САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра Системного Программирования

Подкопаев Антон Викторович

Полиномиальной сложности оптимальные принтер-комбинаторы с выбором

Дипломная работа

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

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

Рецензент:
Бреслав А. А.

Санкт-Петербург
2014

SAINT PETERSBURG STATE UNIVERSITY
Mathematics & Mechanics Faculty

Department of Software Engineering

Anton Podkopaev

Polynomial-Time Optimal Pretty-Printing Combinators with Choice

Graduation Thesis

Admitted for defence.
Head of the chair:
professor Andrey Terekhov

Scientific supervisor:
Dmitri Boulytchev

Reviewer:
Andrey Breslav

Saint Petersburg
2014

Оглавление

Введение 4

1. Обзор предметной области 9

1.1. Принтер-комбинаторы 9

1.2. Принтер-комбинаторы с выбором 10

1.3. BURS 12

1.4. Средства форматирования кода в IDE 14

2. Реализация 17

2.1. Оптимальное форматирование как задача BURS 18

2.2. Реализация сведения задачи оптимального форматирования к BURS . . 20

2.3. Расширение принтер-комбинаторов 23

2.3.1. Мемоизация вычислений для поддеревьев 23

2.3.2. Комбинатор `fill` 23

2.3.3. Дополнительная фильтрация вариантов 24

2.3.4. Вставка в шаблон 25

2.4. Принтер-плагин языка Java для IntelliJ IDEA 27

2.4.1. Практические особенности получения и использования шаблонов 28

2.4.2. Обработка списочных структур 29

2.4.3. Обработка комментариев 31

2.4.4. Анализ производительности 32

2.4.5. Открытые проблемы 32

Заключение 33

Список литературы 34

A. Программный код полиномиальной сложности оптимальных принтер-комбинаторов на языке Haskell 36

Введение

В жизненном цикле программного обеспечения важную роль занимает этап поддержки [3]. На этом этапе особую роль приобретает понимание программного текста поддерживаемой системы, что часто бывает достаточно сложной задачей. В данном контексте существенным достоинством программного текста становится его аккуратное оформление. В качестве примера рассмотрим два описания одной и той же функции на языке C (см. рис. 1 и 2).

```
int foo(int k){if(k<1||k>2){printf("out of range\n");
printf("this function requires a value of 1 or 2\n");}else{
printf("Switching\n");switch(k){case 1:printf("1\n");break;case
2:printf("2\n");break;}}}
```

Рис. 1: Неформатированный код

```
int
foo(int k)
{
    if (k < 1 || k > 2) {
        printf("out of range\n");
        printf("this function requires a value of 1 or 2\n");
    } else {
        printf("Switching\n");
        switch (k) {
        case 1:
            printf("1\n");
            break;
        case 2:
            printf("2\n");
            break;
        }
    }
}
```

Рис. 2: Форматированный код (BSD)

С точки зрения компилятора эти описания функции `foo` семантически и синтаксически эквиваленты. Однако, вариант, приведенный на рис. 1, гораздо труднее для понимания, а следовательно и изменения. Как мы видим из этого примера, для возможности дальнейшей поддержки программные тексты должны явным образом отражать структуру программы, что для широкого класса языков программирования означает явное представление структуры синтаксического дерева программы.

Кроме того, распространенной практикой [19], доказавшей свою состоятельность, является использование общепроектного стандарта кодирования. *Стандарт кодиро-*

вания (СК, coding convention) — это набор соглашений, которые используются при написании программного кода. В него входят: способы выбора имен переменных и других идентификаторов, стили отступов при оформлении логических блоков, способы расстановки ограничителей логических блоков (скобок), формат комментариев. СК также призван упростить анализ и изменение программы, поэтому его важно соблюдать, что вводит дополнительные, относительно требований компилятора, ограничения на исходные тексты. СК разных проектов, даже реализуемых на одном и том же языке программирования, могут существенно различаться. Так, код на рис. 2 соответствует СК BSD[1], а на рис. 3 — GNU[2].

```
int
foo (int k)
{
  if (k < 1 || k > 2)
    {
      printf ("out of range\n");
      printf ("this function requires a value of 1 or 2\n");
    }
  else
    {
      printf ("Switching\n");
      switch (k)
    {
    case 1:
      printf ("1\n");
      break;
    case 2:
      printf ("2\n");
      break;
    }
    }
}
```

Рис. 3: Форматированный код (GNU)

Кроме ситуаций, когда СК в проекте поддерживается при написании программного кода вручную, существует другой важный пример использования СК — в языковых процессорах. *Языковой процессор* (ЯП) — это программное средство, принимающее на вход программу в виде текста на некотором языке (программирования, разметки и т. д.) и решающее определенную задачу над этой программой. К языковым процессорам можно отнести компиляторы, суперкомпиляторы, интерпретаторы, средства статического анализа кода, декомпиляторы, средства рефакторинга, средства реинжиниринга, интегрированные среды разработки (IDE) и др.

[1] http://www.freebsd.org/cgi/man.cgi?query=style&sektion=9
[2] http://www.gnu.org/prep/standards/standards.pdf

Первым этапом работы ЯП является *синтаксический анализ*, то есть сопоставление входного текста (линейной последовательности лексем) с формальной грамматикой языка. В результате работы синтаксического анализатора ЯП получает древовидное представление программы, над которым потом происходит основная работа.

Дальше перед большим классом ЯП возникает задача показать пользователю промежуточный или конечный результат обработки кода. Следовательно, необходимо вернуться к текстовому представлению программы, то есть провести процедуру, обратную синтаксическому анализу. Такая задача называется *pretty printing*, а соответствующий инструмент — *pretty printer*. Далее этот инструмент мы будем называть *принтером*.

Сформулируем требования, которые накладываются на принтер. Как уже понятно, принтер должен продуцировать текст, который удовлетворяет проектному СК. Классически считается, что изменение идентификаторов, как и любое другое преобразование абстрактного синтаксического дерева программы, не входит в задачи принтера, поэтому из всех ограничений СК принтеру остается соблюдать только правила форматирования логических структур и отступов. Это требование, при правильном СК, решает проблему наглядности кода в том смысле, что в полученном тексте явным образом отражается логическая структура программы. Кроме того, в большинстве случаев СК оставляет некоторую свободу в представлении синтаксических конструкций, а значит, может быть введено дополнительное ограничение — ограничение на ширину вывода принтера, а среди представлений, подпадающих под ограничения, может быть найдено *оптимальное*. Часто в данном случае под оптимальным представлением понимают тот вариант, который занимает минимальное число строчек, то есть улучшает свойство *обозримости* текста.

С другой стороны, принтер можно рассматривать как функцию, которая на вход принимает абстрактное или конкрентное синтаксическое дерево и дополнительные параметры, а на выходе выдает текст. В случае, если на вход принтеру подается конкретное синтаксическое дерево, то есть дерево разбора, принтер может быть частичной функцией по отношению к синтаксическим конструкциям языка. Такое ослабление связано с тем, что для конструкций, на которых принтер неопределен, возможно использование текстового представления, заложенного в дерево разбора.

Рассмотрим небольшой пример. Пусть в СК задано условие вида: "последовательные операторы пишутся на одной строке, если помещаются в N символов, а иначе — на разных строках". На рис. 4 изображено синтаксическое дерево последовательности двух операторов. Такое дерево, согласно заданному правилу, может быть напечатано одним из двух вариантов (рис. 5, 6), выбор происходит в зависимости от ширины вывода. Так, при ширине, равной 35 символов (длина строки «`System.out.println("Hello world");`»), должен выбираться вариант, изображенный на рис. 6, так как код на рис. 5 имеет ширину более 35 символов. Могут быть за-

даны и более сложные условия.

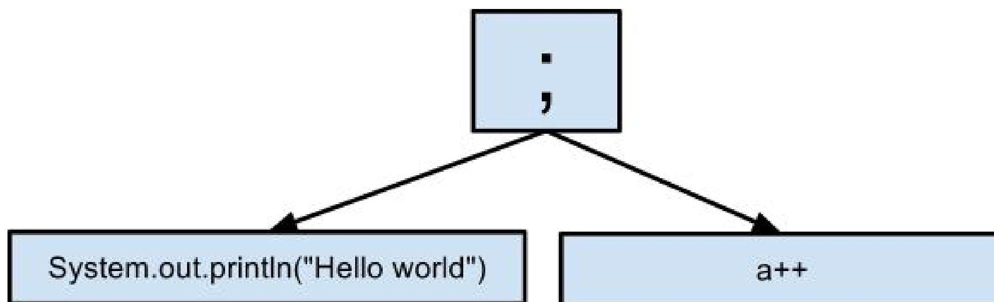


Рис. 4: Последовательные операторы

```
System.out.println("Hello world"); a++
```

Рис. 5: Последовательные операторы в строчку

```
System.out.println("Hello world");
a++
```

Рис. 6: Последовательные операторы в несколько строк

Рассмотрим другой пример. Пусть нам нужно текстовое представление синтаксического дерева конструкции “`if`”, и заданы шаблоны с рис. 7a и 7b. При этом вариант, изображенный на рис. 7a выбирается только в случае, если условие и ветки могут быть напечатаны в одну строчку.

```
if 'expression' then 'branch 1'
                else 'branch 2'
```

(a)

```
if 'expression'
then 'branch 1'
else 'branch 2'
```

(b)

Рис. 7: Представления для конструкции “`if`”

Тогда для деревьев, представленных на рис. 8a и 9a, будут напечатаны тексты, представленные на рис. 8b и 9b соответственно.

Существует несколько подходов к написанию принтеров. Классическим способом задания принтеров в функциональных языках программирования являются принтер-комбинаторы [4–6, 10, 14, 16, 20, 21, 23]. Кроме того, существует работа, посвященная совместной разработке принтера и синтаксического анализатора [18]. В [12, 13, 22] описаны методы создания принтеров с помощью специализированных грамматик.

К сожалению, принтеры, создаваемые с помощью упомянутых подходов, слабопараметризуемы и сложны, поскольку приведенные подходы недостаточно декларатив-

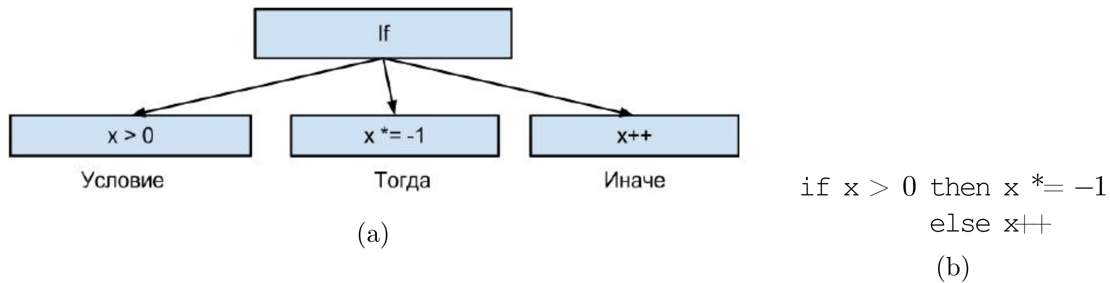


Рис. 8: Использование представления с рис. 7a

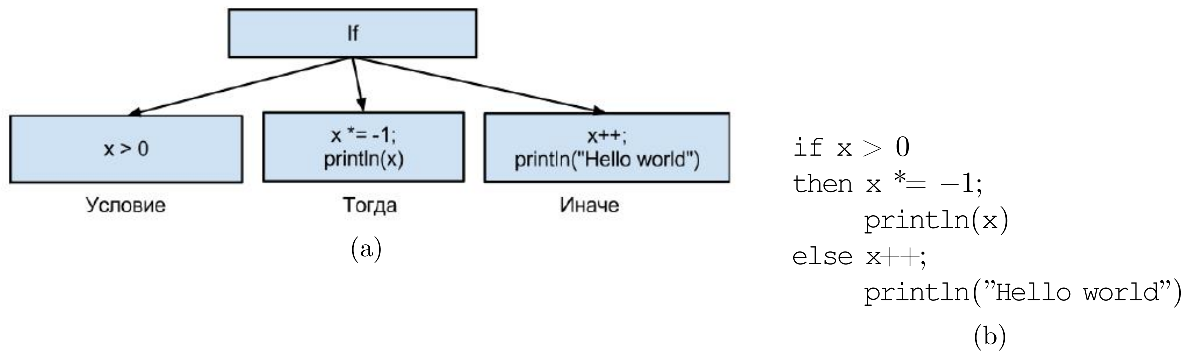


Рис. 9: Использование представления с рис. 7b

ны. Чтобы избежать подобных проблем, принтеры можно задавать с помощью шаблонов, то есть примеров представления для синтаксических конструкций, подобных приведенным на рис. 7. Принципиальная возможность осуществления такого способа была доказана в рамках предыдущей работы автора [1]. В ней был разработан обобщенный принтер для учебного языка L, использующий расширяемый синтаксический анализатор целевого языка для получения шаблонов.

Целью данной работы является разработка эффективной принтер-комбинаторной библиотеки, которая может быть использована при создании принтеров, задаваемых шаблонами, а также апробация самого подхода описания принтеров с помощью шаблонов для языка Java.

1. Обзор предметной области

Одним из подходов к заданию принтеров являются принтер-комбинаторы. В этой главе приведен обзор классических принтер-комбинаторов, а также принтер-комбинаторов с дополнительным оператором *выбора*, которые в рамках данной работы использованы для реализации принтеров, задаваемых с помощью шаблонов. Для решения проблем эффективности комбинаторов с выбором в данной работе был использован подход BURS. Апробация подхода задания принтеров с помощью шаблонов была произведена для языка Java, для которого есть средства форматирования, встроенные в IDE.

1.1. Принтер-комбинаторы

Базовыми принтер-комбинаторными библиотеками являются библиотеки Джона Хьюза [10] и Филиппа Вадлера [23], которые представляют собой функциональную переработку алгоритма Оппена [15]. Ограничимся рассмотрением библиотеки Вадлера, так как библиотека Хьюза обладает теми же свойствами, которые принципиальны в контексте данной работы.

В библиотеке ключевым типом является документ. Он представляет сущность, которая потом может быть переведена в строковое представление алгоритмом принтера. Основные конструкторы для составления документа таковы:

- атомарная строка, которая печатается как есть;
- *разделитель*;
- последовательная композиция двух документов;
- набор связанных документов.

Определяющей особенностью данного подхода является то, что все разделители в рамках одного набора могут быть совместно заменены алгоритмом принтера на пробельный символ или на перевод строки. Выбор для каждого набора разделителей основывается на том, что вывод должен поместиться в заданную ширину, используя минимальное число строк.

Основная проблема такого подхода заключается в его слабой выразительной силе. Документы, построенные по синтаксическому дереву печатаемой программы, обрабатываются слишком единообразно, что иногда приводит к нежелательному результату. Пусть, к примеру, нужно напечатать программу на языке Python[3]. Между последовательными операторами в случае их печати на одной строчке необходимо добавить дополнительный разделитель (“;”), иначе программа станет некорректной (см.

[3] http://python.org

рис. 10). Однако, описанные принтер-комбинаторы не предоставляют возможности задать такое поведение. Кроме того, с помощью таких принтер-комбинаторов невозможно выразить разные проектные СК, так как они всегда печатают текст в одном стиле, который жестко "зашит" в их код.

```
print 5
print 6
```

(a) Корректный код

```
print 5 print 6
```

(b) Некорректный код

Рис. 10: Пример работы принтер-комбинаторов для языка Python

Более подробный обзор библиотек Хьюза и Вадлера представлен в [1].

Большинство принтер-комбинаторных библиотек [6, 14, 16, 20, 21] являются развитиями работ Хьюза и Вадлера. Так, в отличие от базовых, среди них есть реализации с линейной сложностью обработки документа от его размера и *online* алгоритмы, которые не требуют просмотра всего документа для начала печати его текстового представления. Но все они обладают тем же интерфейсом, а значит в них также неразрешимы задачи, в которых требуется задавать варианты текстовых представлений, которые отличаются не только пробелами и переводами строк.

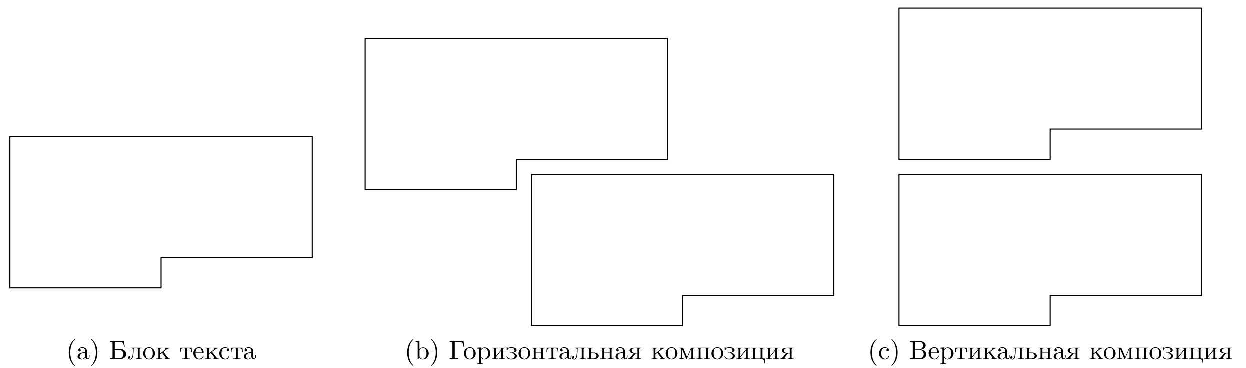
1.2. Принтер-комбинаторы с выбором

Существенно от описанных выше отличаются библиотеки, предоставляющие в своем интерфейсе комбинатор *выбора*, который позволяет задавать для одного поддерева принципиально разные варианты раскладок. Так, в работах [12, 13] используется оператор `ALT`, но алгоритм принтера устроен так, что среди двух альтернатив выбирается первая, если она помещается в заданную ширину, и вторая — иначе, что не дает оптимальный результат на выходе.

Оптимальные принтер-комбинаторы с выбором были впервые представлены в работе [4]. Их реализация является частью Utrecht Tools Library[4] (практическая реализация несколько изменена по отношению к той, что описана в статье, но отличие несущественно). В данном подходе текст строится из блоков прямоугольной формы с возможно неполной последней строчкой (см. рис. 11a). В реализации на Haskell блоки представляются структурой `Format`:

```
data Format = Elem { height        :: Int
                   , lastLineWidth :: Int
                   , width         :: Int
                   , txtstr        :: Int → String → String
                   }
```

[4] http://www.cs.uu.nl/wiki/HUT/WebHome



(a) Блок текста (b) Горизонтальная композиция (c) Вертикальная композиция

Рис. 11: Примитивы блока текста

Первые три поля структуры определяют геометрические размеры блока, а последнее — функция, которая используется для преобразования блока в текст. Здесь используется функция, а не просто строчка, чтобы можно было преобразовывать вложенные блоки за линейное время. Первый аргумент `txtstr` задает сдвиг блока.

Для работы с блоками текста используется следующие четыре примитива:

```
s2fmt     :: String → Format
indentFmt :: Int → Format → Format
aboveFmt  :: Format → Format → Format
besideFmt :: Format → Format → Format
```

Функция `s2fmt` создает блок текста, состоящий из одной строчки; `indentFmt` по блоку создает новый, сдвинутый на заданное число позиций. Действие примитивов композиции `besideFmt` и `aboveFmt` показано на рис. 11b и 11c соответственно.

Также, как и в библиотеках без комбинатора выбора, в работе [4] используется понятие *документа*. Здесь документ можно рассматривать как множество возможных раскладок (набор `Format`-элементов). Документы описываются типом `Doc`, экземпляры которого на этапе построения представляют собой деревья применений приведенных ниже комбинаторов, а на этапе обработки алгоритмом принтера им в соответствие ставится итоговое множество вариантов текстовых представлений.

Документ конструируется с помощью следующих комбинаторов, которые симметричны примитивам построения блоков текста:

```
text   :: String → Doc
indent :: Int → Doc → Doc
beside :: Doc → Doc → Doc
above  :: Doc → Doc → Doc
```

В дополнение появляется пятый комбинатор для документов:

```
choice :: Doc → Doc → Doc
```

Этот комбинатор и является тем самым комбинатором выбора. Он представляет *объединение* множеств раскладок документов, которые были переданы как аргументы

комбинатора. Заметим, что только этот комбинатор может произвести документ с несколькими раскладками из одновариантных аргументов.

Оригинальная реализация существенно опирается на ленивые вычисления. В [4] множество вариантов, соответствующее экземпляру `Doc`, представляется ленивым списком всех возможных раскладок, удовлетворяющих ограничению на максимальную ширину. Этот список отсортирован в порядке "ухудшения" раскладок, то есть в голове списка лежит "лучшая", в терминах оптимальности, раскладка из возможных при заданной ширине. В случае `beside`- и `above`-композиций документов полная (без учета ленивости) сложность вычисления списка нового документа составляет $O(n \times m)$, где n и m — размеры списков, соответствующих соединяемым документам. Размер нового списка также порядка $n \times m$. Он не обязательно точно равен $n \times m$, так как некоторые из полученных представлений могут не подпадать под ограничение на ширину, соответственно их можно отбросить на этапе построения нового списка.

Выбор лучшего представления для самого верхнеуровнего документа происходит просто — нужно из соответствующего списка взять первый элемент. Это создает впечатление, что общее число операций, благодаря ленивым вычислениям, существенно уменьшается. Но это не так из-за реализации обработки документа, построенного с помощью комбинатора `beside`, которая вынуждает полное вычисление дочерних списков. Более того, из-за свойств отношения порядка, построенного по критерию оптимальности, в рамках данной модели списков принципиально нельзя построить ленивую обработку комбинатора `beside`. Так, выбор оптимальной раскладки документа в [4] имеет в худшем случае экспоненциальную сложность от числа комбинаторов, использованных при его построении. В [5] приведены оптимизации для данной модели, но они не улучшают асимптотику решения для деревьев в общем случае.

1.3. BURS

Bottom-Up Rewrite System (BURS) [2] — это метод динамического программирования на деревьях, изначально появившийся в контексте задачи выбора инструкций для генерации машинного кода. Основой BURS является регулярная грамматика с древовидными правилами [7], для которых задана стоимость применения подстановки, то есть грамматика со следующим набором правил:

$$\begin{array}{lcll} N & : & \alpha & [c] \\ N & : & \alpha\,(K_1, \ldots, K_n) & [c] \end{array}$$

Здесь N, K_i это нетерминалы, α — терминал, c — функция стоимости, заданная для каждого правила. Как и для обычной линейной грамматики, вводится стартовый нетерминал S. Считается, что терминальное дерево выводится в данной грамматике, если его можно получить с помощью правил подстановки из одноузлового дерева

S. Каждая подстановка заменяет нетерминал N, находящийся в листе дерева, на дерево $\alpha\,(K_1, \ldots, K_n)$, если в грамматике есть правило $N : \alpha\,(K_1, \ldots, K_n)$. Для каждой подстановки вычисляется стоимость ее применения с помощью функции стоимости c. Аргументами функции могут служить терминальная метка α и стоимости вывода поддеревьев. Задачей, которую решает BURS, является поиск вывода наименьшей стоимости для заданного дерева по заданной грамматике. Такой вывод может быть найден двухпроходным алгоритмом.

Первый проход (*пометка*) обрабатывает дерево снизу вверх и вычисляет для каждого узла набор троек $(K,\ R,\ c)$, где K — нетерминал, из которого может быть выведено поддерево с корнем в обрабатываемом узле, R — первое правило, которое используется для вывода минимальной стоимости из K, c — стоимость такого вывода. Процесс пометки происходит следующим образом:

- для листовой вершины, помеченной терминалом α, в множество троек этого вершины добавляется $(K,\ R,\ c\,(\alpha))$ для каждого правила $R = K :\ \alpha\ [c]$;
- для промежуточной вершины, помеченной терминалом α, с непосредственными поддеревьями $v_1, \ldots, v_n$ в множество добавляется тройка $(K,\ R,\ c\,(\alpha, c_1, \ldots, c_n))$ для каждого правила $R = K :\ \ \alpha\ (K_1, \ldots, K_n)\ [c]$, где $(K_i,\ R_i,\ c_i)$ входит в множество троек для v_i; если есть несколько правил вывода из нетерминала K, то выбирается правило, минимизирующее стоимость вывода.

Второй проход (*свертка*) просматривает дерево сверху вниз, используя сделанные пометки. Первое правило из минимального вывода определяется тройкой $(S,\ R,\ c)$ для корневого узла (если такой тройки нет, то вывод из S невозможен). Это правило однозначно определяет нетерминалы K_i для каждого непосредственного поддерева и процесс повторяется.

Для пометки дерева потенциально необходимо каждое правило грамматики применить к каждому узлу. При фиксированной грамматике алгоритмическая сложность первого прохода — $O\,(|R|)$, где $|R|$ — количество правил (размер множества троек для каждого узла ограничен числом нетерминалов, которое не больше, чем количество правил). Свертка также имеет линейную сложность.

1.4. Средства форматирования кода в IDE

Интегрированные среды разработки программного обеспечения (IDE) предоставляют средства для форматирования программного кода. Рассмотрим их на примере двух Java IDE — IntelliJ IDEA[5] и Eclipse[6]. Для задания требуемого СК используется широкий набор настроек *форматтера*, подпрограммы IDE, отвечающей за форматирование исходных текстов. Примерами таких настроек являются:

- помещать ли фигурную скобку на той же строке, что и предыдущее выражение;
- форматирование списков — всегда на одной строчке, переводить строчку после каждого элемента или печатать на одной строке, пока строчка меньше заданной рекомендуемой ширины.

На рис. 12 и 13 приведены диалоги задания параметров форматирования в IntelliJ IDEA и Eclipse соответственно.

У встроенных форматтеров есть следующие особенности. Во-первых, это невозможность выразить нестандартный СК, так как настройки форматтеров, несмотря на их большое количество, дают ограниченную вариативность для текстовых представлений форматируемых программ. В целом, количество принципиально разных стилей форматирования, которые можно задать с помощью данных настроек, невелико. Во-вторых, в случае, если надо придерживаться СК уже существующего проекта, то по коду этого проекта необходимо вручную задать все настройки форматтера. Причем наборы настроек в разных IDE не совпадают, что увеличивает сложность поддержки единого СК, если разработчики используют отличные от друг друга IDE. Для решения данной проблемы существуют плагины, позволяющие использовать внешние фоматтеры[7]. Кроме того, есть средства, позволяющие экспортировать настройки форматтера в XML-файл для их использования в другой IDE[8].

Важным достоинством описанных форматтеров является малое время работы. Это достигается в том числе и за счет того, что часто форматируется не весь файл, а только его часть (см. [11]), и из-за детерминированности представлений узлов синтаксического дерева, которая следует из специфики настроек форматтера. Однако форматирование целого большого проекта занимает существенное время. Например, обработка кода IntelliJ IDEA Community Edition занимает более часа у встроенного в IDEA форматтера.

[5]http://jetbrains.com/idea/

[6]http://eclipse.org

[7]http://plugins.jetbrains.com/plugin/6546

[8]http://blog.jetbrains.com/idea/2014/01/intellij-idea-13-importing-code-formatter-settings-from-eclipse/

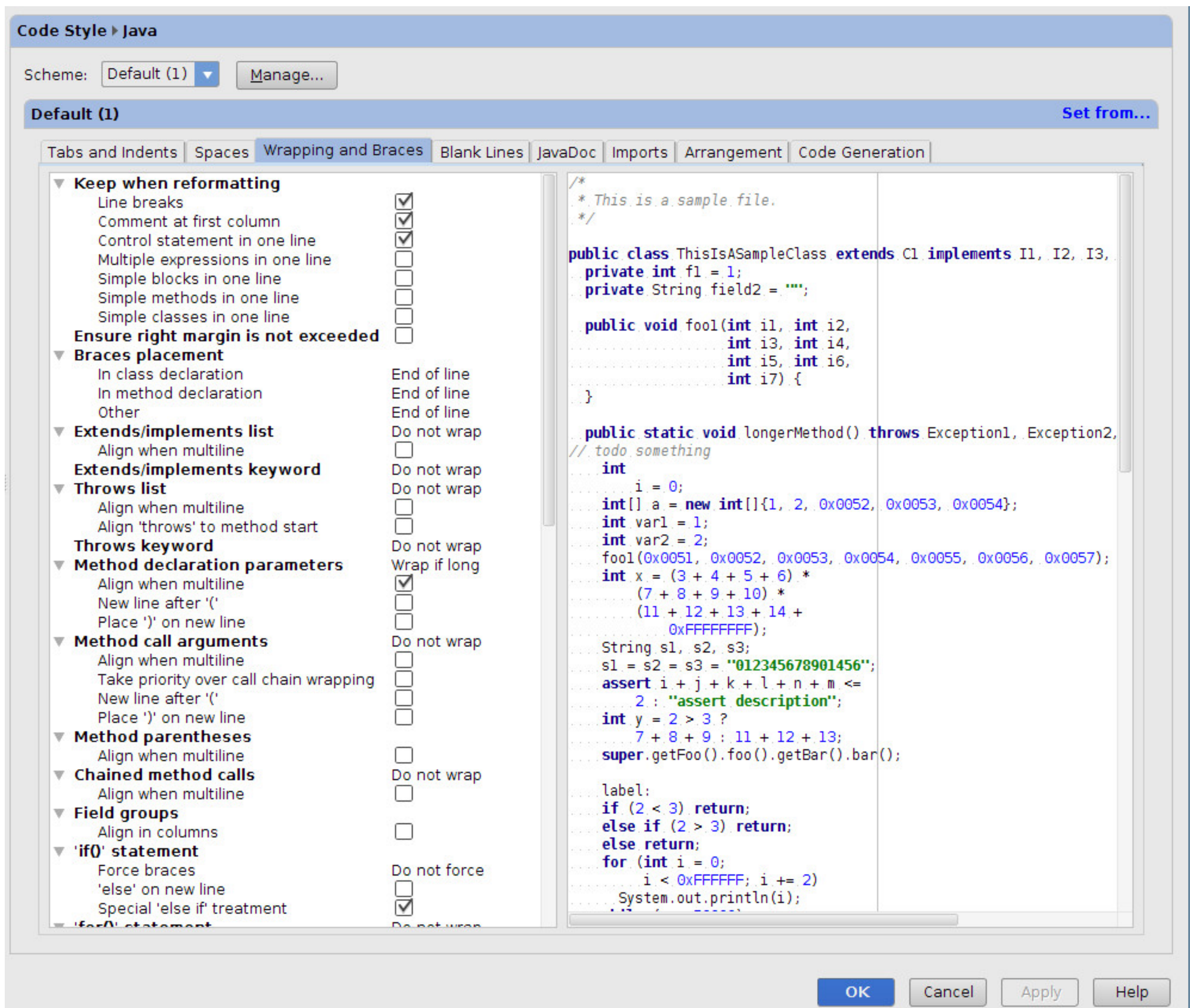


Рис. 12: Окно настройки форматтера IntelliJ IDEA

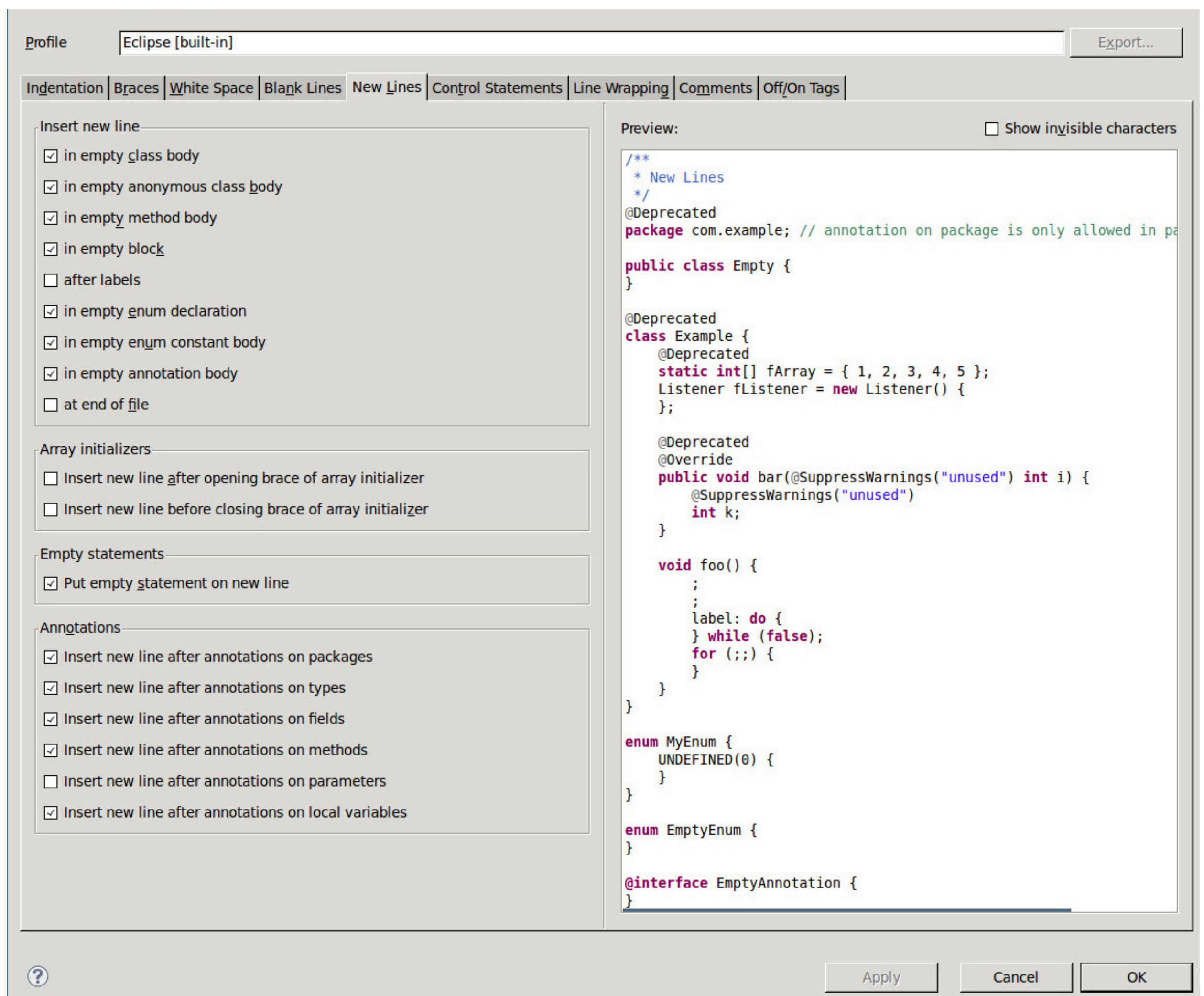


Рис. 13: Окно настройки форматтера Eclipse

2. Реализация

Контекстом данной работы является разработка метода задания принтеров с помощью шаблонов. Под *шаблоном* здесь и далее понимаются данные, сопоставляя которые с переданным на печать синтаксическим деревом, мы будем получать текстовое представление для этого дерева. Более точно, каждый конкретный шаблон позволяет построить текстовые представления для подходящего по типу узла синтаксического дерева, используя уже полученные представления дочерних поддеревьев.

Фактически в реализации шаблон — это размеченное дерево разбора некоторой синтаксической конструкции, в котором по каждой метке можно выяснить ограничения на представление соответствующих поддеревьев и описание, каким образом нужная раскладка поддерева должна быть вставлена в шаблон. При этом сами шаблоны получаются из некоторого эталонного репозитория с исходном кодом на целевом языке определяемого принтера. Так, к примеру, принтер будет использовать форматирование GNU, если ему на вход передать один или несколько проектов в СК GNU.

Кроме того, полученный таким образом принтер должен выдавать оптимальное представление для синтаксического дерева, переданного на печать. На нижнем уровне в принтере используются оптимальные принтер-комбинаторы с выбором. Здесь комбинатор выбора нужен для возможности задания вариантов представления, соответствующих разным шаблонам. Недостатком существующих оптимальных принтер-комбинаторов с выбором [4] является их экспоненциальная сложность. Поэтому для возможности использования того же комбинаторного интерфейса в рамках данной работы с помощью BURS была разработана полиномиальная версия этих комбинаторов [17].

Для апробации метода в случае языка Java был разработан принтер-плагин для IntelliJ IDEA, что позволило переиспользовать синтаксический анализатор Java для получения шаблонов. Разработка велась на языке Kotlin. Kotlin[9] — это функциональный, объектно-ориентированный, компилируемый в JVM-байткод и JavaScript язык, разрабатываемый компанией JetBrains[10]. Kotlin был выбран для реализации принтер-плагина к IntelliJ IDEA по нескольким причинам. Во-первых, Kotlin обладает хорошей интеграцией с Java, что позволяет использовать его с IDEA API. Во-вторых, функции в Kotlin являются объектами первого рода, что позволяет легко реализовывать комбинаторные библиотеки на нем. На данный момент Kotlin находится в стадии разработки, поэтому периодически возникают проблемы с тем, что исходные коды перестают быть совместимыми с новыми версиями языка, но обычно требуется внести небольшой набор исправлений для восстановления работоспособности.

[9] http://kotlin.jetbrains.org/
[10] http://jetbrains.org/

2.1. Оптимальное форматирование как задача BURS

Для снижения алгоритмической сложности задачи поиска оптимального представления документа, заданного комбинаторами из [4], можно произвести сведение к BURS. Сведение основано на следующих наблюдениях. Пусть w есть максимальная допустимая ширина вывода. Поскольку каждая возникающая раскладка представляется блоком текста, для поиска оптимального представления можно рассматривать все промежуточные раскладки как тройки (n, k, h), где $n \leq w$ — это общая ширина раскладки, $k \leq n$ — ширина ее последней строчки, h — ее высота. Тогда очевидно, что при $h_1 \leq h_2$ между (n, k, h_1) и (n, k, h_2) первая тройка предпочтительней на любом этапе вычислений. Таким образом для каждого узла не может быть более w^2 существенных представлений[11].

Документ, для которого мы ищем раскладку, можно рассматривать как дерево, построенное из примитивов `text`, `indent`, `beside`, `above` и `choice`. Тогда задачу раскладки документа можно решать независимо для поддеревьев, а потом объединять решения при учете, что для каждого поддерева и каждой пары $(n, k), k \leq n \leq w$ запоминается такая минимальная h, что для поддерева существует текстовое представление с размерами (n, k, h). Тогда, совершив обход дерева снизу вверх, мы сможем вычислить оптимальное представление для всего дерева. Полезно заметить, что оптимальное представление дерева не всегда получается из оптимальных представлений его поддеревьев, поскольку при использовании поддеревьев возникают дополнительные ограничения на ширину раскладок этих поддеревьев.

Приведенные наблюдения можно формализовать в виде BURS-задачи. Для заданной ширины вывода w введем набор нетерминалов T_n^k, для всех $k \leq n \leq w$. Определим BURS-грамматику так, чтобы цена вывода h нетерминала T_n^k для документа соответствовала его раскладке с параметрами (n, k, h). Для такой BURS-грамматики этап разметки посчитает все существенные раскладки, а этап свертки вернет оптимальную. Определим правила переписывания для этой грамматики:

1. Для терминального узла [`text s`][12] существует два варианта:

    - Если $|s| \leq w$ (где $|s|$ — это длина строки s), то вводится единственное правило $T_{|s|}^{|s|}$: [`text s`] со стоимостью 1; для всех остальных $k, n \neq |s|$ используется T_n^k : [`text s`] со стоимостью ∞;
    - Если $|s| > w$, то вводится правило T_n^k : [`text s`] со стоимостью ∞ для всех k, n.

[11] На самом деле, из-за ограничения $k \leq n$ максимальное число раскладок не w^2, а $\frac{w^2+w}{2}$, но это несущественно.

[12] Квадратные скобки используются для обозначения терминалов, состоящих из одного или нескольких символов.

Действительно, раскладка документа, состоящего из строчки длины $|s|$, может быть только размеров $(|s|, |s|, 1)$. Все остальные размеры недоступны, поэтому имеют стоимости ∞.

2. Для узла [indent m] введем два набора правил:

   (a) T^{k+m}_{n+m} : [indent m] (T^k_n) со стоимостью, равной стоимости раскладки поддерева m в T^k_n, для всех n и k таких, что $n + m \leq w$ и $k \leq n$;

   (b) T^k_n : [indent m] (T^i_j) со стоимостью ∞ в противном случае.

   Понятно, что сдвиг раскладки с параметрами n, k и h на m позиций вправо создает раскладку с параметрами $n + m$, $k + m$, h. Такая раскладка допустима, если $n + m \leq w$ и $k + m \leq w$.

3. Для узла [above] вводим правило $T^{k_2}_{\max(n_1,n_2)}$: [above] $(T^{k_1}_{n_1},\ T^{k_2}_{n_2})$ со стоимостью, равной сумме стоимостей вывода поддеревьев, для всех $k_1 \leq n_1 \leq w$ и $k_2 \leq n_2 \leq w$.

   Действительно, при вертикальном соединении раскладок с параметрами n_1, k_1, h_1 и n_2, k_2, h_2 мы получаем раскладку с размерами $\max(n_1, n_2)$, k_2, $h_1 + h_2$. Вертикальная композиция допустимых раскладок всегда допустима.

4. Для узла [beside] вводим правило $T^{k_1+k_2}_{\max(n_1,\, k_1+n_2)}$: [beside] $(T^{k_1}_{n_1}, T^{k_2}_{n_2})$ для каждой комбинации n_1, n_2, k_1, k_2 такой, что $k_1 + k_2 \leq \max(n_1,\ k_1 + n_2) \leq w$. Стоимость такого вывода равна сумме стоимостей вывода поддеревьев минус 1. Это может быть легко проверено из геометрических соображений.

5. Для узла [choice] вводим правило T^k_n : [choice] $(T^k_n,\ T^k_n)$ для всех $k \leq n \leq w$.

   Цена есть минимум среди цен вывода для поддеревьев. Понятно, что среди двух раскладок с одинаковыми ширинами выбирается та, что имеет меньшую высоту.

Для завершения описания грамматики необходимо ввести правило для стартового нетерминала S. Можно добавить правило $S : r$ с тождественной функцией цены для любого r, или ввести цепное правило $S : T^k_n$ с такой же функцией для всех нетерминалов T^k_n (такое правило требует несущественного расширения определения BURS).

Число нетерминалов в определенной выше грамматике равно w^2. Однако, число правил есть $O(w^4)$, так как в дереве есть узлы со степенью 2. Так, приведенная BURS-реализация оптимального принтера работает за линейное время от числа узлов в дереве документа для фиксированной ширины вывода, при этом сложность от ширины вывода есть $O(w^4)$. Понятно, что данное сведение может быть выполнено и для других примитивов построения дерева, которые могут иметь большую степень, но ценой экспоненциального роста сложности по w.

2.2. Реализация сведения задачи оптимального форматирования к BURS

В рамках данной работы была реализована принтер-комбинаторная библиотека на языке Haskell[13] (см. приложение A) с тем же набором комбинаторов, что и в [4], которая линейна относительно входного документа и полиномиальна относительно максимальной ширины вывода за счет описанного выше сведения задачи к BURS. Данная реализация использует низкоуровневые типы и функции из оригинальной библиотеки [4], в то время как высокоуровневые типы и комбинаторы переопределены.

В реализации нет непосредственного использования BURS: в явном виде не задается BURS-грамматика, нет набора нетерминалов и стандартного алгоритма обработки. Вместо этого для каждого узла документа вычисляется ассоциативный массив, который ставит в соответствие паре (n, k) лучший (минимальный по числу строк) блок текста с параметрами ширин n и k. Так, в ассоциативном массиве записаны пары $((n, k), f)$, где f — это блок текста, соответствующий оптимальному представлению с ширинами (n, k). Значение высоты f есть минимальная стоимость вывода нетерминала T_n^k для данного узла. Заметим, что поскольку f является конечным результатом для T_n^k, а не просто последним правилом, которое нужно применить для построения ответа, то отменяется необходимость применения в дальнейшем алгоритма свертки. В конце, для корня дерева, представляющего документ, из ассоциативного массива выбирается элемент минимальной стоимости.

Наибольший интерес в рамках данного исследования представляет поведение библиотеки в худшем случае. Реализация была апробирована на документах, полученных с помощью функции генерации, приведенной на рис. 14. Они достаточно *“плохи”*, поскольку активно используют комбинаторы `beside` и `choice`, что должно приводит к экспоненциальной зависимости числа нефакторизованных раскладок от размера дерева комбинаторов. В функции генерации используются операторы (>|<), (>–<), (>//<), которые являются инфиксными синоминами для комбинаторов `beside`, `above` и `choice` соответственно.

Результаты времени исполнения для сравнения авторской реализации с оригинальной библиотекой приведены в таблице 1. Из приведенных данных видно, что, начиная с некоторой комбинации ширины вывода и размера документа, библиотека [4] не может вычислить раскладку документа. Значения, записанные подобно “-(> 59)”, обозначают, что после указанного времени процесс аварийно завершился с переполнением стека.

Авторская реализация часто не показывает линейное поведение, как это ожидается при фактическом учетверении числа узлов в документе от строчки к строчке. Дополнительные эксперименты показали связь данного поведения с разреженностью

[13] `http://github.com/anlun/polynomialPPCombinators`

```
data Tree = T String
          | Node Tree Tree

treeToDoc :: Tree → Doc
treeToDoc (T s)         = text s
treeToDoc (Node lt rt) = besideVariant >//< aboveVariant
  where
    p    = text "–"
    lft = treeToDoc lt
    rft = treeToDoc rt
    besideVariant = p >|< lft >|< rft
    aboveVariant  = p >−< lft >−< rft
```

Рис. 14: Функция генерации деревьев для тестов производительности

ассоциативного массива для больших ширин. Другими словами, для маленьких деревьев число значений в соответствующем ассоциативном массиве сильно меньше верхней границы. С ростом размера дерева число записей растет быстрее, чем с линейной скоростью, пока не будут достигнуты значения, близкие к верхней границе.

Кроме того, особенностью описанной реализации является то, что она не использует ленивые вычисления для ускорения построения оптимального текстового представления для поданного на вход документа, а, наоборот, форсирует большинство вычислений, что делает ее легко переносимой на языки без встроенной ленивости.

Число узлов	Оригинальная	Авторская
10921	0.07	0.18
43689	1.28	0.73
174761	6.47	2.86
699049	18.46	11.58
2796201	46.54	47.52
11184809	98.85	187.90

(a) Ширина вывода 25

Число узлов	Оригинальная	Авторская
10921	0.12	0.69
43689	287.55	4.03
174761	- ($>$294)	17.93
699049	- ($>$284)	71.70

(b) Ширина вывода 50

Число узлов	Оригинальная	Авторская
10921	0.06	0.97
43689	97.79	14.92
174761	- ($>$179)	88.71

(c) Ширина вывода 100

Число узлов	Оригинальная	Авторская
10921	0.06	1.01
43689	38.99	20.75
174761	- ($>$59)	204.51

(d) Ширина вывода 150

Таблица 1: Время вычисления раскладки (в секундах)

2.3. Расширение принтер-комбинаторов

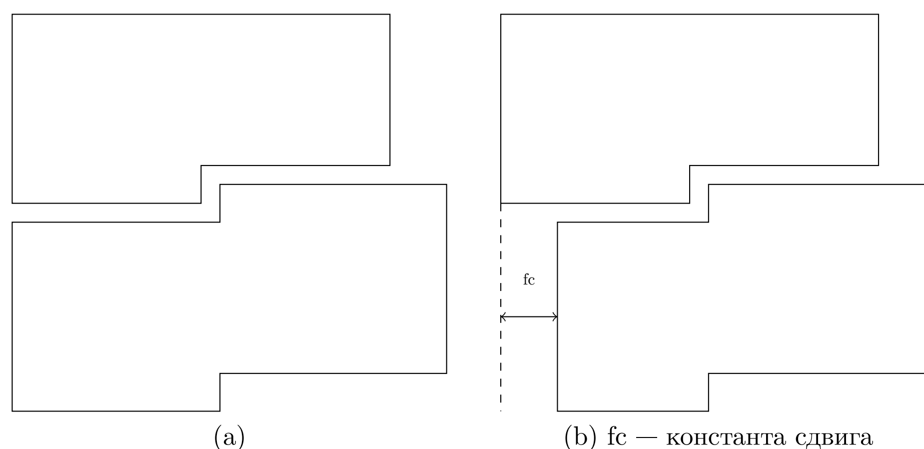
Для дальнейшего использования в принтер-плагине авторская библиотека комбинаторов была переписана на язык Kotlin. Эта реализация расширяет набор комбинаторов и использует дополнительные техники, которые важны при практическом применении библиотеки в контексте принтеров, задаваемых шаблонами.

2.3.1. Мемоизация вычислений для поддеревьев

В реальных принтерах документы, состоящие из комбинаторов `text`, `indent`, `beside`, `above` и `choice`, чаще всего представляют собой не дерево, а *дэг* (DAG, Direct Acyclic Graph). Поэтому при наивной реализации можно получить экспоненциальную сложность от размеров документа, поскольку соответствующее дерево будет иметь экспоненциальный размер от размера дэга. Чтобы избежать этой проблемы достаточно завести ассоциативный массив, который будет хранить посчитанный набор раскладок по конкретному поддереву. Кроме того, при наивной реализации данной оптимизации алгоритм вычисления раскладки становится квадратичным от размера дэга, так как вычисление хэша, используемого ассоциативным массивом, для дерева занимает линейное время. Чтобы преодолеть и эту сложность, в дереве хранится вычисленный хэш.

2.3.2. Комбинатор `fill`

Для реализации принтера, использующего шаблоны, в библиотеку необходимо добавить дополнительный принтер-комбинатор — комбинатор `fill`.



(a) (b) fc — константа сдвига

Рис. 15: Оператор Fill

В качестве примера, показывающего необходимость такого комбинатора, рассмотрим печать метода из языка Java. Пусть нам нужно получить на выходе следующий программный код:

```
public static void main(String[] args) {
  System.out.print("Hello␣");
  System.out.print("World!");
}
```

По описанному раннее подходу, для построения общей раскладки необходимо иметь представления поддеревьев. В частности, представление тела метода:

```
{
  System.out.print("Hello␣");
  System.out.print("World!");
}
```

Тогда для реализации соответствующего соединения, представленного ниже, необходим комбинатор `fill`:

```
public static void main(String[] args) *
***************************************
***************************************
*
```

К сожалению, введение нового комбинатора накладывает дополнительные ограничения на факторизацию. Это связано с тем, что теперь по парам (n, k), где n — общая ширина, а k — ширина последней строчки, нельзя однозначным образом выдать соответствующую пару для раскладки соединения двух блоков. Так, при применении `fill`-комбинатора важной становится и ширина первой строчки нижнего блока, поскольку сумма ее и ширины последней строчки верхнего блока могут дать максимум общей ширины. Таким образом, у нетерминалов из BURS-сведения появляется дополнительный индекс, а их число увеличивается с w^2 до w^3. Аналогично, количество правил перехода возрастает с $O(w^4)$ до $O(w^6)$.

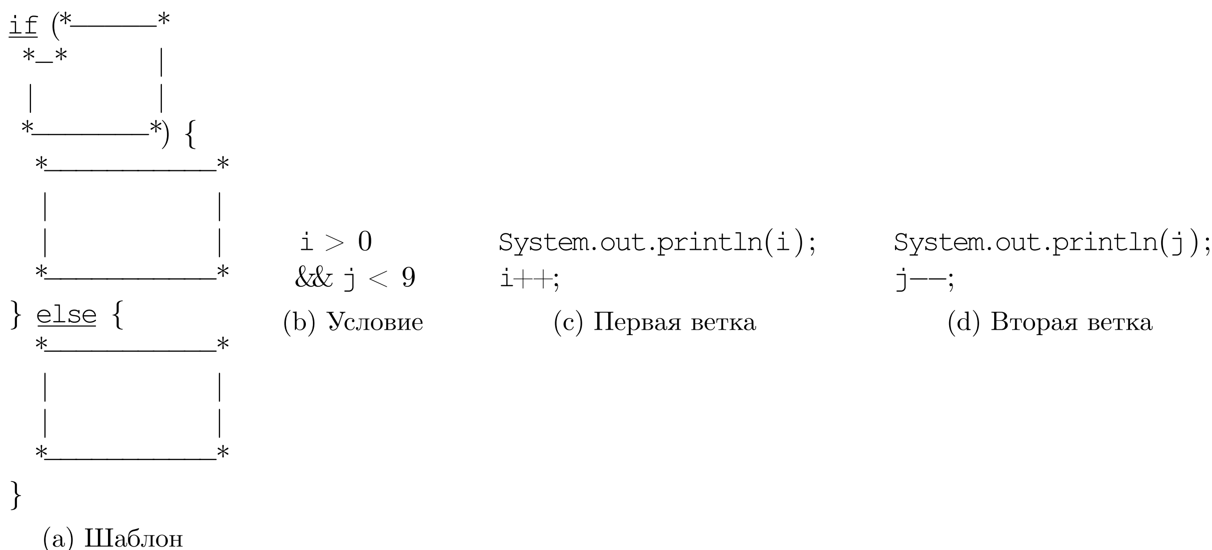
2.3.3. Дополнительная фильтрация вариантов

Для того, чтобы избежать экспоненциальной сложности вычисления оптимальной раскладки документа, мы ввели факторизацию по размерам (ширинам) текстового представления. Количество раскладок для каждого поддерева теперь ограничено, но, к сожалению, слишком большой константой — w^3 (w^2 для набора комбинаторов без `fill`), а сложность вычисления узлов `beside`, `above`, `fill` — $O(w^6)$ ($O(w^4)$ без комбинатора `fill`). Факторизация была основана на том факте, что среди раскладок с одинаковыми ширинами в любом дальнейшем использовании к лучшему результату приводит вариант с меньшей высотой. Для дополнительной фильтрации множества вариантов воспользуемся рассуждением, что если раскладка A обладает и меньшими ширинами, и меньшей высотой, чем раскладка B, то, аналогично предыдущей факторизации, достаточно оставить только ее во множестве вариантов.

Таким образом мы ввели отношение частичного порядка на раскладках, а описанная фильтрация — поиск минимумов на частично упорядоченном множестве представлений [9]. Очевидная реализация поиска имеет квадратичную сложность от размера множества, таким образом асимптотика алгоритма не ухудшается из-за данной фильтрации. К сожалению, эта оптимизация и не улучшает поведение в худшем случае, так как можно привести пример, когда все множество раскладок с допустимыми ширинами будет состоять из несравнимых элементов, но на практике количество вариантов существенно сокращается, что делает весь подход применимым на реальных данных.

2.3.4. Вставка в шаблон

В шаблонном подходе результат печати для узла синтаксического дерева строится за счет вставки представлений его поддеревьев в текст шаблона. Рассмотрим построение раскладки для конструкции `if`. У этой конструкции в общем случае три поддерева: условие, ветка, соответствующая выполнению условия, и ветка, соответствующая невыполнению условия. На рис. 16 представлен пример шаблона и представлений поддеревьев для конструкции `if`.



(a) Шаблон

(b) Условие

(c) Первая ветка

(d) Вторая ветка

Рис. 16: Пример входных данных для построения представления конструкции `if`

В шаблоне места для поддеревьев выделены блоками. Свойства этих блоков задают ограничения на использование представлений соответствующих поддеревьев. Так, в данном случае для каждого из них верно, что представление, применяемое для вставки, может быть многострочным. Блок, связанный с поддеревом условия, помимо этого также задает использование комбинатора `fill` с константой сдвига, соответствующей положению начала второй строчки блока относительно родительского узла. При использовании шаблона и представлений поддеревьев с рис. 16 получается

следующий результат:

```
if (i > 0
 && j < 9) {
  System.out.println(i);
  i++;
} else {
  System.out.println(j);
  j--;
}
```

При наивной реализации сложность вставки в шаблон есть $O(w^{3\times n})$, где w — максимальная ширина вывода, а n — число дочерних поддеревьев обрабатываемого узла, поскольку каждое из поддеревьев может иметь w^3 различных представлений после факторизации. Однако, за счет инкрементальной реализации вставки можно получить сложность $O(n \times w^6)$. Для этого надо разбить шаблон на список, в котором элементы соответствуют либо месту вставки поддерева, либо независимому тексту шаблона. Так, шаблон с рис. 16a представляется списком:

- "`if (`";
- место условия;
- ") {";
- место первой ветки;
- "} `else` {";
- место второй ветки;
- "}".

После над этим списком можно произвести свертку. При этом к аккумулятору на каждом шаге будет присоединяться представление нового элемента списка с помощью комбинаторов `beside`, `above` или `fill`. В таком случае аккумулятор всегда будет содержать не более w^3 представлений, как и каждый элемент обрабатываемого списка. Поэтому операция присоединения на каждом шаге будет иметь сложность $O(w^6)$, а вся вставка — $O(n \times w^6)$, так как количество участков текста не более чем $n + 1$, а значит списочное представление шаблона имеет размер не более $2 \times n + 1$.

2.4. Принтер-плагин языка Java для IntelliJ IDEA

Работа принтер-плагина разбивается на два этапа: подготовка шаблонов и непосредственно печать синтаксического дерева.

Для получения шаблонов алгоритм проходит по переданному пользователем репозиторию с эталонным исходным кодом. По каждому узлу деревьев разбора файлов из репозитория строится шаблон соответствующей синтаксической конструкции. Важной особенностью метода в целом является то, что для дальнейшей печати конструкций, имеющих плавающие число поддеревьев, необходимо иметь отдельный шаблон для каждой комбинации поддеревьев. Так, к примеру, для конструкции `if` языка Java необходимо иметь не менее двух шаблонов — с веткой `else` и без нее. В данном случае такое требование не кажется слишком обременительным, но если рассмотреть случай Java-класса, то ситуация несколько хуже — у него может не быть списка модификаторов доступа, суперкласса и реализуемых интерфейсов. Это серьезное ограничение, но его невозможно обойти в рамках данного подхода, поскольку единственной информацией о представлении конструкций должны быть полученные шаблоны.

Вычисление текстового представления переданного на вход синтаксического дерева происходит путем восходящего переписывания. Для каждого узла строится множество возможных раскладок (набор блоков текста) по тем же правилам факторизации, чтобы были применены для комбинаторной библиотеки. В дальнейшем представления поддеревьев используются для подстановки в шаблоны родительского узла, что на выходе дает набор раскладок для него. Шаблоны в данном контексте можно рассматривать как BURS-правила. Однако, несмотря на то, что число детей у родительского узла может быть велико (в случае некоторых конструкций языка Java более 5), это не приводит к экспоненциальному росту сложности по ширине вывода, которые следуют из рассуждений раздела 2.1, благодаря технике, описанной в разделе 2.3.4. Эта техника позволяет проводить подстановку раскладок за $O(n \times w^6)$ для любого $n \geq 2$ ($O(w^3)$ для $n = 1$), где w — ширина вывода, а n — число дочерних поддеревьев обрабатываемого узла. Таким образом, сложность вычисления сушественных раскладок для узла дерева составляет $O(t \times n \times w^6)$, где t — общее число шаблонов, а в целом поиск оптимального представления всего синтаксического дерева, переданного на форматирование, занимает $O(m \times t \times n \times w^6)$, где m — количество узлов в дереве.

Стоит заметить, что с инженерной точки зрения для решения отдельно стоящей задачи форматирования по образцу кода на Java в IntelliJ IDEA лучше подходит получение настроек для существующего форматтера с помощью известных техник обучения. Подобный подход используется в работе [8]. Но описанный в данной работе метод позволяет выразить представления, которые не предусмотрены встроенным форматтером, кроме того потенциально предоставляет возможность проще создавать принтеры для новых языков.

2.4.1. Практические особенности получения и использования шаблонов

Поскольку каждый узел дерева разбора из эталонного репозитория порождает шаблон, то частой ситуацией является присутствие шаблонов, которые функционально неотличимы. Их наличие ухудшает время работы принтера, так как повторяются аналогичные переходы от представлений поддеревьев обрабатываемого узла, приводящие к одному и тому же результату. Чтобы избежать данной проблемы достаточно отождествить шаблоны, обладающие одинаковым функциональном поведением. На практике это было реализовано в виде факторизации по текстовому представлению шаблона, в котором на месте меток записана информация об ограничении на раскладки соответствующих поддеревьев.

Противоположную проблему создают шаблоны, получаемые из уникальных, встречаемых только в специальном контексте представлений. Так, рассмотрим шаблоны для поля класса Java, получаемые из следующего текста:

```
private int a;
public double b;
protected String c;
```

Здесь на выходе получается один и тот же факторизованный шаблон. Несколько изменим пример. Пусть теперь в эталонном коде было сделано дополнительное выравнивание по типам и модификаторам доступа:

```
private   int    a;
public    double b;
protected String c;
```

По такому эталону получается три разных шаблона, каждый из которых не может быть использован в произвольном случае. Кроме того, так как подход в целом не умеет представлять поддеревья в зависимости от контекста, то полученные шаблоны бесполезны и вредны. Поэтому для борьбы с подобными ситуациями имеет смысл предпочитать то представление, что было получено из факторизованного шаблона с наибольшим *весом*, где под весом понимается встречаемость шаблона в эталонном тексте.

2.4.2. Обработка списочных структур

Особую сложность для представленного метода имеет печать синтаксических структур с переменным, неограниченным числом подвыражений. К примеру, это списки и бинарные выражения. Проблема вызвана неясностью в том, как для таких структур задавать шаблоны. Можно хранить представления для списков по количеству элементов до некоторой максимальной длины, но, во-первых, это достаточно обременительно, учитывая количество разнообразных типов списков, а, во-вторых, такой подход не сможет работать на списках большей длины. В рамках данной работы было реализовано два разных решения для этой задачи: классический, применяемый форматтерами из IDE, и с использованием шаблона для пары элементов списка.

Классический способ решения проблемы заключается в том, что списки просто печатаются заполняющим методом (см. рис. 17a), то есть перевод строки происходит в случае, когда следующий элемент уже не помешается на строчку, или каждый новый элемент печатается на новой строчке (см. рис. 17b). Кроме того, обычно позиция разделителя относительно элемента списка жестко "зашита" в форматтер — к примеру, запятая в списке параметров функции идет сразу же за параметром. Этот метод явным образом выбивается из модели шаблонов, так как вместо информации, предоставляемой эталонным кодом, использует наши ожидания о представлении списков, которые не всегда применимы, несмотря на то, что для большинства реальных ситуаций данное форматирование соответствует СК. Достоинством подхода является простота его реализации.

```
public int func(double a, String b,
                int c, char d,
                Integer e, double f) {
  return 0;
}
```

(a) Заполняющее форматирование списков

```
public int func(double a,
                String b,
                int c,
                char d,
                Integer e,
                double f) {
  return 0;
}
```

(b) Форматирование списка с переводом строки на каждый элемент

Рис. 17: Пример форматирования списков в IDE

Другое решение заключается в использовании специального шаблона, который задает переход между представлением головы списка и следующим обрабатываемым элементом. По сути метод является сверткой списка, где начальным значением является множество представлений первого элемента, а функция перехода с помощью шаблона соединяет посчитанные раскладки и результат форматирования следующего элемента. Данный подход позволяет получить нестандартные способы форматирова-

ния из эталонного кода (см. рис. 18).

```
public int func(double a
              , String b
              , int c
              , char d
              , Integer e
              , double f) {
  return 0;
}
```

Рис. 18: Пример нестандартного форматирования списков

На самом деле, одного шаблона перехода в большинстве ситуаций недостаточно, так как получаемое представление списка может стать слишком широким и не попасть в ограничение (в случае, если шаблон *"горизонтальный"*). Так, необходимо использовать согласованную группу шаблонов (см. рис. 19), которая позволит реализовывать некоторое обобщение заполняющего метода. Недостатком такого способа решения является сложность в идентификации подобных групп шаблонов, а также в сильно возрастающем числе самих шаблонов при наивном использовании всех переходов из эталонного кода.

```
A, B
```

(a) "Горизонтальный" шаблон

```
  A
, B
```

(b) "Вертикальный" шаблон

Рис. 19: Группа шаблонов для списков

Кроме того, некоторые стили форматирования списков могут быть сильно специфичными для мест использования. В частности, форматирование списка реализуемых интерфейсов Java-класса может концептуально зависеть от шаблона, используемого для форматирования всего класса в целом. Эта специфика приводит к усложнению общего алгоритма в смысле реализации и алгоритмически, так как для каждого узла дерева, переданного на печать, надо хранить не только единый набор раскладок, но и оптимальные множества представлений для каждого шаблона более высокого уровня.

2.4.3. Обработка комментариев

Отдельно стоит отменить задачу обработки комментариев. В целом, комментарии существенно выбиваются из общей картины. Первой их особенностью является то, что обычные неструктурированные комментарии, которые не предназначены для автоматической обработки подобно JavaDoc[14] и Doxygen[15], невозможно видоизменить, так как для этого необходимо понимать семантику комментария. Вторая особенность заключается в том, что для комментариев нет специально отведенных мест в абстрактном синтаксическом дереве программы (в соответствии с некоторыми определениями, комментариев в нем вообще быть не может), а значит для подхода, ориентированного на структурное синтаксическое сравнение, они являются специальным случаем.

В рамках данной работы было применен следующий подход: каждый блок комментариев связывается с узлом синтаксического дерева. Для связывания выбирается узел, который ближе всего к комментарию, при этом комментарий лежит полностью в его родительском узле. После построения вариантов раскладки для выбранного узла, к каждому из них присоединяется комментарий сверху или снизу, в зависимости от изначального положения. Получившееся множество представлений считается ответом для выбранного узла.

Существенным недостатком того, что комментарии остаются без изменений, является то, что нельзя управлять их шириной, что может привести к нежелательному увеличению минимальной возможной ширины раскладки всего дерева. Так, наличие строки комментария в 100 символов не дает возможности отформатировать текст на меньшую ширину. В частности, это объясняет, почему в нижеследующем анализе производительности для форматирования используются столь большие ширины (200 и 250). Возможным решением этой проблемы может быть смягчение требования на ширину текста — разрешение комментариям выходить за границы.

Дополнительным артефактом приведенного алгоритма обработки комментариев является то, что результат работы принтер-плагина не является его неподвижной точкой, поскольку при повторном запуске комментарии могут быть присоединены к другим синтаксическим узлам, нежели при предыдущем запуске.

Также комментарии влияют на получение шаблонов: наличие комментария внутри или рядом с синтаксической структурой может серьезно менять форматирование в эталонном коде, поэтому от большинства шаблонов, содержащих комментарии, приходится отказываться.

[14] http://java.sun.com/j2se/javadoc/
[15] http://doxygen.org/

2.4.4. Анализ производительности

Для оценки производительности были рассмотрены средние и большие исходные файлы проекта IntelliJ IDEA Community Edition[16]. Средние файлы проекта имеют размер от 100-500 строк, большие — несколько тысяч строк. Результаты производительности для средних и больших файлов приведены в таблице 2.

Имя файла	Кол-во строк	Время (шир. 200)	Время (шир. 250)
SearchRequestCollector.java	169	0.04	0.05
XDebugProcess.java	236	0.02	0.02
InitialConfigurationDialog.java	455	0.11	0.11
QuickEditHandler.java	504	0.21	0.21
PsiDirectoryImpl.java	618	0.11	0.12
Messages.java	2007	0.73	0.74
UIUtil.java	2808	1.15	1.14
AbstractTreeUi.java	5112	1.62	2.01
EditorImpl.java	6789	2.07	2.09
ConcurrentHashMap.java	7191	1.38	1.39

Таблица 2: Время форматирования файлов (в секундах)

Эти данные показывают, что принтер может быть использован для форматирования малых и средних файлов, но на больших файлах время работы слишком велико для практического применения. Проблему производительности можно будет считать решенной, если время работы снизится до 0.5 секунд.

2.4.5. Открытые проблемы

Проблемой, которую не получилось решить в рамках данной работы, является задание порядка поддеревьев для конструкций, которые его явным образом не определяют. К примеру, это верно для классов в Java: часто порядок полей, методов и подклассов может быть произвольным в контексте семантики программы, но в большинстве СК описаны рекомендации для их размещения относительно друг друга. В форматтерах IDE данная проблема решается просто — среди настроек есть указание порядка подконструкций для подобных случаев. В шаблонах это свойство явно не задается. Кроме попытки использования уже существующего подхода, данную проблему также можно решать с помощью обучения на эталонном репозитории или с дополнительным конфигурационным файлом. Но в целом такая задача несколько выбивается из простой печати синтаксического дерева, так как изменение порядка, к примеру, Java-полей может привести к изменению семантики программы, поэтому для проведения упорядочивания элементов конструкций может требоваться дополнительный статический анализ.

[16] http://github.com/jetbrains/intellij-community/

Заключение

В рамках данной работы достигнуты следующие результаты:

1. Для задачи поиска оптимальной раскладки документа, определенного с помощью комбинаторов из [4], на заданную ширину конструктивно доказана ее линейность относительно размеров дерева комбинаторов;

2. Разработан подход к заданию принтеров с помощью образцов для языков программирования;

3. Реализован плагин форматирования программных текстов на языке Java для IntelliJ IDEA в рамках апробации общего подхода.

Существует несколько направлений для развития данной работы. В рамках плагина стоит добавить анализ эталонного репозитория на полноту, чтобы для полученного по эталонному коду принтера можно было гарантировать возможность обработки любого абстрактного синтаксического дерева Java. Также для возможности практического применения необходимо повысить производительность плагина. Для общего подхода самой важной задачей является разработка абстрагированной от целевого языка программной системы. Кроме того, необходимо получить способ описания порядка поддеревьев для тех случаев, когда он не задан явно синтаксисом языка.

Список литературы

[1] А.Подкопаев. Форматирование текста программ на основе комбинаторов, сопоставления с образцом и синтаксических шаблонов // Курсовая работа, кафедра Системного Программирования, МатМех СПбГУ, 2013.

[2] A.V.Aho, M.Ganapathi, S.W.K.Tjiang. Code Generation Using Tree Matching and Dynamic Programming // ACM Transactions on Programming Languages and Systems, 11(4), 1989.

[3] G.Alkhatib. The maintenance problem of application software: an empirical analysis // Journal of Software Maintenance, 4(2):83–104, 1992.

[4] P.Azero, S.D.Swierstra. Optimal Pretty-Printing Combinators. `http://www.cs.ruu.nl/groups/ST/Software/PP`, 1998.

[5] P.Azero, S.D.Swierstra, J.Saraiava. Designing and Implementing Combinator Languages // Advanced Functional Programming, 1999.

[6] O.Chitil. Pretty Printing with Lazy Dequeues // ACM Trans. Program. Lang. Syst. 27(1), 2005.

[7] H.Comon, M.Dauchet, R.Gilleron *et al.* Tree Automata Techniques and Applications, `http://www.grappa.univ-lille3.fr/tata`, 2007.

[8] F.Corbo, C.Del Grosso, M.Di Penta. Smart Formatter: Learning Coding Style from Existing Source Code // Software Maintenance. ICSM 2007. IEEE International Conference, pp.525,526, 2-5, 2007.

[9] C.Daskalakis, R.M.Karp, E.Mossel, S.Riesenfeld, E.Verbin. Sorting and Selection in Posets. In Proceedings of the twentieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '09). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 392-401, 2009.

[10] J.Hughes. The Design of a Pretty-printing Library // Advanced Functional Programming, 1995.

[11] S.Jackson, P.Devanbu, K.Ma. Stable, Flexible, Peephole Pretty-Printing // Journal Science of Computer Programming, 72(1-2), 2008.

[12] M.De Jonge. A Pretty-Printer for Every Occasion // Proceedings of the 2nd International Symposium on Constructing Software Engineering Tools, 2000.

[13] M.De Jonge. Pretty-printing for Software Reengineering // Proceedings of the International Conference On Software Maintenance, 2002.

[14] O.Kiselyov, S.Peyton-Jones, A.Sabry. Lazy vs. Yield: Incremental, Linear Pretty-Printing // 10th Asian Symposium on Programming Languages and Systems, 2012.

[15] D.C.Oppen. Pretty-printing // ACM Transactions on Programming Languages and Systems, 2(4), 1980.

[16] Peyton Jones S. Haskell Pretty-printer Library // `http://www.haskell.org/ghc/docs/latest/html/libraries/pretty-1.1.1.0/Text-PrettyPrint.html`, 1997.

[17] A.Podkopaev, D.Boulytchev. Polynomial-Time Optimal Pretty-Printing Combinators with Choice // Принято на PSI-2014.

[18] T.Rendel, K.Ostermann. Invertible syntax descriptions: unifying parsing and pretty printing // SIGPLAN Not. 45, 11, 1-12, 2010.

[19] J.Shore. The Art of Agile Development. O'Relly Media, 2010.

[20] S.D.Swierstra. Linear, Online, Functional Pretty Printing (corrected and extended version). Technical report, UU-CS-2004-025a. Institute of Information and Computing Sciences, Utrecht University, 2004.

[21] S.D.Swierstra, O. Chitil. Linear, Bounded, Functional Pretty-printing // J. Funct. Program., Vol. 19, 2009.

[22] M.Van den Brand, E.Visser. Generation of Formatters for Context-free Languages // ACM Trans. Softw. Eng. Methodol., 5(1), 1996.

[23] P.Wadler. A Prettier Printer // The Fun of Programming. Palgrave MacMillan, 2003.

A. Программный код полиномиальной сложности оптимальных принтер-комбинаторов на языке Haskell

```haskell
import qualified Data.HashMap.Strict as Map
import qualified Data.List as List
import Format -- Format from Azero, Swierstra library

type Variants = Map.HashMap Frame Format
type Doc = Int -> Variants

update :: Format -> Variants -> Variants
update fmt = Map.insertWith min (fmtToFrame fmt) fmt

checkUpdate :: Int -> Format -> Variants -> Variants
checkUpdate n f = if isSuitable n f then update f else id

text :: String -> Doc
text s n = checkUpdate n (s2fmt s) Map.empty

indent :: Int -> Doc -> Doc
indent i d n = Map.fromList vs where
 vars = Map.elems $ Map.filter (isSuitable (n-i)) (d n)
 vs   = map ((\f -> (fmtToFrame f, f)) . indentFmt i) vars

choice, beside, above :: Doc -> Doc -> Doc
choice a b n = Map.foldl' (flip update) (b n) (a n)
beside a b   = cross besideFmt a b
above  a b   = cross aboveFmt  a b
((>//<), (>|<), (>-<)) = (choice, beside, above)

cross :: (Format -> Format -> Format) -> Doc -> Doc -> Doc
cross f a b n = Map.foldl' bFold Map.empty (a n) where
 bFold m fa = Map.foldl' (flip $ checkUpdate n . f fa) m bv
 bv         = b n

pretty :: Int -> Doc -> String
pretty n d = case Map.elems (d n) of
               [] -> error "No layout"
               xs -> (\x -> txtstr x 0 "") $ List.minimum xs
```