Программное обеспечение вычислительных, телекоммуникационных и управляющих систем

DOI: 10.5862/JCSTCS.224.4
УДК 004.4'232

А.В. Подкопаев, А.Ю. Коровянский, И.С. Озерных

ЯЗЫКОНЕЗАВИСИМОЕ ФОРМАТИРОВАНИЕ ТЕКСТОВ ПРОГРАММ НА ОСНОВЕ СОПОСТАВЛЕНИЯ С ОБРАЗЦОМ И СИНТАКСИЧЕСКИХ ШАБЛОНОВ

A.V. Podkopaev, A.Yu. Koroviansk ii, I.S. Ozernykh

A LANGUAGE-INDEPENDENT CODE FORMATTING BY SYNTACTIC MATCHING AND TEMPLATES

Рассмотрена проблема форматирования программных текстов. Предложен новый подход, позволяющий форматировать целевой код по образцу. Разработанное решение вычисляет оптимальное представление текста за полиномиальное время. В рамках апробации разработаны форматеры для языков Java и Haskell.

ФОРМАТИРОВАНИЕ; ШАБЛОН; КОМБИНАТОРЫ.

In paper we consider a code-formatting problem. A novel concept of declarative printers is introduced. These devices can perform formatting in accordance with the style of the rest of the source code. For this purpose, declarative formatters extract syntactic templates from the sample code and use them to construct a new representation of the input program. A proposed solution produces optimal program presentation in polynomial time. It is achieved by using polynomial-time pretty-printer combinators and introducing a partial order on text representations. We also present the results of evaluating the approach in the Java and Haskell languages. The development of a declarative printer for simple imperative language While is described. A comparison with modern IDEs code formatters is also presented.

FORMATTING; TEMPLATE; COMBINATORS.

Широко известным фактом является то, что программный код больших информационных систем должен быть легко читаем. Для этого код подвергается *форматированию*, т. е. в него добавляются символы, которые не несут семантического значения с точки зрения языка программирования, однако, позволяют выделить структурные элементы программы. Какие структурные элементы требуют выделения и способ их выделения определяется *стандартом кодирования* (СК, coding convention), который выбирается командой разработчиков для проекта.

Стоит отметить, что не существует универсального СК, т. к. каждый конкретный стандарт сильно зависит от языка программирования, к проектам, на котором стандарт применяется. Так, абсолютно бесполезно форматировать программный код, написанный на C++, по правилам форматирования S-выражений языка Lisp. Однако и для одного и того же языка программирования может существовать множество стандартов. Например, для C++ существует

как минимум три широко распространенных стандарта: BSD[1], GNU[2], Google[3].

Современные среды интегрированной разработки (IDE), такие как IntelliJ IDEA, Eclipse и Visual Studio, как и средства реинжиниринга [2], помогают придерживаться выбранного СК. Они обладают встроенными *форматерами*, которые производят форматирование выбранной части кода или целого файла. Какого именно СК должен придерживаться форматер, задается множеством настроек. Например, для языка C++ есть подобные: помещать фигурную скобку начала тела функции на той же строчке, что и имя функции, или на следующей, использовать отступ равный двум или четырем пробелам для операторов внутри фигурных скобок и т. д.

У данного подхода есть несколько недостатков. Во-первых, настройки форматеров приведенных IDE не позволяют выразить СК, сильно отличающийся от общепринятых. Например, в них невыразим стиль, подразумевающий помещение закрывающей скобки на той же строчке, что и последний оператор блока (см. листинг 1). Во-вторых, часто возникает необходимость следовать СК некоторого существующего кода. Для этого в рамках приведенного подхода нужно изучить кодовую базу и вручную задать настройки форматирования в соответствии с ней, что является трудной задачей из-за количества различных настроек.

В этой работе представлен метод, позволяющий создавать форматеры для различных языков программирования, которые настраиваются на требуемый СК по образцу кода. Подобные форматеры мы будем называть *декларативными*. Декларативность достигается путем вычленения из образца *синтаксических шаблонов* для конструкций целевого языка и построения с их помощью представлений для переданных на форматирование программ. Важной особенностью декларативных форматеров является то, что они вычисляют *оптимальное*, в терминах статьи [1], представление для обрабатываемой программы за линейное время относительно размера ее абстрактного синтаксического дерева. Линейная сложность получена за счет использования модернизированной библиотеки принтер-комбинаторов из [7]. В рамках апробации метода была разработана инфраструктура для создания форматер-плагинов для среды интегрированной разработки IntelliJ IDEA, и с ее помощью были разработаны декларативные форматеры для языков Java, Haskell и учебного языка While. Разработка форматер-плагина для последнего детально описана далее в статье.

Оптимальное представление программы. Одной из важных характеристик программного текста, с точки зрения его оформления, является ширина текста, т. е. максимум количества символов в его строчках. Стоит отметить, что слишком широкие тексты не будут рассматриваться нами как хорошие. Так, восприятие текста, который не помещается по ширине на экран, серьезно затруднено. Однако допустимая ширина текста может варьироваться в зависимости от задач, поэтому библиотеки принтер-комбинаторов [1, 5, 7, 8] и форматеры IDE позволяют явным образом задавать верхнюю границу для ширины продуцируемого текста. Это верно и для предложенных декларативных форматеров.

```
void sort(int* arr, int length)
{ for (int i = 0; i < length-1; i++)
   { for (int j = i+1; j < length; j++)
      { if (arr[i] > arr[j])
         { swap(arr, i, j); } } } }
```

Листинг 1. Программный код с нестандартным форматированием

Под *оптимальным* представлением программы, вслед за [1], понимается то, что имеет минимально возможное количество строк при соблюдении СК и ограничения на ширину. Интуиция за этим определением следующая: полученное представление должно быть максимально обозримым.

Библиотека принтер-комбинаторов. На нижнем уровне декларативные форматеры

[1] https://www.freebsd.org/cgi/man.cgi?query=style&sektion=9

[2] http://www.gnu.org/prep/standards/standards.html

[3] https://google-styleguide.googlecode.com/svn/trunk/cppguide.html

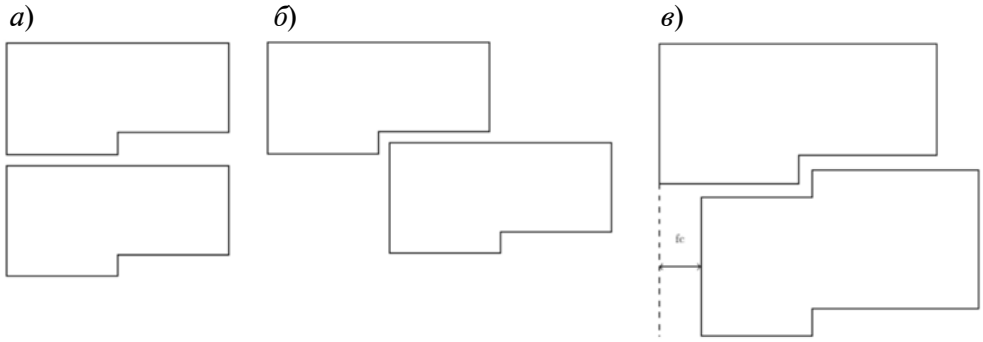

Рис. 1. Блоки и виды их склеек:
а – вертикальная; *б* – горизонтальная; *в* – горизонтальная со сдвигом

используют модернизированную библиотеку принтер-комбинаторов [7]. Центральным понятием оригинальной библиотеки является *блок* (рис. 1) – модель части текста с указанными геометрическими характеристиками: высота (количество строк), максимальная ширина до последней строчки и ширина последней строчки. Блок может быть получен из текста или путем соединения двух других с помощью *вертикальной* или *горизонтальной склеек* (см. рис. 1 *а* и *б*). Стоит заметить, что характеристики результата склейки зависят исключительно от характеристик соединяемых блоков кода. Для целей декларативных форматеров в интерфейс библиотеки пришлось добавить дополнительный вид склейки – *горизонтальную склейку со сдвигом* (см. рис. 1 *в*). Из-за нововведения для блоков дополнительно нужно хранить ширину первой строчки. Приведенные операции обобщаются на множества блоков естественным образом – склейка двух множеств есть множество попарных склеек. Дополнительно вводится операция *выбора* (choice), объединяющая два множества в одно.

Склейки и операции выбора позволяют композиционно задавать простые форматеры для различных типов данных, в том числе и синтаксических деревьев. Результатом работы таких форматеров становится множество блоков, из которых далее по характеристикам выбирается оптимальное представление. В [7] замечено, что в промежуточных множествах не все элементы нужны, поскольку нас интересует исключительно оптимальный результат. Для фильтрации ненужных блоков используется специальное представление для множеств – таблица, индексами которой служат ширины блоков. Так, в ячейке таблицы (100, 50, 80) хранится блок кода с максимальной шириной, равной 100, шириной последней строчки – 50, а первой строчки – 80. При этом данный блок обладает минимальной высотой среди всех потенциальных элементов с теми же ширинами. Стоит отметить, что максимальные значения для индексов равны максимально допустимой ширине. Кроме того, очевидно, что не все ячейки таблиц обязаны быть заполнены. Верхняя оценка на число элементов в таким образом построенном множестве блоков – w^3 (здесь и далее w – максимально допустимая ширина). Заметим, что алгоритмическая сложность склеек в этом случае равна $O(w^6)$, а операции выбора – $O(w^3)$.

На самом деле множество может быть организовано еще эффективнее. Для этого над блоками нужно ввести отношение частичного порядка, такое что для любых блоков A и B верно, что A меньше или равно B тогда и только тогда, когда все характеристики (ширины и высота) A меньше или равны соответствующим характеристикам B. Тогда во множестве достаточно хранить минимумы изначального множества по этому отношению. Наивная реализация поиска минимумов [4] в данном случае имеет сложностную оценку $O(w^6)$, как квадрат от размера множества, а значит, если ее выполнять после операций склейки, то асимптотика не ухудшается. С другой стороны, данная оптимизация и не улучшает оценку

```
for (@1; @2; @3)        // Поддеревья 1, 2, 3 должны иметь
{ @4 }                  // однострочные представления.
                        // Поддерево 4 может иметь
                        // многострочное представление.
```

Листинг 2. Пример синтаксического шаблона конструкции **for**

на число блоков во множествах, а следовательно, и сложностную оценку решения, однако, на практике дает существенный прирост производительности.

Таким образом, модифицированная библиотека позволяет находить оптимальное представление за $O(n \times w^6)$, где n – число склеек. В контексте декларативных форматеров библиотека используется для организации вставки представлений поддеревьев в шаблон. Реализация библиотеки находится по адресу https://github.com/anlun/format

Один из недостатков описанной библиотеки и, как это будет видно далее, декларативных форматеров заключается в том, что для некоторых входов может не быть результата, как следствие требований критерия оптимальности. Так происходит, если максимальная ширина вывода слишком мала, чтобы вместить какое-нибудь представление, удовлетворяющее СК. Однако этот недостаток может быть легко решен, если дополнительно хранить во множестве блоков один или несколько, выходящих за границы ширины элементов.

Декларативные форматеры. Глобально работа декларативных форматеров разбивается на два этапа: получение из образца информации об СК, которого необходимо придерживаться, и непосредственно форматирование целевого кода. На первом этапе по образцу кода, который представляет собой один или несколько файлов на целевом языке, строится множество *синтаксических шаблонов*. Синтаксические шаблоны являются данными, 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. На практике шаблон является размеченным деревом 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. Пример синтаксиче-

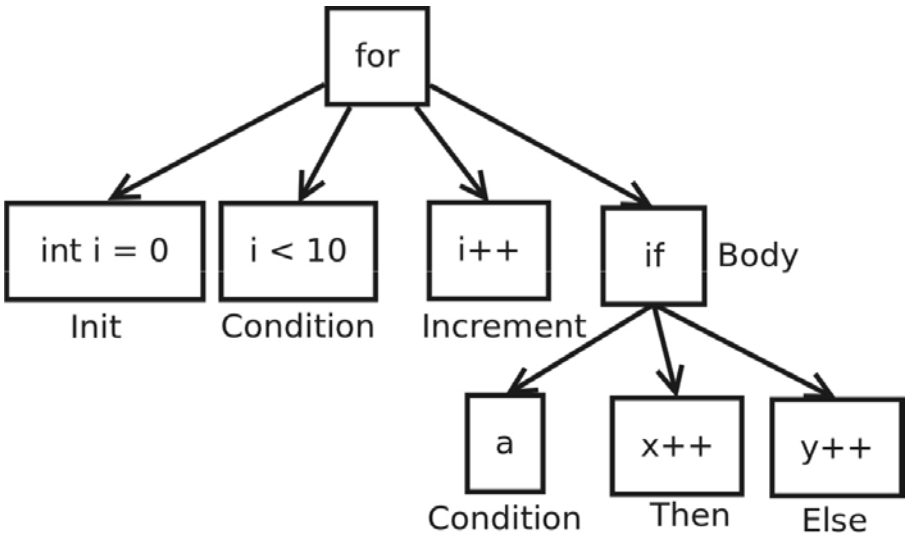

Рис. 2. Пример синтаксического дерева конструкции **for**

ского шаблона конструкции **for**, полученного из кода на листинге 1, представлен в листинге 2. Важно заметить, что для одного типа конструкции может быть извлечено несколько синтаксических шаблонов. В таком случае все извлеченные шаблоны будут использоваться на втором этапе работы декларативных форматеров.

На втором этапе происходит форматирование переданного на обработку текста. Для текста строится его синтаксическое дерево, по которому восходящий алгоритм конструирует новые текстовые представления для каждого значимого узла, используя полученные из образца синтаксические шаблоны. Так, для узла дерева в подходящие ему по типу синтаксические шаблоны вставляются представления непосредственных поддеревьев узла. Рассмотрим дерево на рис. 2. Если для него использовать шаблон из листинга 2, то по представлениям его поддеревьев получаем результат, представленный в листинге 3. В итоге из полученного множества представлений нужно выбрать одно окончательное. Выбор происходит согласно критерию оптимальности.

```
Init: {“int i = 0”}
Condition: {“i < 10”}
Increment: {“i++”}
Body: {“if (a) x++ else y++”;
“if (a) x++
  else y++”
}

For: {
“for (int i = 0; i < 10; i++)
  { if (a) x++ else y++ }”;
“for (int i = 0; i < 10; i++)
  { if (a) x++
     else y++ }”
}
```

Листинг 3. Пример получаемых представлений для конструкции **for**

Вычислительная сложность выполнения первого подготовительного этапа линейно зависит от размеров переданного образца. Поскольку первый этап выполняется один раз для настройки форматера, после чего тот готов обрабатывать произвольное число программ, то производительность первого этапа не имеет серьезного значения. Так, переданный для анализа образец может быть очень большим. Сложность второго этапа равна $O(T \times n \times w^6)$, где T – максимальное количество шаблонов одного типа, n – размер синтаксического дерева форматируемой программы. Такая оценка имеет место, поскольку каждый узел синтаксического дерева обрабатывается не более одного раза. При этом представления для узла строятся с применением шаблонов подходящего типа, а вставка в шаблон занимает $O(w^6)$, как будет показано далее.

Стоит отметить, что все представления целевой программы, построенные с помощью извлеченных из образца синтаксических шаблонов, мы считаем соответствующими требуемому СК. При этом без дополнительной доработки декларативные форматеры игнорируют контекстно-зависимые характеристики форматирования узлов. Это ограничение связано с тем, что на данный момент неизвестен общий подход для извлечения контекстно-зависимой информации из образца. Кроме того, при контекстно-зависимом форматировании сложность второго этапа (этапа форматирования) перестает быть линейной, т. к. при обработке узла может потребоваться пересмотр результатов форматирования его дочерних узлов, для которых он является контекстом. В худшем случае это приводит к экспоненциальной сложности обработки синтаксического дерева.

Извлечение синтаксического шаблона. В начале обработки образца кода происходит его синтаксический анализ. Для каждого узла полученного синтаксического дерева, соответствующего интересной с точки зрения форматера конструкции, происходит извлечение шаблона. В первую очередь из образца нужно вырезать текст обрабатываемого узла. Стоит отметить, что в вырезанном тексте должен отсутствовать сдвиг относительно объемлющих конструкций, поскольку он контекстно-зависим и в дальнейшем будет мешать при операциях склейки. Далее для конструкций с переменным числом непосредственных поддеревьев в шаблоне

появляется запись о том, какие подузлы в нем присутствуют. Например, конструкция **if** может иметь или не иметь ветку **else**. Для каждого присутствующего подузла устанавливается, является ли текущее представление однострочным или многострочным, а также каков сдвиг относительно конструкции, по которой строится шаблон, в многострочном случае.

Важно заметить, что извлечение шаблона при условии, что вырезанный текст содержит комментарии, в общем случае невозможно по двум причинам. Во-первых, оставить комментарии нельзя, т. к. они несут некоторую контекстно-зависимую семантику. Во-вторых, вырезать комментарии простым способом тоже нельзя, т. к. это может привести к шаблону, не соответствующему СК. Поэтому в нашей реализации для извлечения шаблонов используются только части кода, не содержащие комментарии в значимых местах, т. е. вне позиций подузлов.

Согласно оценке алгоритмической сложности из предыдущего раздела, форматирование программ линейно зависит от числа полученных шаблонов. Как следствие, при увеличении размера образца в k раз, во столько же раз увеличивается время на форматирование целевой программы. Очевидно, что такое поведение крайне нежелательно. Чтобы обойти это ограничение, мы используем *факторизацию* множества шаблонов. Так, шаблоны, имеющие одинаковые ограничения на поддеревья и неотличимые при вставке, мы отнесем к одному классу эквивалентности и будем хранить по одному представителю каждого класса. Данный метод существенным образом уменьшает количество шаблонов для образцов кода, которые действительно следуют некоторому СК, т. к. конструкции в них представлены единообразно.

Вставка в синтаксический шаблон. Из примера вставки в синтаксический шаблон для конструкции **for**, приведенный ранее, должна быть понятна общая идея данного процесса. Вставка в шаблон происходит с помощью горизонтальных склеек. Для этого текст шаблона разделяется на отрезки, соответствующие участкам поддеревьев и тексту самого шаблона. Пример разделения шаблона конструкции **for** представлен в листинге 4. Далее на каждом шаге происходит склейка текущего результата и множества представлений, соответствующего обрабатываемому отрезку. Для отрезков поддеревьев используются предподсчитанные множества представлений, а для текстовых отрезков – множества из единственного блока, построенного по тексту. Вид склейки (со сдвигом или без) для конкретного подузла записан в шаблоне.

Сложность вставки в шаблон равна $O(K \times w^6)$, где K – максимальное количество поддеревьев конструкции. Заметим, что K является константой для целевого языка, которая для всех известных нам языков не превосходит 20, поэтому далее мы будем опускать ее из оценок.

```
1) “for (“
2) @1
3) “; “
4) @2
5) “; “
6) @3
7) “)\n { ”
8) @4
9) ” }”
```

Листинг 4. Разделения шаблона **for** *на отрезки для последующей склейки*

```
printf(“%d %d %d, %d”
        , a
        , b
        , c
        , d)
```

Листинг 5. Нестандартное форматирование для списка аргументов функции

Обработка списочных структур. Определенную проблему представляют конструкции, потенциально обладающие неограниченным числом поддеревьев. К таким конструкциям относятся списки. Списки выходят за границы общего подхода, поскольку мы не можем завести по шаблону для каждой возможной длины списка, т. к. это было бы слишком обременительно. Кроме того, всегда может появиться спи-

сок, для которого у нас не будет шаблона. Имеются два возможных варианта для решения данной проблемы. Первый, классический, используется в существующих IDE, а второй подразумевает наличие так называемых *переходных* шаблонов.

Классический способ заключается в том, что списки печатаются заполняющим методом, т. е. переход на новую строку происходит в тот момент, когда достигнуто ограничение по ширине текста, или каждый элемент списка печатается на новой строчке. При этом вид и представление разделителя элементов (“, ”, “; ”, “| ” и т. д.) постоянно для каждого конкретного типа списков. Такой способ решения проблемы выходит за пределы декларативного подхода, однако, может быть достаточно просто и эффективно реализован.

Альтернативный способ подразумевает получение и использование *переходных* шаблонов, т. е. конструкции, позволяющей соединить представление головы списка и очередного элемента. Они позволяют конструировать нестандартные представления для списков. Примером может послужить помещение запятой на новой строчке перед элементом списка (см. листинг 5). Дополнительным требованием для практического использования способа становится необходимость иметь несколько переходных шаблонов, позволяющих получать представление для списка, которое подпадает под ограничение ширины. Например, наравне с «горизонтальным» шаблоном @1, @2 нужно иметь «вертикальный» шаблон @1,
@2.

При этом шаблоны должны быть согласованными с точки зрения СК. Препятствием для использования данного способа является сложное вычленение шаблонов из образца и экспоненциальная сложность получения представления по списку.

В нашей инфраструктуре реализованы оба способа, но в большинстве случаев применяется классический.

Обработка комментариев. Сложность создают и некоторые комментарии. Комментарии бывают двух видов: структурированные (JavaDoc, DoxyGen) и неструктурированные (обычные комментарии). Структурированные комментарии не выбиваются из общего подхода, т. к. для них точно также могут быть извлечены и применены шаблоны. Неструктурированные комментарии не могут быть подвержены изменению, поскольку взаимоположение слов и строк в них может быть семантиче-

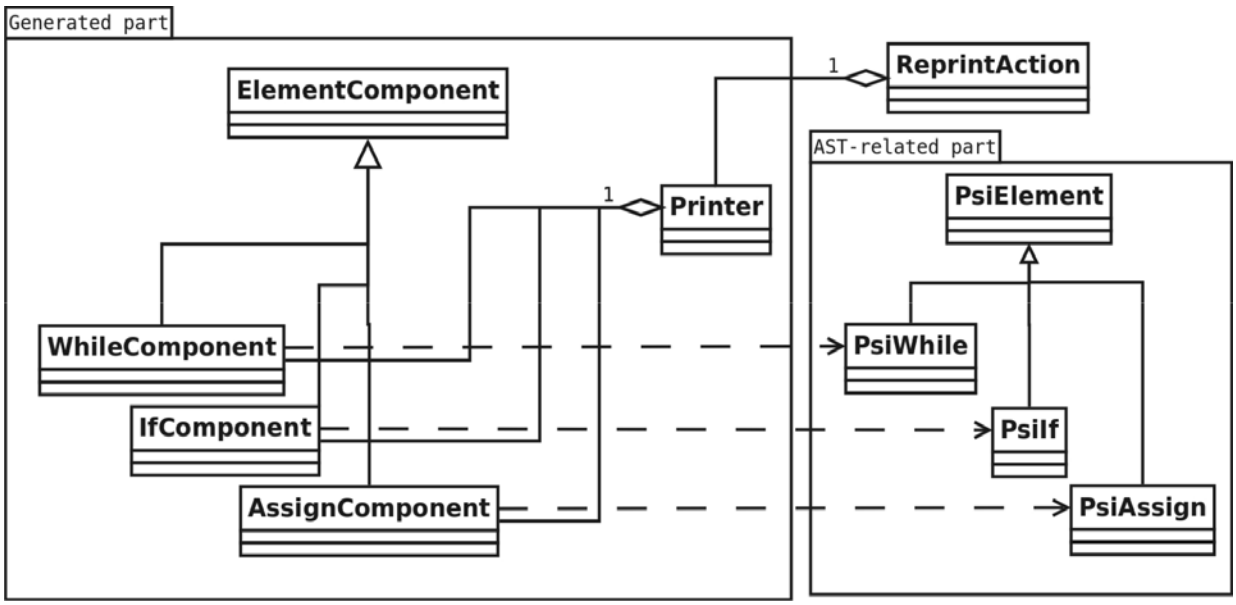

Рис. 3. Архитектура декларативного форматера языка While

ски важно. Так, если комментарий содержит ASCII-диаграмму, то даже незначительное изменение может испортить изображение. Поскольку у нас нет механизмов для изменения комментариев, то слишком длинные комментарии усложняют выполнение ограничения ширины. Одним из вариантов решения может быть игнорирование комментариев при подсчете ширин представлений. Кроме того, неструктурированные комментарии не имеют фиксированного места в синтаксическом дереве (а в соответствии с некоторыми определениями, комментариев в нем вообще не должно быть), что не позволяет обрабатывать их так же, как и остальные конструкции.

В нашей реализации использован следующий подход. Каждый комментарий связывается с самым близким ему узлом синтаксического дерева, чей непосредственный родитель полностью содержит комментарий внутри себя. После вычисления представлений для связанного узла к ним сверху или снизу, в зависимости от изначального расположения, присоединяется текст комментария. Получившееся считается итоговым множеством представлений для связанного узла.

Модельный язык While. Язык While [6] является простым императивным языком программирования. Он обладает небольшим набором конструкций: оператор ветвления **if**, оператор цикла **while**, присваивание переменной значения, чтение переменной из потока **read** и вывод значения выражения в поток **write**. Пример программы на этом языке приведен в листинге 6. В листинге 7 представлена грамматика языка While в форме Бэкуса–Наура.

Несмотря на небольшое число конструкций, язык представляет интерес с точки зрения декларативных форматеров. Так, для данного языка можно варьировать множество стилистических характеристик: ставить ли пробел до/после скобок в конструкциях **write** и **read**, помещать ключевое слово **do** в конструкции **while** на той же строчке, что и условие цикла, или нет и т. д. На примере учебного языка While мы рассмотрим архитектуру и разработку форматер-плагина для IntelliJ IDEA.

```
read(x);
read(n);
res := 1;
while (n > 0) do
  if (n % 2 = 0)
  then n := n / 2;
         x := x * x;
  else  n := n - 1;
         res := x * res;
  fi
od
write(res);
```

Листинг 6. Быстрое возведение в степень на языке While

```
prog ::= stmtList
stmtList ::= stmt *
stmt ::= skip | write | read | while | if | assign
skip ::= “skip” “;”
write ::= “write” “(” exp “)” “;”
read ::= “read” “(” id “)” “;”
while ::= “while” “(” bexp “)” “do” stmtList “od”
if ::= “if” “(” bexp “)” “then” stmtList
         [“else” stmtList] “fi”
assign ::= id “:=” exp “;”

exp ::= literal | id | exp op exp
op ::= “+” | ”-” | ”*” | ”/” | “%”
bexp ::= “true” | “false” | bexp bop bexp |
           exp rel exp | “not” bexp
bop ::= “or” | “and”
rel ::= “<=” | “<” | “=” | “>” | “>=”
```

Листинг 7. Грамматика языка While в форме Бэкуса–Наура

Декларативный форматер для языка While. Частичная архитектура декларативного форматера для языка While приведена на рис. 3. Ключевым классом форматера является Printer. Он извлекает шаблоны из образца кода, а потом применяет их для целевой программы. Printer содержит в себе ссылки на экземпляры классов-наследников ElementComponent, определяющие специфичные для конструкций языка вспомогательные методы для извлечения и вставки в шаблон. Так, на диаграмме изо-

```
<component name=”Assign” psiComponentClass=”PsiAssign”>
   <subtree name=”variable” psiGetMethod=”Id” isRequired=”true” />
   <subtree name=”expression” psiGetMethod=”Expr” isRequired=”true” />
</component>

<component name=”If” psiComponentClass=”PsiIf”>
   <subtree name=”thenBranch” psiGetMethod=”ThenBranch” isRequired=”true” />
   <subtree name=”elseBranch” psiGetMethod=”ElseBranch” isRequired=”false” />
   <subtree name=”condition” psiGetMethod=”Bexpr” isRequired=”true” />
</component>

<component name=”StmtList” psiComponentClass=”PsiStmtList”>
   <subtree name=”list” psiGetMethod=”StmtList” isRequired=”true”
         hasSeveralElements=”true” composition=”vertical”>
   </subtree>
</component>
```

Листинг 8. XML-описания конструкций **assign**, **if** *и списка конструкций языка While*

бражены WhileComponent, IfComponent, AssignComponent, соответствующие конструкциям while, if и присваивания языка While соответственно. Также на диаграмме представлен класс ReprintAction, являющийся посредником между плагином и средой разработки IntelliJ IDEA и, в зависимости от действий пользователя, отдающий команды Printer извлечь шаблоны из переданного образца или форматировать целевой текст. Правая компонента диаграммы иллюстрирует классы синтаксического дерева языка While (“Psi” – стандартный префикс классов синтаксических узлов в платформе IntelliJ IDEA).

Классы компоненты, изображенной на рис. 3 справа, являются естественной частью синтаксического анализатора целевого языка. При разработке декларативного форматера языка, который уже поддерживается платформой IntelliJ IDEA, данную компоненту не нужно реализовывать – она есть и может быть снова использована. Иначе, как и в случае с языком While, классы компоненты, вместе с синтаксическим анализатором, могут быть получены путем генерации из описания грамматики в форме Бэкуса–Наура с помощью GrammarKit[4]. Реальная грамматика языка While, использованная для генерации синтаксического анализатора, сложнее той, что представлена в листинге 7, поскольку в ней исключена левая рекурсия в продукциях.

Компонента, изображенная на рис. 3 слева, может быть получена по XML-описанию конструкций целевого языка с помощью разработанного нами генератора. XML-описание состоит из указания имени конструкции, связанного с ней класса синтаксического дерева и описаний подконструкций. Для подконструкции указывается: ее имя, название метода связанного класса синтаксического дерева, вызов которого возвращает соответствующее поддерево, флаг обязательности подконструкции. Кроме того, если подконструкция является списком, то указывается и способ соединения ее подэлементов. Примеры XML-описаний для конструкций **assign**, **if** и списка конструкций приведены в листинге 8.

Таким образом, для создания декларативного форматера языка с использованием нашей инфраструктуры достаточно иметь синтаксический анализатор и разработать XML-описания для его конструкций. Форматер-плагин для языка While находится по адресу https://github.com/anlun/

[4] https://github.com/JetBrains/Grammar-Kit

а)

```
read(n);
if(n>=0)then
  a:=0;
  b:=1;
  res:=1;
  while(n>0)do
    res:=a+b;
    a:=b;
    b:=res;
    n:=n-1;
  od
  write(res);
else
  res:=0-1;
  write(res);
fi
```

б)

```
read ( n );
if ( n >= 0 )
then a := 0;
       b := 1;
       res := 1;
       while ( n > 0 )
       do res := a+b;
            a := b;
            b := res;
            n := n-1; od
       write ( res );
else res := 0 - 1;
      write ( res ); fi
```

Листинг 9. Образцы форматирования

а)

```
read(x);
read(n);
res:=1;
while(n>0)do
  if(n%2=0)then
    n:=n/2;
    x:=x*x;
  else
    n:=n-1;
    res:=x*res;
  fi
od
write(res);
```

б)

```
read ( x );
read ( n );
res := 1;
while ( n > 0 )
do if ( n % 2 = 0 )
    then n := n / 2;
           x := x * x;
    else n := n - 1;
           res := x * res; fi od
write ( res );
```

Листинг 10. Результат форматирования программы из листинга 6 по образцам из листингов 9 а и б

whileLang-idea-plugin

Пример работы декларативного форматера для языка While. В листингах 9 *а* и *б* приведены программы на языке While, использованные в качестве образцов для форматирования одного и того же кода (см. листинг 6). Результаты форматирования представлены в листингах 10 *а* и *б*.

Апробация подхода для языков Java и Haskell. Для более серьезной апробации метода были выбраны языки Java и Haskell. Язык Java интересен для апробации, поскольку он входит в семейство C-подобных языков, а значит, получив форматер для него, достаточно просто прийти к языкам C, C++ и C#. Кроме того, язык Java имеет серьезную поддержку со стороны IntelliJ IDEA, в частности, для него реализованы синтаксический анализатор и стандартный форматер.

Таблица 1

Результаты тестирования декларативного форматера для языка Java

Имя файла	Количество строк	Время при ширине 200, с	Время при ширине 250, с
SearchRequestCollector.java	169	0,04	0,05
XDebugProcess.java	236	0,02	0,02
QuickEditHandler.java	504	0,21	0,21
PsiDirectoryImpl.java	618	0,11	0,12
Messages.java	2007	0,73	0,74
AbstractTreeUi.java	5112	1,62	2,01
EditorImpl.java	6789	2,07	2,09
ConcurrentHashMap.java	7191	1,38	1,39

Таблица 2

Результаты тестирования декларативного форматера для языка Haskell

Имя файла	Количество строк	Время при ширине 200, с	Время при ширине 250, с
JSON.hs	95	0,08	0,07
MangoPay.hs	186	0,09	0,09
Compositions.hs	247	0,23	0,23
Darcs.hs	307	0,19	0,19
HsPretty.hs	403	0,37	0,36
HughesPJ.hs	996	0,41	0,41
TraceTrans.hs	2723	0,73	0,72
Lojban.hs	11732	1,67	1,62

При разработке декларативного форматера языка Java выяснилось, что не только списки являются особым случаем для описанного подхода, но также и конструкция **try-catch-finally**, поскольку она может иметь произвольное число секций **catch**. Сложность данной конструкции была преодолена с помощью подхода, аналогичного альтернативному способу обработки списков. Так, необходимо завести шаблоны перехода от конструкции **try** к первой секции **catch**, от **catch** к следующему **catch**, и от последнего **catch** к **finally**. К сожалению, для этой конструкции пришлось отказаться от сгенерированной по XML-описанию реализации класса-компоненты и написать ее вручную.

Язык Haskell был выбран по другим со-

ображениям. Во-первых, он обладает множеством отличных от языка Java структур, что позволяет проверить выразительность нашей системы. Во-вторых, двумерный синтаксис языка является дополнительным испытанием для работы форматера. Однако никаких проблем, кроме упомянутых в описании общего подхода, при реализации форматера не встретилось.

В табл. 1 и 2 представлены результаты форматирования программных текстов разных размеров на языках Java и Haskell соответственно.

Тестирование производительности проводилось при ограничениях на ширину вывода, равных 200 и 250 символов. Такие ограничения являются серьезным испытанием для нашего подхода, т. к. асимптотика времени работы алгоритма форматирования зависит кубически от максимальной ширины вывода. При этом большие ширины не рассматривались осмысленно, т. к. они на практике не используются.

Файлы для тестирования в случае языка Java выбирались из проекта IntelliJ IDEA Community Edition[5], а в случае Haskell — из различных репозиториев на GitHub[6]. При всех тестах отклонение не превышало 10 %.

Данные приведенных экспериментов показывают, что декларативные форматеры могут использоваться для обработки 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 с.

Декларативные форматеры для языков Java и Haskell доступны в репозитории проекта (https://bitbucket.org/alexeykor/printerplugin).

Декларативные форматеры имеют ряд достоинств и недостатков по сравнению с форматерами современных IDE.

Так, представленные в данной статье форматеры имеют возможность настраиваться на нужный СК по примеру кода. Стоит заметить, что если нужный набор настроек для классических форматеров может быть восстановлен по образцу, то этот их недостаток может быть легко устранен. Однако на данный момент не представлено систем, способных на это. Похожее решение предложено в [3], но оно позволяет получить по образцу исключительно значение базового сдвига для подконструкций. Кроме того, вариативность получаемых СК будет все также ограничена множеством и разнообразием настроек форматеров.

Декларативные форматеры вычисляют результат, удовлетворяющий критерию оптимальности. Однако поиск оптимального представления существенно повышает время форматирования текста. В то же время классические форматеры принимают решения о раскладке поддеревьев локально, с ограниченным предпросмотром, что дает им существенное преимущество в производительности.

Ограничением предложенного подхода является то, что он требует наличия синтаксического анализатора целевого языка, поскольку иначе невозможно получение шаблонов. Однако при реализации декларативного форматера для среды разработки это несущественно, т. к. анализатор уже имеется или нужен еще и для других задач.

Исследование выполнено при поддержке компании JetBrains (http://jetbrains.com).

СПИСОК ЛИТЕРАТУРЫ

1. **Azero P., Swierstra S.D.** Optimal Pretty-Printing Combinators. 1998 [электронный ресурс]/ URL: http://www.cs.ruu.nl/groups/ST/Software/PP

2. **Boulychev D., Koznov D., Terekhov A.A.** On Project-Specific Languages and Their Application in Reengineering // Proc. of the 6th European Conf. on Software Maintenance and Reengineering, IEEE Computer Society. Budapest, 2002.

[5] https://github.com/JetBrains/intellij-community
[6] http://github.com

Pp. 177–185.

3. **Corbo F., Del Grosso C., Di Penta M.** Smart Formatter: Learning Coding Style from Existing Source Code // Software Maintenance. IEEE International Conf. 2007. Pp. 525–526.

4. **Daskalakis C., Karp R.M., Mossel E., Riesenfeld S., Verbin E.** Sorting and Selection in Posets // Proc. of the 2th Annual ACM-SIAM Symposium on Discrete Algorithms. Society for Industrial and Applied.

5. **Hughes J.** The Design of a Pretty-printing Library // Advanced Functional Programming. 1995.

6. **Nielson F., Nielson H.R., Hankin C.** Principles of Program Analysis. Springer-Verlag New York Inc., 1999.

7. **Podkopaev A., Boulytchev D.** Polynomial-Time Optimal Pretty-Printing Combinators with Choice. In Perspectives of System Informatics // Lecture Notes in Computer Science. Berlin. 2015. No. 8974. Pp 257–265.

8. **Wadler P.** A Prettier Printer // The Fun of Programming. Palgrave MacMillan, 2003.

REFERENCES

1. **Azero P., Swierstra S.D.** *Optimal Pretty-Printing Combinators*. 1998. Available: http://www.cs.ruu.nl/groups/ST/Software/PP

2. **Boulychev D., Koznov D., Terekhov A.A.** On Project-Specific Languages and Their Application in Reengineering. *Proceedings of the 6th European Conference on Software Maintenance and Reengineering*, *IEEE Computer Society*. Budapest, 2002, Pp. 177–185.

3. **Corbo F., Del Grosso C., Di Penta M.** Smart Formatter: Learning Coding Style from Existing Source Code, *Software Maintenance. IEEE International Conference*, 2007, Pp. 525–526.

4. **Daskalakis C., Karp R.M., Mossel E., Riesenfeld S., Verbin E.** Sorting and Selection in Posets. *Proceedings of the 2th Annual ACM-SIAM Symposium on Discrete Algorithms. Society for Industrial and Applied*, 2009.

5. **Hughes J.** The Design of a Pretty-printing Library. *Advanced Functional Programming*, 1995.

6. **Nielson F., Nielson H.R., Hankin C.** *Principles of Program Analysis*. Springer-Verlag New York Inc., 1999.

7. **Podkopaev A., Boulytchev D.** Polynomial-Time Optimal Pretty-Printing Combinators with Choice. In Perspectives of System Informatics. *Lecture Notes in Computer Science*, Berlin, 2015, No. 8974, Pp 257–265.

8. **Wadler P.** A Prettier Printer. *The Fun of Programming*. Palgrave MacMillan, 2003.

ПОДКОПАЕВ Антон Викторович – *аспирант кафедры системного программирования Санкт-Петербургского государственного университета*.
199034, Россия, Санкт-Петербург, Университетская наб., д. 7-9.
E-mail: a.podkopaev@2009.spbu.ru

PODKOPAEV Anton V. *St. Petersburg State University*.
199034, Universitetskaya emb. 7-9, St. Petersburg, Russia.
E-mail: a.podkopaev@2009.spbu.ru

КОРОВЯНСКИЙ Алексей Юрьевич – *студент кафедры системного программирования Санкт-Петербургского государственного университета*.
199034, Россия, Санкт-Петербург, Университетская наб., д. 7-9.
E-mail: aleksei.korovianskii@student.spbu.ru

KOROVIANSKII Aleksei Yu. *St. Petersburg State University*.
199034, Universitetskaya emb. 7-9, St. Petersburg, Russia.
E-mail: aleksei.korovianskii@student.spbu.ru

ОЗЕРНЫХ Игорь Станиславович – *студент кафедры системного программирования Санкт-Петербургского государственного университета*.
199034, Россия, Санкт-Петербург, Университетская наб., д. 7-9.
E-mail: st011628@student.spbu.ru

OZERNYKH Igor S. *St. Petersburg State University*.
199034, Universitetskaya emb. 7-9, St. Petersburg, Russia.
E-mail: st011628@student.spbu.ru