Правительство Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Санкт-Петербургский государственный университет»
Математико-механический факультет

Кафедра Системного Программирования

Озерных Игорь Станиславович

# Генерация декларативных принтеров по грамматике в форме Бэкуса-Наура

Бакалаврская работа

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

Рецензент:
асп. Подкопаев А. В.

Санкт-Петербург
2016

SAINT PETERSBURG STATE UNIVERSITY
Mathematics and Mechanics Faculty

Software Engineering Chair

Igor Ozernykh

# Declarative printer generation in grammar in the Backus-Naur form

Bachelor's Thesis

Scientific supervisor:
associate professor Boulytchev D. IU.

Reviewer:
assistant Podkopaev A. V.

Saint Petersburg
2016

# Оглавление

**Введение** 4

**1. Постановка задачи** 8

**2. Обзор** 9

2.1. Подходы к заданию принтеров и форматеров 9

2.1.1. Задание форматеров по описанию синтаксиса 9

2.1.2. Форматеры, встроенные в IDE 10

2.2. Grammar-Kit 11

**3. Метод генерации декларативных принтеров** 15

3.1. Определение поддеревьев и их свойств для правил грамматики 15

3.2. Необходимые преобразования правил грамматики 17

3.2.1. Устранение левой рекурсии 17

3.2.2. Другие правила грамматики 18

3.3. Поиск правил, описывающих списочных структуры 18

3.4. Итоги 19

**4. Реализация** 20

4.1. Изменение архитектуры принтер-плагина 20

4.2. Генератор принтеров в плагине Grammar-Kit 21

4.3. Генерация файловой компоненты 22

4.4. Ограничения 23

**5. Апробация** 24

5.1. Принтер для языка While 24

5.2. Принтер для языка Erlang 26

5.3. Итоги 28

**Заключение** 29

**Список литературы** 30

# Введение

Интегрированные среды разработки (Integrated Development Environment, IDE) являются неотъемлемой частью современного программирования и разработки программных продуктов. Появившись в 1970-х годах, они прошли длительный путь развития и совершенствования. В наши дни типичная среда разработки включает в себя текстовый редактор, компилятор и/или интерпретатор, средства автоматической сборки, отладчик и другие инструменты. Популярными IDE для языка Java являются Eclipse[1] и IntelliJ IDEA[2]. Они так же имеют средства для работы с системами контроля версий, рефакторинга, форматирования кода. Кроме того, существует возможность расширения функциональности этих IDE со стороны пользователей путем создания плагинов — программных компонентов, добавляющих новые возможности. В частности, плагины могут добавлять поддержку новых языков в IDE. Для разработки подобного плагина необходимо в большинстве случаев реализовать синтаксический анализатор целевого языка, который в случае IntelliJ IDEA можно получить с помощью плагина Grammar-Kit по грамматике этого языка в форме Бэкуса-Наура. Так же, при реализации поддержки нового целевого языка, часто возникает потребность в реализации форматера для программ на этом языке.

Задача автоматического форматирования текстов программ является классической в контексте различных языковых процессоров. Она подразумевает построение текста программы по ее синтаксическому дереву или дереву разбора. Для каждого дерева существует множество различных текстовых представлений. Так, на рис. 1 приведен пример синтаксического дерева оператора ветвления языка C, а на рис. 2 различные текстовые представления этого дерева. Программный компонент, занимающийся форматированием, мы будем называть *принтером.* Как правило, результат работы принтера должен соответствовать

[1] https://eclipse.org

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

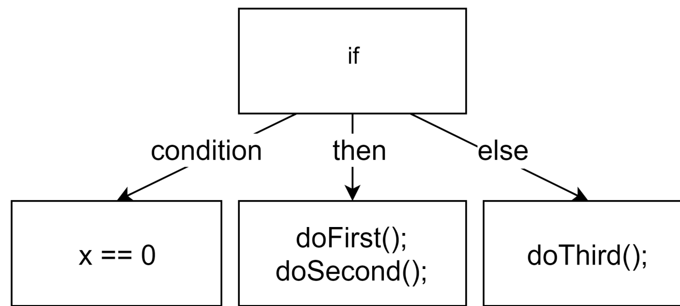


Рис. 1: Представление оператора ветвления в виде дерева разбора

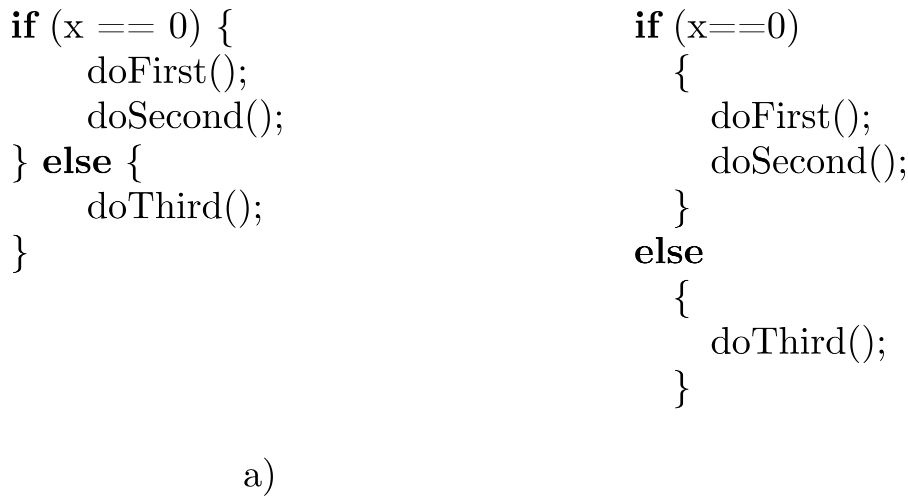


Рис. 2: Текстовые представления синтаксического дерева с рис. 1

некоторому *стандарту кодирования* (СК, coding convension) — набору правил и соглашений, используемых при написании кода на соответствующем языке программирования. Так, стандарт кодирования для языка Java задает расположение фигурных скобок и операторов внутри тела функции, размер и формат отступов и др.

Одним из способов форматирования программных текстов является метод, основанный на *синтаксических шаблонах* [7]. Под *шаблоном* понимаются данные, сопоставление которых с элементом синтаксического дерева дает текстовое представление этого элемента (и его потомков). На рис. 3, *а* представлен шаблон форматирования, который может быть применен к дереву разбора с рис. 1. На рис. 3, *б* представлен результат применения шаблона к этому дереву.

Такой подход был применен в работе [7] при разработке принтер-плагина для среды разработки IntelliJ IDEA. На рис. 4 приведен про-

```
if (@condition) {
    @expr
} else {
    @expr
}
```

а) Шаблон для оператора ветвления

```
if (x == 0) {
    doFirst();
    doSecond();
} else {
    doThird();
}
```

б) Текст, полученный при применении шаблона к дереву разбора

Рис. 3: Оператор ветвления и шаблон для него

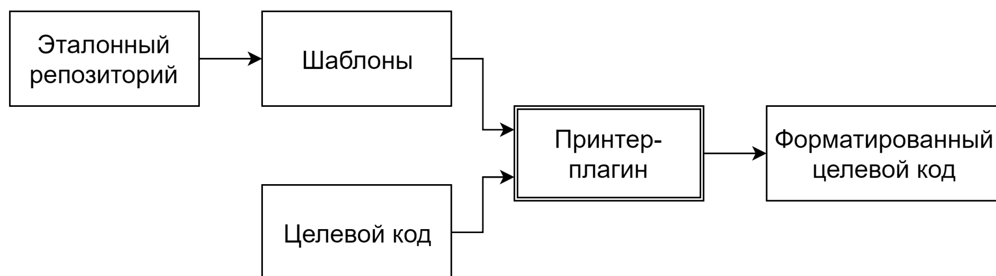


Рис. 4: Принцип работы принтер-плагина

цесс работы принтер-плагина. На вход принтеру подаются содержащий требуемое форматирование код (эталонный репозиторий), из которого извлекаются шаблоны форматирования, и код, который необходимо отформатировать (целевой код). Полученные шаблоны применяются к целевому коду, и результатом работы принтера становится отформатированный целевой код.

Список поддерживаемых принтер-плагином языков можно расширять. Для этого необходимо создать языкозависимую прослойку между ядром платформы и представлением синтаксического дерева, полученного в результате работы анализатора. Метод, описанный в [7], позволяет генерировать такую прослойку по языкозависимому описанию *компонент принтера* — классов, описывающих, как форматировать соответствующие элементы синтаксического дерева. Описание каждой компоненты принтера (component) является XML-файлом. Каждый такой файл содержит имя компоненты (name), класс внутреннего представления в IDE (psiComponentClass), а так же список поддеревьев (subtree)

и их свойств (name, psiGetMethod, hasSeveralElements, isRequired). На рис. 5 приведено XML-описание компоненты принтера, соответствующей циклу с предусловием. Современные языки программирования содержат большое число структур, поэтому необходимо большое число подобных XML-описаний для задания принтера языка. Процесс созда-

```
<component name="While" psiComponentClass="PsiWhileStmt">
    <subtree name="condition" psiGetMethod="Bexpr"
            hasSeveralElements="false" isRequired="true" />
    <subtree name="body" psiGetMethod="StmtList"
            hasSeveralElements="true" isRequired="false" />
</component>
```

Рис. 5: Описание компоненты принтера, форматирующей циклы с предусловием

ния такого описания трудоемок и требует глубоких знаний о системе [7].

Грамматика языка, которая используется в Grammar-Kit и по которой генерируется код синтаксического анализатора, содержит большую часть необходимой информации для создания этой языкозависимой прослойки. Поэтому принтер для целевого языка можно получать по грамматике этого языка.

# 1. Постановка задачи

Целью данной работы является автоматизация процесса задания декларативных принтеров путем их генерации по грамматике языка в форме Бэкуса-Наура. Данная грамматика также используется для получения синтаксического анализатора программ на целевом языке с помощью плагина Grammar-Kit.

Поставлены следующие задачи:

- разработка методики генерации декларативных принтеров по грамматике в форме Бэкуса-Наура;
- интеграция метода в проект Grammar-Kit путем реализации генератора принтеров;
- реализация интеграции полученных принтеров с принтер-плагином для IDE IntelliJ IDEA;
- апробация метода на основе грамматики языков While [4] и Erlang.

While — учебный язык, содержащий самые основные конструкции. На его примере производилась апробация метода генерации принтеров по языкозависимому описанию компонент [7]. Erlang[3] — промышленный язык программирования, для которого уже существует грамматика, по которой генерировался синтаксический анализатор с помощью плагина Grammar-Kit.

[3]https://www.erlang.org

# 2. Обзор

В данном обзоре рассматриваются некоторые подходы к заданию принтеров и форматеров, а также плагин Grammar-Kit для IntelliJ IDEA, позволяющий по БНФ-грамматике задавать синтаксический анализатор целевого языка. Используемые плагином грамматики рассматриваются на примере грамматики языка While [4].

## 2.1. Подходы к заданию принтеров и форматеров

Рассмотрим некоторые подходы к заданию принтеров и форматеров.

### 2.1.1. Задание форматеров по описанию синтаксиса

Существуют различные подходы к заданию принтеров для целевого языка. Один из них описан в [6]. Авторы предлагают язык описания синтаксиса, по которому можно получить и синтаксический анализатор, и принтер для языка. Описание синтаксиса состоит из правил, которые схожи с правилами формальных грамматик, но которые также задают и стиль форматирования для данной структуры языка. Приведенное ниже правило вывода описывает основные арифметические выражения:

templates
  Exp.**Num** = <<INT>>
  Exp.Plus = <<Exp> + <Exp>>
  Exp.Times = <<Exp> * <Exp>>

Первое преобразование определяет шаблон для выражений, состоящих из чисел. Остальные преобразования задают шаблоны для операций сложения и вычитания. Каждое из них состоит из двух меток <Exp> для подстановки выражений, арифметического знака, а также двух пробелов вокруг этого знака. Само правило явно задает способ, которым будут отформатированы арифметические выражения полученного языка. Вместо пробелов можно также указать табуляции и/или переносы строк. Недостатком данного подхода является то, что правила

форматирования задаются заранее, и, чтобы их изменить, необходимо менять описание синтаксиса языка. Кроме того, каждое такое правило задает единственный вариант форматирования данной структуры.

Наиболее близкий к нашей работе метод описан в [3]. Принтер языка может быть получен по ASF+SDF описанию языка [2], что представляет собой контекстно-свободную грамматику. При этом правила форматирования не указываются. Недостатком данного подхода является то, что при генерации принтера задается базовый стиль форматирования, и, чтобы его изменить, необходимо редактировать сгенерированный код, тогда как подход с использованием синтаксических шаблонов позволяет пользователю декларативным образом настраивать принтер.

### 2.1.2. Форматеры, встроенные в IDE

Так же широко распространены форматеры, встроенные в IDE. Они используют множество настроек для задания стиля форматирования (рис. 6). Среди них: тип и размер отступов, расположение фигурных скобок в С-подобных языках, политика переноса списочных выражений на новую строку и десятки других. Набор этих настроек выбирается разработчиком форматера на основе его представлений о возможных стилях кодирования для целевого языка. Такие форматеры обычно тесно интегрированы с IDE, имеют высокую скорость работы, и их выразительности, как правило, достаточно для задания необходимого стиля форматирования. Однако для поддержки нового языка необходимо определить нужный набор настроек и вручную реализовать форматер. В случае, если пользователю необходимо придерживаться стиля кодирования некоторой существующей кодовой базы, то ему нужно самостоятельно определить значения этих настроек. Данный недостаток отсутствует в работе [1], где авторы предлагают инструмент, позволяющий получить некоторые настройки форматера из существующего программного кода. Недостатком подхода является то, что число этих настроек невелико: система позволяет определять только стиль отступов, стили именования идентификаторов, необходимое количество комментариев. Другой подход [5] предлагает использование генетического

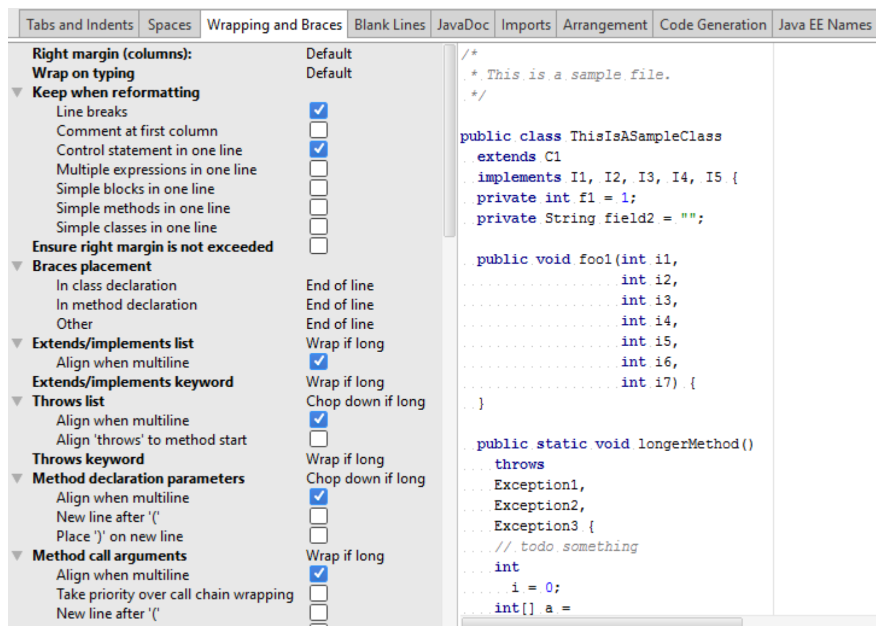


Рис. 6: Настройки форматера языка Java в IntelliJ IDEA

алгоритма для поиска настроек форматера в исходном коде программ на языке C.

## 2.2. Grammar-Kit

Как уже было отмечено, для поддержки нового языка в платформе IntelliJ IDEA необходимо разработать синтаксический анализатор. Для его генерации можно использовать плагин Grammar-Kit. В качестве системы описания синтаксиса языка используется БНФ-грамматика. Результатом работы плагина является код синтаксического анализатора и иерархия классов внутреннего представления.

Рассмотрим грамматику языка While, для которого производилась апробация метода, описанного в [7]. While — язык программирования, содержащий следующие конструкции: чтение из стандартного потока (**read**) и запись в стандартный поток (write), оператор ветвления (**if**), цикл с предусловием (while); процедуры (proc), бинарные выражения (binary_expr), в том числе булевы (binary_bexpr) и т. д. На рис. 7 представлена часть грамматики языка While, задающая множество операторов. Рассмотрим правило грамматики, задающее оператор ветвления. Правая часть правила состоит из терминалов '**if**', '**then**' '('

```
1  whileFile ::= proc_list stmt_list
2  stmt_list ::= stmt*
3  proc_list ::= procedure*
4  stmt ::= skip|assign|if|while|write|read
5  skip ::= 'skip' ';'
6  write ::= 'write' '(' expr ')' ';'
7  read ::= 'read' '(' id ')' ';'
8  assign ::= id ':=' expr ';'
9  if ::= 'if' '(' bexpr ')' 'then' stmt_list ('else' stmt_list)? 'fi'
10 while ::= 'while' '(' bexpr ')' 'do' stmt_list 'od'
11 procedure ::= 'proc' id '(' param_list ')' stmt_list 'endp'
12 param_list ::= ref_expr? (',' ref_expr)*
13 ...
```

Рис. 7: Грамматики языка While. Операторы языка

и др.; нетерминалов: bexpr, stmt_list, а также условного вхождения ('**else**' stmt_list)? (то есть конструкция может отсутствовать в программах на данном языке). Некоторые правила грамматики имеют модификаторы, которые используются для дополнительных указаний генератору синтаксического анализатора (рис. 8). Модификатор fake ука-

```
1  ...
2  fake ar_op ::= plus_op|mul_op
3  fake binary_expr ::= expr ar_op expr
4
5  expr ::= factor plus_expr *
6  left plus_expr ::= plus_op factor
7  plus_op ::= '+'|'−'
8  private factor ::= primary mul_expr *
9  left mul_expr ::= mul_op primary
10 mul_op ::= '*'|'/'|'%'
11 private primary ::= literal_expr | ref_expr | paren_expr
12 paren_expr ::= '(' expr ')'
13 ref_expr ::= id
14 literal_expr ::= number
15
16 fake bl_op ::= or|and
17 fake binary_bexpr ::= bexpr bl_op bexpr
18 ...
```

Рис. 8: Грамматика языка While. Выражения с модификаторами

зывает системе, что не нужно генерировать код синтаксического анализатора для обработки данной структуры, однако генерируется иерар-

хия классов внутреннего представления, private указывает, что не будет сгенерирована иерархия классов, left используется для поддержки левоассоциативности, а также некоторые другие[4]. Модификатор private используется для правил грамматики, которые не имеют представления в синтаксическом дереве. Среди них те, которые используются для устранения левой рекурсии. Например, на рис. 9 представлено описание правил с рис. 8 (строки 5–14), но в более естественной для человеческого восприятия леворекурсивной форме. Однако грамматика, с которой

```
1 expr ::= plus_expr | mul_expr | paren_expr | ref_expr | literal_expr
2 plus_expr ::= expr plus_op expr
3 plus_op ::= '+'|'−'
4 mul_expr ::= expr mul_op expr
5 mul_op ::= '*'|'/'|'%'
6 paren_expr ::= '(' expr ')'
7 ref_expr ::= id
8 literal_expr ::= number
```

Рис. 9: Грамматики языка While. Выражения в естественной леворекурсивной форме

работает Grammar-Kit, не должна содержать леворекурсивных правил. Устраняя левую рекурсию, мы получим описание выражений на рис. 8 (строки 5–14). Кроме того, появляются новые правила, которые с точки зрения синтаксического анализа (а следовательно, и форматирования) являются избыточными. В данном случае такими являются factor и primary на рис. 8. Каждый файл с грамматикой языка содержит в себе заголовок, в котором описывается различная дополнительная инфор-

[4]Посмотреть полный список можно по адресу https://github.com/JetBrains/Grammar-Kit

```
1 { parserClass="com.intellij.whileLang.parser.WhileParser"
2     psiClassPrefix="Psi"
3     psiImplClassSuffix="Impl"
4     psiPackage="com.intellij.whileLang.psi.impl"
5     tokens=[...]
6     ...
7 }
8 ...
```

Рис. 10: Заголовок файла с грамматикой

мация: используемые в сгенерированных файлах классы, префиксы и суффиксы сгенерированных классов внутреннего представления, Java-пакеты, множество терминальных символов грамматики (*tokens*) и др. (см рис. 10).

# 3. Метод генерации декларативных принтеров

В данном разделе описывается соответствие между XML-описаниями [7] и правилами грамматики, необходимые преобразования грамматики, а также процесс отбора правил, необходимых для получения принтера языка.

## 3.1. Определение поддеревьев и их свойств для правил грамматики

Текущий способ добавления новых языков в принтер-плагин предполагает создание описания всех значимых компонент языка. На рис. 11 представлено XML-описание структуры языка While. Количество таких структур в языке может быть большим, а значит, процесс создания такого описания может быть весьма трудоемким. Для каждой компоненты (*component*) принтера в XML-описании необходимо указать ее имя (*name*), класс внутреннего представления (*psiComponentClass*), а так же список поддеревьев (*subtree*) и их свойства. Данные свойства приведены ниже.

1. Имя поддерева (*name*).

2. Метод класса внутреннего представления, возвращающий данное поддерево или коллекцию поддеревьев такого типа (*psiGetMethod*).

```
<component name="While" psiComponentClass="PsiWhileStmt">
 <subtree name="condition" psiGetMethod="Bexpr"
    hasSeveralElements="false" isRequired="true" />
 <subtree name="body" psiGetMethod="StmtList"
    hasSeveralElements="false" isRequired="true" />
</component>
```

Рис. 11: XML-описание цикла с предусловием языка While

3. Является ли поддерево обязательным для синтаксической корректности структуры (*isRequired*).
4. Является ли поддерево набором однотипных элементов (*hasSeveralElements*).

Третье свойство с точки зрения принтера указывает на необходимость построения текстового представления для данного поддерева. Так, в языке Java блок **else** оператора ветвления необязателен, соответственно, для него может не существовать представления в синтаксическом дереве, значит, текстовое представление дерева оператора ветвления не будет содержать этот блок. В свою очередь, блок **then** является обязательным, и невозможность получения текстового представления для него приведет к невозможности получения текстового представления и для всего оператора ветвления. Четвертое свойство указывает, что поддерево является набором однотипных узлов синтаксического дерева. Например, тело функции состоит из нескольких операторов, и чтобы получить текстовое представление тела функции, необходимо получить текстовые представления для этих операторов и объединить их, расположив друг под другом. В случае поддерева body на рис. 11 hasSeveralElements="false", так как синтаксического дерево имеет отдельный узел для представления набора операторов (stmt_list), то есть body состоит из одного stmt_list.

Все указанные выше свойства можно получить прямо из грамматики языка. Название компоненты, класса синтаксического анализатора, метода для получение поддерева задаются с использованием логики генератора синтаксического анализатора. Поддеревьями будут являться нетерминалы в правой части. Поддерево будет обязательным, если оно не является элементом выбора ($A \to B \mid C \mid D$), элементом с условным вхождением (“?” — 0 или 1) или элементом с повторением (“*” — любое число раз). Если нетерминал в правой части допускает множественное вхождение (“+” — 1 или более раз, “*” — любое число раз), то считается, что поддерево состоит из нескольких элементов.

Ниже приведено правило грамматики языка While, задающее ту же

структуру языка, что и описание на рис. 11:

while_stmt ::= 'while' '(' bexpr ')' '**do**' stmt_list 'od'

Данное правило грамматики имеет два нетерминала в правой части bexpr и stmt_list. При этом конструкция в правой части не является выбором $(B \mid C \mid D)$, и каждый из нетерминалов не имеет условного или множественного вхождения. Таким образом, мы получаем те же данные, что указаны на рис. 11.

## 3.2. Необходимые преобразования правил грамматики

Грамматика языка содержит множество правил, однако некоторые из них могут не иметь представления в синтаксическом дереве или быть ненужными с точки зрения принтера. Генерация компонент принтера по таким правилам может привести к его некорректности или увеличению времени его работы. Чтобы полученный принтер был корректным и производительным, необходимо определить набор правил, задающих структуры языка, которые могут иметь различные текстовые представления, то есть будут форматироваться принтером. Такие правила будем называть *значимыми* (в контексте принтера). Остальные правила назовём *избыточными*, так как для них либо существует одно текстовое представление, либо не существует никакого. Поэтому генерация кода принтера для них бессмысленна, так как увеличивается время работы и генератора, и принтера.

### 3.2.1. Устранение левой рекурсии

Как уже было отмечено, грамматика, с которой работает Grammar-Kit, не может иметь леворекурсивных правил (непосредственная левая рекурсия: $A \to A\alpha$). Существует способ устранения левой рекурсии [8], однако при этом создаются вспомогательные правила, которые не имеют представления в синтаксическом дереве, а значит, не могут иметь

и текстового представления. Такие правила также считаются избыточными.

### 3.2.2. Другие правила грамматики

Грамматика языка может содержать особые правила, которые должны помечаться специальными модификаторами. Среди них: *private*, *external* — для правил с такими модификаторами не генерируются классы внутреннего представления. Так как компоненты принтера зависят от классов внутреннего представления и их методов, то генерация компонент по таким правилам не нужна.

## 3.3. Поиск правил, описывающих списочных структуры

Все структуры языка можно разделить на содержащие списочные поддеревья и не содержащие. Например, структура "вызов метода" имеет поддерево "список параметров". В статье [7] описываются особенности форматирования таких структур, поэтому их необходимо обрабатывать иным образом, то есть генерировать другой код. Для этого необходимо заранее знать, какая перед нами структура. Из грамматики языка мы не можем узнать, являются ли поддеревья данной структуры списками. Для решения этой проблемы в грамматику был введен новый модификатор для правил *list*. Предполагается, что пользователь системы найдет списочные правила в грамматике и отметит их вручную. Также списки в некоторых языках могут иметь отличные от запятой разделители: ";", "|" – и другие. Поэтому пользователю предлагается указывать еще и тип разделителя для списочной структуры, задаваемой данным правилом, если этот разделитель отличен от запятой. Для этого требуется указать значение параметра listSep:

```
list alt_list ::= expr ('|' expr)* { listSep='|' }
```

## 3.4. Итоги

Таким образом, было установлено соотношение между XML-описаниями, использовавшимися для задания компонент принтера, и правилами грамматики, что позволяет получить требуемую нам информацию. Из всего множества правил были выделены те, которые задают структуры, форматирование которых будет изменяться. По полученному множеству правил далее будут генерироваться компоненты принтера.

# 4. Реализация

В данном разделе описывается реализация генератора принтеров и изменения в архитектуре принтер-плагина, позволяющие интегрировать в него полученные генератором принтеры, а так же некоторые ограничения, накладываемые на полученный код.

## 4.1. Изменение архитектуры принтер-плагина

Изначально плагин создавался с целью форматирования только программ на языке Java, и не предполагалось никаких механизмов расширения числа поддерживаемых языков. Метод добавления поддерживаемых языков [7] по сути предполагал создание аналогичного плагина, но для программ на другом языке. То есть помимо принтера требовалось реализовывать интерфейс взаимодействия плагина с пользователем. Структура кода была монолитной, и поэтому большую его часть приходилось переносить в каждый новый плагин. Намного удобнее было бы свести все поддерживаемые языки в единую систему, чтобы пользователю не приходилось задумываться об установке отдельного плагина для каждого языка, а затем, при форматировании, выбирать соответствующий язык программирования. Для этого требовалось изменить архитектуру системы путем выделения языконезависимой части в отдельный модуль. При этом было необходимо минимизировать количество кода, который нужно будет генерировать (только принтер и компоненты). На рис. 12 представлена конечная архитектура принтер-плагина[5][6]. Для добавления поддержки нового языка в принтер-плагин требуется генерировать только языкозависимую часть (с помощью Grammar-Kit).

[5] https://bitbucket.org/igorozernykh/printerplugin
[6] https://github.com/IgorOzernykh/printer-core

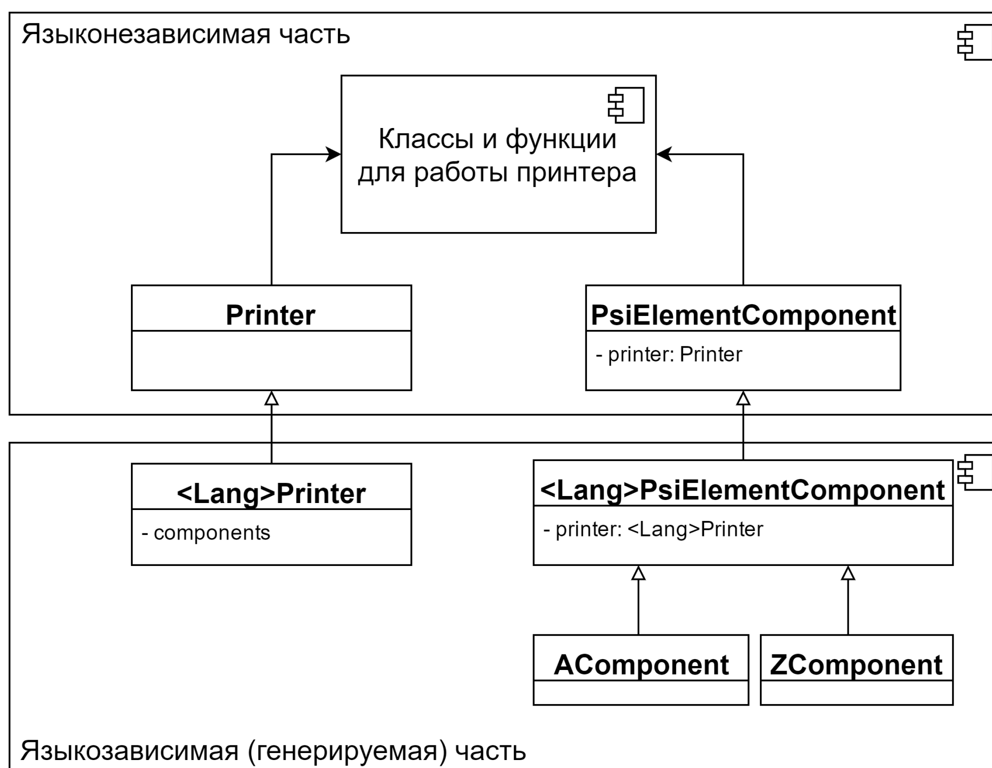


Рис. 12: Архитектура принтер-плагина

## 4.2. Генератор принтеров в плагине Grammar-Kit

После определения значимых правил, поддеревьев и их свойств происходит генерация кода компонент принтера. Генерация кода происходит с использованием текстовых файлов, которые имеют набор меток вида @TEXT@ для вставки соответствующей информации (рис. 13). Метки могут быть предназначены как для вставки названий классов, типов, так и для целых методов этих классов. Для каждого метода также имеется свой шаблон с набором меток. Таким образом, построение итогового кода осуществляется путём заполнения более мелких шаблонов нужной информацией и подстановкой их в более крупные.

Существует два основных шаблона: для обычных компонент и для списочных. Необходимость такого разграничения появилась из-за различного набор методов и их реализации. Упомянутый ранее модификатор *list*, применяемый к правилам грамматики, явно задает шаблон для генерации компонент. Аналогичным образом генерируется класс принтера.

```
<..>
class @NAME_CC@Component(printer: @LANG@Printer)
    : @LANG@PsiElementComponent<@COMP_CLASS@, <..>>
{
    @DECL_TAGS@
    @GEN_SUBTREES@
    @GET_NEW_ELEM@
    @UPDATE_SUBTREES@
    @PREPARE_SUBTREES@
    @GET_TAGS@
    @IS_TEMPL_SUIT@
    @GET_TEMPLATE@
}
```

Рис. 13: Шаблон для генерации компонент

## 4.3. Генерация файловой компоненты

Полученный в результате работы генератора принтер и его компоненты зависят от классов внутреннего представления программ. Так как классы внутреннего представления и соответствующие компоненты принтера генерируются по правилам грамматики, то мы можем гарантировать их корректное взаимодействие. Однако принтер вынужден также взаимодействовать и с классами, которые создаются вручную. Одним из них является класс внутреннего представления, описывающий структуру файла данного языка. Разработчик плагина для поддержки нового языка может задать данный класс разными способами, поэтому возникают трудности при генерации компоненты стандартным способом (разное число поддеревьев, которое можно получить при генерации и которое указано в синтаксическом узле; несоответствие имен методов, их типов). Решением этой проблемы стало явное задание поддеревьев, которые будут сгенерированы. Например, поддеревья для структуры, описывающей файла языка Java, можно задать следующим образом:

fileSubtrees=``PackageStatement, ImportList, Classes!*''.

Названия поддеревьев соответствуют методам узла синтаксического де-

рева, “*” указывает, что возвращаемых тип — коллекция (массив или список), а “!” — является ли поддерево обязательным (аналог аннотаций @NotNull и @Nullable).

## 4.4. Ограничения

Для преобразования программного текста в объекты, которыми оперирует система, используется *фабрика элементов*. Она, как и класс, описывающий файл языка, создается вручную. Причем заранее неизвестно, каким образом инстанциируются объекты данного класса, неизвестна сигнатура методов, поэтому, чтобы обеспечить взаимодействие принтера и фабрики элементов, необходимо вручную реализовать метод *createElementFromText* в классе принтера.

# 5. Апробация

Апробация производится для языков While и Erlang.

## 5.1. Принтер для языка While

Выбор данного языка был обусловлен тем, что количество структур в языке невелико и что для него уже существовал принтер, заданный с использованием метода [7] и позволяющий форматировать программы на данном языке. Грамматику потребовалось дополнить следующим образом. В заголовок необходимо было добавить имя пакета для генерируемого принтера (*printerPackage*), имя класса фабрики элементов (*factoryClass*), имя класса внутреннего представления для файла данного языка (*fileClass*). Эти значения необходимо указывать, чтобы верно определить зависимости генерируемых классов. Также в заголовке грамматики указывается список поддеревьев для файла данного языка (*fileSubtrees*), о чем было упомянуто в разделе 4. В данном случае, fileSubtrees="procList, stmtList". Далее необходимо было найти списочные правила и отметить их модификатором *list*. В языке While существует единственное списочное правило param_list. Таким образом, чтобы получить грамматику языка, потребовалось внести следующие изменения:

```
{
    <..>
    printerPackage="com.intellij.whileLang"
    factoryClass="com.intellij.whileLang.WhileElementFactory"
    fileClass="com.intellij.whileLang.WhileFile"
    fileSubtrees="procList, stmtList"
    <..>
}
<..>
list param_list ::= ref_expr? (COMMA ref_expr)*
<..>
```

Полученный в результате работы генератора принтер[7] корректно форматировал программы на данном языке. На рис. 14 представлена программа на языке While, которую требуется отформатировать. В качестве образцов форматирования будем использовать программы, представленные на рис. 15, *а* и 15, *б*. Результат работы принтера представлен на рис. 16, *а* и 16, *б* соответственно.

```
read(a); read(b);
res:=1;
while(b>0) do
  if (b%2=1) then
   res:=res*a; fi
  b:=b/2;
  a:=a*a;
od
write(res);
```

Рис. 14: Пример кода на языка While

```
read(a);
read(b);
while (a <> b)
do
    if (a > b)
      then a := a − b;
      else b := b − a; fi od
write(a);
```

а)

```
read(a);
read(b);
while (a <> b) do
    if (a > b) then
        a := a − b;
    else
        b := b − a;
    fi
od
write(a);
```

б)

Рис. 15: Образцы форматирования

[7] https://github.com/prettyPrinting/whileLang-idea-plugin

```
read(a);
read(b);
res:=1;
while (b > 0)
do
   if (b % 2 = 1)
      then res := res * a; fi
   b := b / 2;
   a := a * a; od
write(res);
```

а)

```
read(a);
read(b);
res := 1;
while (b > 0) do
      if (b % 2 = 1) then
            res := res * a;
      fi
      b := b / 2;
      a := a * a;
od
write(res);
```

б)

Рис. 16: Результаты форматирования программы из рис. 14

## 5.2. Принтер для языка Erlang

Язык Erlang был выбран исходя из того, что для этого языка существовала грамматика, по которой генерировался синтаксический анализатор и классы внутреннего представления с помощью плагина Grammar-Kit. Для генерации принтера по этой грамматики нужно было найти и отметить списочные правила, а также указать некоторую дополнительную информацию в заголовке файла, содержащего грамматику.

Однако полученный в результате реализованного генератора принтер некорректно форматировал некоторые структуры языка. Во-первых, возникла проблема с форматированием некоторых списочных структур. Связано это с тем, что в Erlang существуют списки, в которых одновременно могут быть различные типы разделителей. Рассмотрим конструкцию сопоставления с образцом (pattern matching) для списков: [X,Y,Z|ZS], где X, Y, Z – элементы списка, ZS – его “хвост”. X, Y, Z, ZS с точки зрения синтаксического дерева являются выражениями, “,” и “|” – разделителями. Существующий способ форматирования поддерживает лишь единственный тип разделителей, поэтому при форматировании таких списков все разделители будут заменены на явно заданный или установленный по умолчанию разделитель (в данном случае “|” будет

заменено на “,”), что приведет к некорректности конструкции.

Во-вторых, некоторые правила грамматики данного языка недостаточно выразительны, что не влияет на синтаксический разбор выражений языка, но негативно отражается на их форматировании. Например, бинарные операторы в грамматике Erlang задаются следующим образом:

private add_op ::= '+' |'−' | bor | bxor | bsl | bsr | **or** | xor
private mult_op ::= '/' |'*' | **div** 0| **rem** | band | **and**

По данным правилам не генерируются классы внутреннего представления, а значит, по бинарному выражению невозможно узнать, какой именно оператор в нем задан (так как не генерируются соответствующие get-методы), то есть бинарный оператор не является поддеревом бинарных выражений. Поэтому и принтер не будет строить представления для операторов, то есть выбирать тот, который соответствует данному узлу дерева и который требуется подставить в шаблон. Таким образом, оператор будет явно “зашит” в шаблон для данной структуры: @expression + @expression, где @expression – место для вставки подвыражений. Применяя такой шаблон к выражению с оператором “-”, получится выражение с оператором “+”, то есть семантика программы изменится, чего не должно происходить при переформатировании. Чтобы решить эту проблему в грамматике языка Erlang, требуется полностью изменить способ задания бинарных выражений, что изменит классы внутреннего представления, а значит, может повлиять на работу плагина языка.

Однако некоторые структуры языка форматируются полученным принтером корректно. На рис. 17 представлен образец кода на языке Erlang. Применяя шаблоны, полученные из этого образца к коду, представленному на рис. 18, *а*, мы получаем код на рис. 18, *б*.

```
foo(A, 0) −> A;
foo(A, B) −> foo(B, 0).
```

Рис. 17: Образец кода на языке Erlang

```
b_not(true)->false;
b_not(false)->true.
b_and(true,true)->true;
b_and(_,_)->false.
b_or(false,false)->false;
b_or(_,_)->true.
```

а) Неотформатированный код

```
b_not(true) -> false;
b_not(false) -> true.
b_and(true, true) -> true;
b_and(_, _) -> false.
b_or(false, false) -> false;
b_or(_, _) -> true.
```

б) Код с заданным форматированием

Рис. 18: Пример форматирования программы на языке Erlang

## 5.3. Итоги

Апробация для языка Erlang показала, что не любая грамматика позволит получить принтер для языка, задаваемого этой грамматикой. Во-первых, для структур языка, которые могут иметь различные текстовые представления, правила должны быть заданы таким образом, чтобы по ним генерировались классы внутреннего представления. Например, чтобы избежать проблем с описанными выше бинарными выражениями, требуется, чтобы отдельно было задано правило для бинарных операций, что позволило бы рассматривать их как поддерево бинарных выражений, а не как терминальные символы. Во-вторых, поддеревья структуры не должны пересекаться или быть вложенными друг в друга. По умолчанию это условие выполняется, однако пользователь может также явно задавать поддеревья[8]. В случае пересечения поддеревьев будет невозможно построить для них текстовые представления.

[8]https://github.com/JetBrains/Grammar-Kit/blob/master/HOWTO.md#32-organize-psi-using-fake-rules-and-user-methods

# Заключение

В рамках этой работы были достигнуты следующие результаты:

- Предложен метод генерации декларативных принтеров по грамматике в форме Бэкуса-Наура.
- Реализован генератор принтеров в рамках проекта Grammar-Kit[9].
- Добавлена возможность интеграции сгенерированных принтеров с принтер-плагином для IDE IntelliJ IDEA.
- Получен принтер для языка While, позволяющий форматировать программы на этом языке.
- Установлено, что не по любой грамматике языка можно получить корректный принтер. Представлены требования к грамматике.

[9]https://github.com/IgorOzernykh/Grammar-Kit

# Список литературы

[1] F. Corbo, C. Del Grosso, M. Di Penta. Smart Formatter: Learning Coding Style from Existing Source Code. — ICSM, 2007.

[2] Klint P. A Meta-environment for Generating Programming Environments. — ACM Trans. Softw. Eng. Meth. 2, 2, p176-201, 1993.

[3] M.G.J. van den Brand, Visser E. Generation of Formatters for Context-free Languages. — ACM Trans. Softw. Eng. Meth. 5, 1, p1-41, 1996.

[4] Nielson F., Nielson H.R., Hankin C. Principles of Program Analysis. — Springer-Verlag New York Inc., 1999.

[5] Utkin A., Shein R., Kazakova A. Applying Genetic Algorithms to Automatic Code Formatting. — URL: https://blog.jetbrains.com/clion/2015/11/applying-genetic-algorithms-to-automatic-code-formatting/ (online; accessed: 09.05.2016).

[6] Vollebregt T., Lennart C.L. Kats, Visser E. Declarative Specification of Template-Based Textual Editors. — Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications. Article No. 8, 2012.

[7] Подкопаев А.В., Коровянский А.Ю., Озерных И.С. Языконезависимое форматирование текстов программ на основе сопоставления с образцом и синтаксических шаблонов. — Научно-технические ведомости СПбГПУ 4' (224), 2015.

[8] Хопкрофт Д., Мотвани Р., Ульман Д. Введение в теорию автоматов, языков и вычислений, 2-е изд. : Пер. с англ. — Москва, Издательский дом «Вильямс», 2002.