значения типа `int`. После вызова метода на стеке остается одно значение.

Листинг 3: Пример кода выражения в JVM с переходами

```
 8: new           graph
11: dup
12: invokespecial graph.<init>()V
15: astore_0
16: iconst_1
17: istore_1
18: iconst_2
19: istore_2
20: iconst_3
21: istore_3
28: aload_0
29: iload         2  ; vertex 1
32: ifgt          58
33: iload_2          ; vertex 2
35: ifle          65
36: iload_2          ; vertex 4
38: iload_3
40: goto          80
58: iload_3          ; vertex 3
60: ifge          82
65: iload_1          ; vertex 5
66: iload_2
67: iload_3
70: ifle          90
80: goto          100 ; vertex 7
82: iload_1           ; vertex 6
84: iload_3
85: goto          90
90: goto          100 ; vertex 8
100: invokevirtual graph.foo(II)I ;vertex 9
```

Этот код действительно нельзя получить из Java, поскольку при вызове ме-

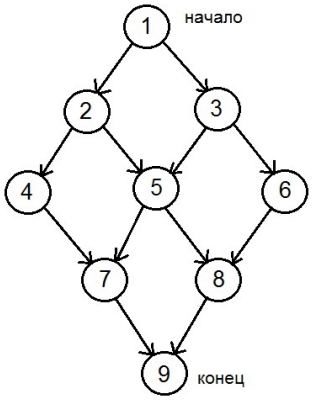

Рис. 2: Граф потока упраления, соответствующий выражению

тода параметры загружаются на стек прямо перед вызовом метода. А в данном примере это происходит гораздо раньше. Мы могли бы получить эквивалентный код в Java только вызвав три раза метод `foo` при разных параметрах в разных ветках условных переходов.

Такой вид выражений требуют глубокого анализа кода. Перед декомпиляцией его необходимо трансформировать в эквивалентный код на JVM, подходящий для промежуточного представления. Декомпиляция такого вида выражений пока не реализована.

3.4 Выражения, не содержащие переходы

Будет рассмотрен разбор участка кода без переходов, который оставляет на стеке после себя одно значение, при нулевом начальном состоянии стека.

Пример выражения без переходов

Код JVM, соответствующий выражению, не содержащий переходов, можно получить из высокоуровневых языков. Например, скомпилировав Java-класс. Ниже в листинге 4 приведен пример кода на Java соответствующий выражению.

Его байт-код (приведен в листинге 5) декомпилируется созданным декомпилятором.

Листинг 4: Пример кода выражения в Java без переходов

```
int c = 1;
int d = 3;
Example ex=new Example();
double b = ((c + d) * 3 - ex.foo(c,d)) / 10 + c*d;
```

Листинг 5: Пример кода выражения в JVM без переходов

```
 0: iconst_1
 1: istore_0
 2: iconst_3
 3: istore_1
 4: new           #3                  // class Example
 7: dup
 8: invokespecial #4                  // Method "<init>":()V
11: astore_2
12: iload_0
13: iload_1
14: iadd
15: iconst_3
16: imul
17: aload_2
18: iload_0
19: iload_1
20: invokevirtual #5                  // Method foo:(II)I
23: isub
24: bipush        10
26: idiv
27: iload_0
28: iload_1
29: imul
```

```
30: iadd
31: i2d
32: dstore_3
33: return
```

Получается следующий код (выбранный целевой язык — Java) при декомпиляции (листинг 6). Этот код совпадает с исходным, что и требуется при построении декомпилятора.

Листинг 6: Пример кода выражения в JVM без переходов

```
int c = 1;
int d = 3;
Example ex = new Example();
double b = ((c + d) * 3 - ex.foo(c,d)) / 10 + c * d;
```

Реализация

В программе декомпилятора используется событийная модель анализа кода из библиотеки ASM. Эта модель создает событие и передает управление в метод, управляющий таким событием, когда при обходе байт-кода встречается определенная языковая конструкция. В этой модели есть `MethodVisitor`, анализирующий методы, и `CLassVisitor`, анализирующий классы. Для декомпиляции выражений используется функциональность `MethodVisitor`:

- `visitInsn` для обработки инструкций без операндов (их большинство);
- `visitIntInsn` для обработки инструкций с одним операндом;
- `visitVarInsn` для инструкций с локальными переменными;
- `visitMethodInsn` для вызовов методов;
- `visitJumpInsn` для обработки инструкций переходов.

Для представления выражения создан абстрактный класс `Expression`, от которого унаследованы разные классы видов выражений: бинарные, унарные, константы, вызовы функций, инструкция `new`. В бинарных и унарных выражениях есть, соответственно, левое и правое подвыражение или одно подвыражение. Они и составляют структуру JVM-дерева.

Для создания JVM-дерева используется стек выражений, получаемый из стека операндов. Каждое значение на стеке операндов — это значение некоторого выражения. Когда встречается инструкция с двумя аргументами, значениями на стеке операндов, тогда снимается два верхних выражения со стека выражений и кладется на стек новое выражение, соответствующее данной инструкции и имеющее в качестве подвыражений предыдущие два. Если встретилась загрузка константы или переменной, то на стек кладется выражение, соответствующие константе или переменной.

При вызове метода со стека снимаются столько значений, сколько аргументов у метода. Далее в зависимости от вида вызова метода на стек кладется либо выражение вызова обычного метода (при этом сначала еще со стека снимается объект, от которого вызывается метод), статического, конструктора или метода суперкласса. Внутри выражений для методов хранятся их аргументы в виде подвыражений.

Из-за того что на стеке операндов в конце рассматриваемого участка кода лежит одно значение, то и на стеке выражений будет храниться одно выражение, соответствующее этому значению.

Заключение

В рамках курсовой работы разработан декомпилятор выражений по байт-коду JVM. При этом декомпилируемый код должен обладать следующими свойствами:

- Код проходит верификацию.
- Часть кода, соответствующая выражению, является линейной без использования условных и безусловных переходов.
- Выражение начинается с пустого стека.
- Выражение заканчивается при глубине стека, равного 1. Это значение и является значением выражения.
- Могут быть использованы арифметические инструкции, загрузка, выгрузка, снятие значения на стеке, вызов метода, инструкции с полями класса.

Для представления выражений создается абстрактное дерево в терминах JVM. В узлах хранятся сущности, соответствующие инструкциям, потомки узлов — подвыражения, соответствующие операндам инструкции.

Список литературы

[1] Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley. The Java Virtual Machine Specification. Java SE 7 Edition, 2013.
docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf

[2] Eric Bruneton. ASM 4.0 A Java Bytecode Engineering Library.
http://download.forge.objectweb.org/asm/asm4-guide.pdf

[3] Robert Wilson, Robert French, Christopher Wilson, Saman Amarasinghe, Jennifer Anderson. The SUIF Compiler System: a Parallelizing and Optimizing Research Compiler. 1994. Computer Systems Laboratory, Stanford University.
http://dl.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=891422/