Санкт-Петербургский Государственный Университет
Математико-механический факультет
Кафедра системного программирования

Форматирование текста программ на основе комбинаторов, сопоставления с образцом и синтаксических шаблонов

Курсовая работа студента 445 группы
Подкопаева Антона Викторовича

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

Санкт-Петербург
2013

Содержание

Введение

С появлением первых языков программирования особую важность приобрели языковые процессоры. **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.

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.

Достаточно часто 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**.

Одной из проблем, возникающей при разработке принтера, является то, что критерии качества его работы трудно формализуемы. Очевидно, что одного соответствия с точки зрения синтаксиса и семантики полученного кода и древовидного представления недостаточно. Рассмотрим программы на рисунках 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

Они эквиваленты синтаксически и семантически с точки зрения компилятора C, но для пользователя вариант с рисунка 2 предпочтительней, так как он проще для восприятия. Как мы видим, при отображении данных необходимо явным для пользователя образом представлять иерархическую структуру информации. В большинстве случаев решение этой задачи сводится к добавлению или удалению символов, не несущих информации для синтаксического анализа, или другому преобразованию текста без изменения его семантики.

Кроме того определенную сложность в описание и исполнение конкретного принтера вносит тот факт, что в большинстве случаев нельзя однозначным образом сопоставить синтаксическую конструкцию с единственным представлением. Необходима вариативность в зависимости от дополнительных условий, наложенных на результат его работы.

Рассмотрим небольшой пример. Пользователь задает условие вида: “последовательные операторы пишутся на одной строке, если помещаются в N символов, а иначе — на разных строках”.

На рисунке 3 изображено синтаксическое дерево последовательности двух операторов. Такое дерево, согласно заданному правилу, может быть напечатано одним из двух вариантов (рисунки 4, 5).

Выбор происходит в зависимости от ширины вывода. Так, при ширине равной 35 символов (длина строки «System.out.println(“Hello world”); »), должен выбираться вариант,

```
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

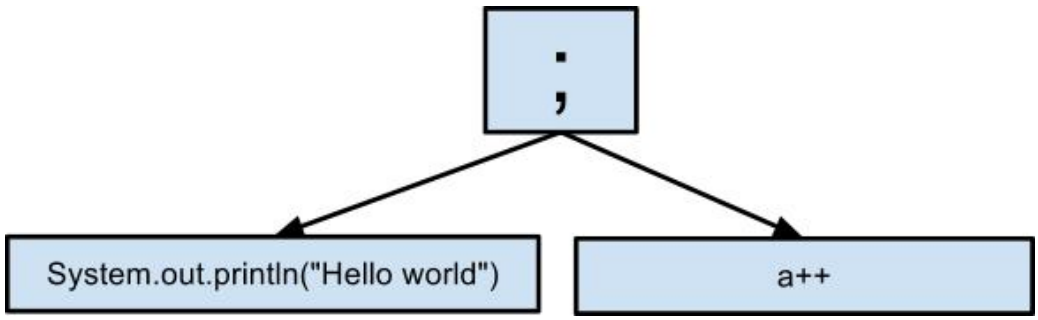


Рис. 3: Последовательные операторы

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

Рис. 4: Последовательные операторы в строчку

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

Рис. 5: Последовательные операторы в несколько строк

изображенный на рисунке 5, так как код на рисунке 4 имеет ширину более 35 символов. Могут быть заданы и более сложные условия.

Рассмотрим другой пример. Пусть нам нужно текстовое представление синтаксического дерева конструкции “if”, и заданы шаблоны с рисунков 6a и 6b, причем вариант, изображенный на рисунке 6a выбирается в случае, если условие и ветки могут быть напечатаны в одну строчку.

Тогда для деревьев, представленных на рисунках 7a и 8a, будут напечатаны коды с рисунков 7b и 8b соответственно.

То, что не существует четких критериев красоты кода, а также отсутствие библио-

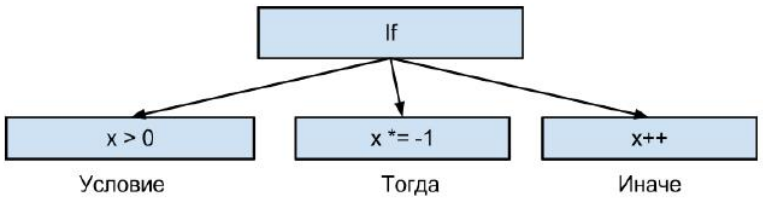
```
if 'expression' then 'branch 1'
               else 'branch 2'
```

(a)

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

(b)

Рис. 6: Представления для конструкции “if”

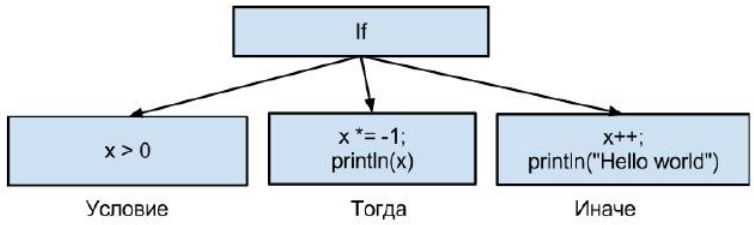


(a)

```
if x > 0 then x *= −1
         else x++
```

(b)

Рис. 7: Использование представления с рис. 6a



(a)

```
if x > 0
then x *= −1;
     println(x)
else x++;
     println("Hello world")
```

(b)

Рис. 8: Использование представления с рис. 6b

тек, предоставляющих возможность описать принтер в виде, подобном рассмотренному примеру с конструкцией “if”, то есть с помощью шаблонов, часто приводит к тому, что принтеры становятся крайне сложными и наполненными эвристиками.

Задачей данной работы было изучение существующих подходов к построению принтеров в рамках функциональных языков программирования и разработка способа определения принтеров с помощью синтаксических шаблонов.

1 Обзор существующих библиотек

В рамках исследования был проведен анализ основных существующих функциональных принтер-библиотек. Все выбранные библиотеки оказались комбинаторными, что естественно для функциональных языков.

1.1 Модельный язык L

В дальнейшем центральным примером, для которого мы будем разрабатывать принтеры, будет модельный язык L. Именно для этого языка будет реализован шаблонный принтер.

Язык L состоит из небольшого числа операторов:

1. присваивание;
2. цикл с предусловием;
3. ветвление;
4. последовательное выполнение;
5. чтение с занесением в переменную;
6. печать целочисленного выражения.

Также в языке есть выражения. Выражения бывают трех типов:

1. константа;
2. переменная;
3. бинарная операция.

На рисунке 9 приведен пример программы на языке L. В данном случае, это программа, которая считывает с консоли два числа, а потом возводит второе число в степень, равную первому.

```
{
        read(k);
        read(n);
        r = 1;
        while k > 0
              do if k % 2 == 1
                    then r = r * n
                    else skip;
                 n = n * n;
                 k = k / 2;
        write(r)
}
```

Рис. 9: Быстрое возведение в степень на языке L

1.2 Библиотека Хьюза

Библиотека Джона Хьюза[2] считается первой комбинаторной принтер-библиотекой. Она основана на алгоритме, предложенном Дереком Оппеном [4], и по сути является его реализацией в функциональном стиле на языке Haskell[1]. Также библиотека Хьюза, расширенная Саймоном Пейтоном Джонсом [3], является стандартной принтер-библиотекой для языка Haskell.

В данной библиотеке ключевым типом является "`Doc`". Он представляет документ, который потом может быть переведен в строковое представление. Основные комбинаторы для составления документа:

```
text :: String → Doc
(<>) :: Doc → Doc → Doc
($$) :: Doc → Doc → Doc
nest :: Int → Doc → Doc
sep  :: [Doc] → Doc
```

Так, с помощью функции "`text`" по строке получается документ, оператор "`<>`" соединяет два документа горизонтально (см. рисунок 10a), а оператор "`$$`" соединяет документы вертикально (см. рисунок 10b).

(a) Комбинатор "`<>`" (b) Комбинатор "`$$`"

Рис. 10: Пример работы комбинаторов

Функция "`nest`" добавляет к каждой строке документа заданное число ведущих пробелов. Функция "`sep`" является ключевым комбинатором, который в этой библиотеке позволяет задавать плавающую раскладку документа. Она принимает как параметр список документов, а на выходе получается документ, который представляет из себя либо вертикальную склейку элементов списка, либо горизонтальную склейку (в этом случае если к документу из списка применялась функция "`nest`", то к этому документу не добавляются ведущие пробелы, то есть применение "`nest`" попросту игнорируется), причем между документами вставляется пробельный символ. Вариант раскладки выбирается функцией "`pretty`":

```
pretty :: Int → Int → Doc → String
```

Кроме самого документа, функция "`pretty`" также принимает два числа: максимальную длину и максимальную наполненность строки. Здесь наполненность строки значит длину текста без ведущих пробельных символов. В ходе работы этой функции и происходит выбор раскладки документа, полученного с помощью комбинатора "`sep`". Если горизонтальная раскладка удовлетворяет ограничениям на ширину строки, то она и выбирается. Иначе выбирается вертикальная раскладка.

Рассмотрим пример описания принтера с помощью библиотеки Хьюза. Для этого используем учебный язык L. Часть принтера для языка L, отвечающая за представление операторов, показана на рисунке 11. В примере используется не описанный выше комбинатор "`<+>`", который определяется следующим образом:

```
a <+> b = a <> (text "␣") <> b
```

В данном случае принтер получился несложным, но абсолютно не наглядным. Поскольку в библиотеке нет механизмов, позволяющих явно варьировать раскладку документа в

[1] http://haskell.org

```
docFromOperation :: Operation → Doc

docFromOperation (varName `Assignment` exp) =
        sep[(text varName) <+> (text "="), docFromExpression exp]

docFromOperation (leftOp `Sequence` rightOp) =
        sep[(docFromOperation leftOp) <> (text ";"), docFromOperation rightOp]

docFromOperation (IfThenElse exp trueOp falseOp) =
        (text "if") <+> sep[(docFromExpression exp),
                (text "then") <+> (docFromOperation trueOp),
                (text "else") <+> (docFromOperation falseOp)]

docFromOperation (exp `WhileDo` loopOp) =
        (text "while") <+> sep [(docFromExpression exp),
                (text "do") <+> (docFromOperation loopOp)]

docFromOperation (Read varName) =
        (text "read(") <> (text varName) <> text(")")

docFromOperation (Write exp) =
        (text "write(") <> (docFromExpression exp) <> text(")")
```

Рис. 11: Принтер, написанный с помощью библиотеки Хьюза

зависимости от раскладки его поддокументов, невозможно выразить пример с рисунка 12. То есть, в случае многострочного выражения в операторе "write", напечатать закрывающую скобку на уровне самого оператора.

```
write(longNameVariable
      +
      anotherLongNameVariable
)

write(littleOne)
```

Рис. 12: Желательный пример печати конструкции "write"

С помощью реализации принтера с рисунка 11, в данном случае для оператора "write" получится немного не то (см. рисунок 13).

```
write(longNameVariable
      +
      anotherLongNameVariable)

write(littleOne)
```

Рис. 13: Результат для изначального принтера конструкции "write"

Если попробовать поменять функцию “docFromOperation” для конструкции “write” (см. рис. 14), то для многострочного выражения все получится правильно, но в случае однострочного — появится лишний пробел перед закрывающей скобкой (см. рис. 15)).

```
docFromOperation (Write exp) =
        sep [text "write(" <> (docFromExpression exp); text ")"]
```

Рис. 14: Измененный принтер конструкции “write”

```
write(longNameVariable
      +
      anotherLongNameVariable
)

write(littleOne )
```

Рис. 15: Результат для измененого принтера конструкции “write”

1.3 Библиотека Вадлера

В [5] Филипп Вадлер описал свою комбинаторную бибилиотеку для форматированного вывода на языке Haskell. Она является модификацией библиотеки Хьюза, описанной в предыдущем разделе. Код библиотеки сократился с $\approx$ 110 строк до $\approx$ 80 строк, и, по исследованию Вадлера, на 30% увеличилась скорость вычисления раскладки документа.

Рассмотрим основые комбинаторы этой библиотеки:

```
(<>)  :: Doc → Doc → Doc
nil   :: Doc
text  :: String → Doc
line  :: Doc
nest  :: Int → Doc → Doc
group :: Doc → Doc
```

Вадлер решил отказаться от двух разных способов соединения документов, оставив лишь горизонтальную склейку. Но для того, чтобы можно было выражать не только однострочные документы, в библиотеке Вадлера появилась функция "`line`". Она создает документ, который может быть переведен в символ новой строки или в пробел. Функция "`group`" имеет то же назначение, что и оператор "`sep`" в библиотеке Хьюза, но работает не со списком документов, а с одним документом, и по сути предоставляет альтернативы для алгоритма перевода документа в "`String`": в документе, на который подействовал "`group`", либо все вхождения "`line`" заменяются на пробел, либо остаются переводами строки (если они не являются частью вложенных "`group`"-документов).

В таком виде библиотека потеряла в выразительности, что признается в статье Вадлера. Но кроме потери выразительности, есть еще один серьезный недостаток, возникающий из-за оператора "`group`". То, что любой документ им может быть преобразован в однострочный, делает библиотеку неприменимой в некоторых ситуациях.

Рассмотрим следующий пример. Пусть нам надо написать принтер для языка Python[2]. Для конструкции последовательных операторов принтер изображен на рисунке 16. Подругому его написать нельзя — мы хотим, чтобы последовательные операторы печатались на новых строках. Но если такая конструкция попадет внутрь "`group`"-документа, то последовательные строчки могут склеиться пробелом, что сделает код некорректным, так как в Python несколько операторов на одной строке должны разделяться символом ";".

```
docFromOperation (leftOp `Sequence` rightOp) =
        (docFromOperation leftOp) <> line <> docFromOperation rightOp
```

Рис. 16: Принтер для последовательных операторов в Python

Так корректный код (см. рисунок 17a) может превратиться в некорректный (см. рисунок 17b).

```
print 5
print 6
```

(a)

```
print 5 print 6
```

(b)

Рис. 17: Пример работы принтера для языка Python

[2] http://python.org

1.4 Библиотека Азеро и Свиерстры

Библиотека Азеро и Свиерстры[3], описанная в [1], отличается от предыдущих библиотек тем, что дает возможность явным образом задать несколько несвязанных вариантов раскладки документа. В этой библиотеке есть комбинатор "<|>":

```
(<|>) :: Doc → Doc → Doc
```

Этот комбинатор берет два документа и создает новый, который при раскладке может стать первым или вторым, в зависимости от того, какой из документов раскладывается оптимальней. *Оптимальной* раскладкой для документа считается раскладка, удовлетворяющая ограничению на ширину документа и имеющая минимальную высоту.

Наличие комбинатора "<|>" сразу же решает проблему со скобкой, которая была поднята в обзоре библиотеки Хьюза (см. рис. 18).[4]

```
docFromOperation (Write exp) =
       hor1_vert ((text "write(") <> (docFromExpression exp)) (text ")")

hor1_vert :: Doc → Doc → Doc
hor1_vert a b = (a $$ b) <|> (element_h1 (a <> b))
```

Рис. 18: Принтер конструкции "`write`", удовлетворяющий примеру с рис. 12

Библиотека Азеро и Свиерстры обладает самым богатым набором комбинаторов и, благодаря оператору "<|>", позволяет выразить практически любые принтеры. Но, также как остальные рассмотренные библиотеки, не дает механизмов для простого и наглядного задания принтеров.

[3] В данном тексте, с целью не усложнять восприятие, изменены обозначения комбинаторов библиотеки Свиерстры на обозначения, подобные тем, что были уже рассмотрены в библиотеке Хьюза. Семантика комбинаторов описана без изменений, согласно оригинальной статье и соответствующей библиотеке.

[4] В примере используется функция "`element_h1`". Эта функция выбирает из вариантов раскладки документа те, которые имеют высоту 1.

2 Реализация принтера, основанного на шаблонах

Основной целью данной работы была разработка прототипа принтера, который бы использовал для печати шаблоны.

2.1 Описание общего подхода

Работа принтера разбивается на два этапа: подготовка шаблонов и непосредственно печать синтаксического дерева.

На подготовительном этапе из файла с шаблонами языковых конструкций строится набор образцов. *Образцом* назовем пару из текста шаблона и обобщенного синтаксического дерева шаблона. Дерево шаблона — обобщенное, так как на месте некоторых узлов стоят специальные метки, которые хранят информацию об ограничениях на текстовое представление соответствующих узлов.

Во время основной фазы работы принтера дерево, которое необходимо непечатать, сравнивается с деревьями из образцов. В случае согласованности образца и дерева для печати используется текст образца, в который на места меток вставляются представления соответствующих поддеревьев.

2.2 Шаблон

Внутри шаблона используется специальный язык разметки. Символы "`@−`" на позиции поддерева синтаксической конструкции означают, что для применения данного шаблона поддерево должно быть напечатано в одну строку. Семантика символов "`@* N @*`" совпадает с семантикой "`@−`" с точностью до того, что напечатанное поддерево должно занимать строку длины не более N. Символы "`@| @|`" означают, что соответствующее поддерево может быть напечатано и на нескольких строках. Для выделения отдельных шаблонов используются строки "`t_start`", "`t_end`".

Рассмотрим пример шаблонов для конструкции "`write`" языка L (см. рис. 19a и 19b). Эти шаблоны задают именно такое представление `write`", которого мы добивались в обзоре принтер-библиотек.

```
t_start
write(@−)
t_end
```

(a)

```
t_start
write(@|
      @|
)
t_end
```

(b)

Рис. 19: Шаблоны для конструкции "`write`"

Рассмотрим шаблоны для конструкции "`if`−`then`−`else`" (см. рис. 20a и 20b). С помощью них можно напечатать дерево, изображенное на рисунке 21 (см. рис. 22).

Одним из неочевидных свойств шаблонов является то, что с их помощью можно выражать не только базовые конструкции языка. Например, можно завести отдельный шаблон для случая, когда обе ветки конструкции "`if`−`then`−`else`" представляют собой операторы "`write`" (см. рис. 23). Если добавить такой шаблон, то пример дерева с рисунка 21 получит новое представление (см. рис. 24).

```
t_start
if @* 5 @* then @* 5 @* else @* 5 @*
t_end
```

(a) Однострочный вариант

```
t_start
if @|
   @|
then @|
     @|
else @|
     @|
t_end
```

(b) Многострочный вариант

Рис. 20: Шаблоны для конструкции “if−then−else”

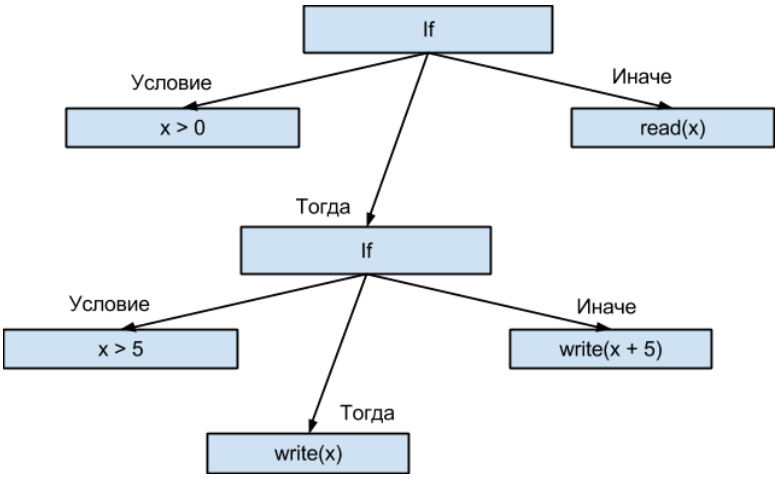


Рис. 21: Пример дерева “if−then−else”

```
if x > 0
then if x > 5
     then write(x)
     else write(x + 5)
else read(x)
```

Рис. 22: Представление с помощью шаблонов с рис. 20

```
if @− then write(@−)
      else write(@−)
```

Рис. 23: Пример задания шаблона для сложной конструкции

2.3 Построение образцов

Для работы принтера неообходимо иметь возможность сравнивать синтаксическое представление шаблонов с деревом, которое печатается. Поэтому необходим синтаксический

```
if x > 0
then if x > 5 then write(x)
              else write(x + 5)
else read(x)
```

Рис. 24: Представление дерева с рис. 21 с использованием шаблона с рис. 23

анализатор, который может разбирать шаблонные конструкции в рамках целевого языка. Удобным способом разработать данный анализатор является использование расширяемого синтаксического анализатора целевого языка. В результате работы расширенного анализатора получается набор образцов. В узлах синтаксического дерева, соответствующих меткам шаблонов, хранится информация о положении метки внутри текста образца, чтобы на этапе печати документа знать, куда вставлять представления поддеревьев.

2.4 Печать дерева с использованием набора образцов

На этапе, когда уже имеются необходимые образцы, происходит их сопоставление с синтаксическим деревом, переданным на печать. Псевдокод алгоритма приведен ниже (см. рис. 25).

```
H — ассоциативный массив, связывает деревья с их текстовым представлением
M — набор образцов для языковых конструкций

Комментарий: По дереву строит его текстовое представление
function PRINT(tree)
    if tree ∈ H then
        return H[tree]
    else
        Вызов print для поддеревьев tree
        return TEMPLATEITER(tree)
    end if
end function

Комментарий: Перебирает все образцы и пытается их применить к дереву
function TEMPLATEITER(tree)
    for all (templateTree, text) ∈ M do
        Комментарий: В случае исключения, переходит к новому элементу M
        list := TEMPLATECOMPARE(tree, templateTree)
        H[tree] := построенное представление для tree по list и text
        return H[tree]
    end for
end function

Комментарий: Возвращает список координат метки в тексте шаблона и соответствующее
текстовое представление поддерева
function TEMPLATECOMPARE(tree, templateTree)
    if tree и templateTree одного типа then
        Вызвать templateCompare для соответствующих поддеревьев
        return соединенный список результатов вызова для поддеревьев
    end if
    if templateTree является меткой then
        if H[tree] соответствует ограничениям метки then
            return [(координаты метки, H[tree])]
        else
            создать исключение
        end if
    end if
    Создать исключение, т.к. переданные деревья разной структуры
end function
```

Рис. 25: Псевдокод сопоставления дерева с набором образцов

Из псевдокода видно, что в случае, если для какого-нибудь поддерева не найдется соответствующий образец, то дерево невозможно будет напечатать. То есть множество образцов должно быть достаточным. Это естественное ограничение.

Попробуем оценить время работы алгоритма. Для каждого узла дерева, переданного принтеру, вычисляется следующее:

1. текстовое представление для детей;
2. сравнение с имеющимися образцами;
3. текстовое представление узла по выбранному образцу и представлениям детей.

Текстовое представление вычисляется для каждого узла один раз. Сравнение с образцами занимает $O(B \times |M|)$ времени, где B — максимальное число узлов в дереве образца, а $|M|$ — количество образцов. Построение текстового представления по выбранному образцу занимает $O(A)$, где A — максимальное количество меток в шаблоне, но так как очевидно, что $A \leq B$, то оценку можно заменить на $O(B)$. Таким образом, оценка на работу алгоритма для дерева с T узлами равна $O(T \times B \times |M|)$.

Для сравнения, если реализовать аналогичный принтер с помощью библиотеки Азеро и Свиерстры, то это приведет к экспоненциальному от размера дерева расчету текстового представления. В случае библиотек Хьюза и Вадлера похожий принтер будет иметь квадратичную сложность.

2.5 Реализованный принтер

Описанный подход был реализован на примере принтера языка L, написанного на языке OCaml[5]. Для написания синтаксического анализатора была использована библиотека Ostap[6].

В качестве примера работы принтера с разными шаблонами рассмотрим уже упоминавшуюся программу быстрого возведения в степень (см. рис. 9). С помощью шаблонов, приведенных в дополнении A, принтер печатает программу в виде, изображенном на рисунке 26.

```
{
  { read(k); read(n) };
  r := 1;
  while (k > 0) do
    {
      if ((k % 2) != 0)
      then r := (r * n)
      else skip;
      n := (n * n);
      k := (k / 2)
    };
  write(r)
}
```

Рис. 26: Программа быстрого возведения в степень на языке L, напечатанная с помощью шаблонов из дополнения A

Шаблоны, приведенные в дополнении B, представляют программу несколько иным образом (см. рис. 27).

```
{ { read(k); read(n) };
  r := 1;
  while
    (k > 0)
  do
    { if ((k % 2) != 0) then r := (r * n)
                 else skip;
      n := (n * n);
      k := (k / 2) };
  write(r) }
```

Рис. 27: Программа быстрого возведения в степень на языке L, напечатанная с помощью шаблонов из дополнения B

Из приведенных примеров виден один из недостатков реализации. Если присмотреться к конструкции if−then−else, то можно заметить, что then и else находятся на разных уровнях. Естественно, это нежелательный результат. Пока эта проблема не решена, в дальнейшем планируется разобраться с этой проблемой путем расширения языка шаблонов.

[5]http://ocaml.org
[6]http://oops.math.spbu.ru/projects/ostap

3 Заключение

В рамках данной курсовой работы были изучены основные подходы по построению принтеров в рамках функциональных языков, разработана методика задания принтера целевого языка с помощью шаблонов, реализован принтер для языка L. Также в ходе работы был изучен язык OCaml, на котором была выполнена реализация данного подхода.

В рамках развития работы планируется создать библиотеку для языка OCaml, которая позволит легко реализовывать описанный подход.

Список литературы

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

[2] Hughes J. The Design of a Pretty-printing Library // Advanced Functional Programming. Springer Verlag. 1995. P. 53-96.

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

[4] Oppen D. Pretty Printing // Stanford Verification Group. Report No. 13. Computer Science Department Report No. STAN-CS-79-770. 1979.

[5] Wadler P. A Prettier Printer // Journal of Functional Programming. Palgrave Macmillan. 1998. P.223-244.

A Первый вариант набора шаблонов для языка L

```
t_start
write(@−)
t_end

t_start
write(@|
      @|
)
t_end

t_start
if @∗ 10 @∗ then @∗ 10 @∗ else @∗ 10 @∗
t_end

t_start
if @|
   @|
then @|
     @|
else @|
     @|
t_end

t_start
{ @∗ 10 @∗; @∗ 10 @∗; @∗ 10 @∗; @∗ 10 @∗ }
t_end

t_start
{
  @|
  @|;
  @|
  @|;
  @|
  @|;
  @|
  @|
}
t_end

t_start
{ @∗ 15 @∗; @∗ 15 @∗; @∗ 15 @∗ }
t_end

t_start
{
  @|
  @|;
```

```
  @|
  @|;
  @|
  @|
}
t_end

t_start
{ @* 20 @*; @* 20 @* }
t_end

t_start
{
  @|
  @|;
  @|
  @|
}
t_end

t_start
while @- do while @- do
  @|
  @|
t_end

t_start
while @- do
  @|
  @|
t_end

t_start
while @|
      @|
do
  @|
  @|
t_end

t_start
(@* 10 @* + @* 10 @*)
t_end

t_start
(@|
@|
  +
@|
@|)
```

```
t_end

t_start
(@* 10 @* - @* 10 @*)
t_end

t_start
(@|
@|
  -
@|
@|)
t_end

t_start
(@* 10 @* * @* 10 @*)
t_end

t_start
(@|
@|
  *
@|
@|)
t_end

t_start
(@* 10 @* / @* 10 @*)
t_end

t_start
(@|
@|
  /
@|
@|)
t_end

t_start
(@* 10 @* % @* 10 @*)
t_end

t_start
(@|
@|
  %
@|
@|)
t_end
```

```
t_start
(@* 10 @* > @* 10 @*)
t_end

t_start
(@|
@|
  >
@|
@|)
t_end

t_start
(@* 10 @* < @* 10 @*)
t_end

t_start
(@|
@|
  <
@|
@|)
t_end

t_start
(@* 10 @* >= @* 10 @*)
t_end

t_start
(@|
@|
  >=
@|
@|)
t_end

t_start
(@* 10 @* <= @* 10 @*)
t_end

t_start
(@|
@|
  <=
@|
@|)
t_end


t_start
```

```
(@* 10 @* != @* 10 @*)
t_end

t_start
(@|
@|
  !=
@|
@|)
t_end
```

B Второй вариант набора шаблонов для языка L

```
t_start
write(@−)
t_end

t_start
write(@|
      @|)
t_end

t_start
if @∗ 20 @∗ then @∗ 20 @∗
            else @∗ 20 @∗
t_end

t_start
if @|
   @|
then @|
     @|
else @|
     @|
t_end

t_start
{ @∗ 10 @∗; @∗ 10 @∗; @∗ 10 @∗; @∗ 10 @∗ }
t_end

t_start
{ @|
  @|;
  @|
  @|;
  @|
  @|;
  @|
  @| }
t_end

t_start
{ @∗ 15 @∗; @∗ 15 @∗; @∗ 15 @∗ }
t_end

t_start
{ @|
  @|;
  @|
  @|;
  @|
```

```
  @| }
t_end

t_start
{ @* 20 @*; @* 20 @* }
t_end

t_start
{ @|
  @|;
  @|
  @| }
t_end

t_start
while
  @—
do
  @|
  @|
t_end

t_start
while
  @—
do
  @—
t_end

t_start
while
  @|
  @|
do
  @|
  @|
t_end

t_start
(@* 10 @* + @* 10 @*)
t_end

t_start
(
  @|
  @|
+
  @|
  @|
)
```

```
t_end

t_start
(@* 10 @* - @* 10 @*)
t_end

t_start
(
  @|
  @|
-
  @|
  @|
)
t_end

t_start
(@* 10 @* * @* 10 @*)
t_end

t_start
(
  @|
  @|
*
  @|
  @|
)
t_end

t_start
(@* 10 @* / @* 10 @*)
t_end

t_start
(
  @|
  @|
/
  @|
  @|
)
t_end

t_start
(@* 10 @* % @* 10 @*)
t_end

t_start
(
```

```
  @|
  @|
%
  @|
  @|
)
t_end

t_start
(@* 10 @* > @* 10 @*)
t_end

t_start
(
  @|
  @|
>
  @|
  @|
)
t_end

t_start
(@* 10 @* < @* 10 @*)
t_end

t_start
(
  @|
  @|
<
  @|
  @|
)
t_end

t_start
(@* 10 @* >= @* 10 @*)
t_end

t_start
(
  @|
  @|
>=
  @|
  @|
)
t_end
```

```
t_start
(@* 10 @* <= @* 10 @*)
t_end

t_start
(
  @|
  @|
<=
  @|
  @|
)
t_end


t_start
(@* 10 @* != @* 10 @*)
t_end

t_start
(
  @|
  @|
!=
  @|
  @|
)
t_end
```