Санкт-Петербургский Государственный
Университет
Математико-механический факультет

Кафедра системного программирования

Применение неполных грамматик для вычисления метрик программного кода

Дипломная работа студента 461 группы
Королева Дмитрия Николаевича

Научный руководитель	..... /подпись/	ст. преп. Я.А. Кириленко
“Допустить к защите” заведующий кафедрой	..... /подпись/	д.ф.-м.н., проф. А.Н. Терехов

Санкт-Петербург
2008

Содержание

1 Введение

В сфере информационных технологий нередко складывается ситуация, когда продолжать поддержку существующей программы становится слишком дорого, а полностью отказаться от неё невозможно. На этом этапе возникает задача реинжиниринга. Реинжиниринг – это деятельность по модернизации ранее реализованных технических решений на действующем объекте [7].

Довольно часто программы являются основным, а иногда даже единственным хранилищем информации о процессах организации, но вносить в них изменения и исправлять ошибки становится со временем все сложнее. Такие программы могут использоваться десятилетиями, за это время не только их разработчики, но и специалисты, знакомые с использованными, при создании этих программ, технологиями, становятся недоступны.

Процессы реинжиниринга можно разделить на несколько категорий [15]: извлечение бизнесс логики (Business Rule Extraction for Software Re-engineering); расширение типа данных (традиционный пример — Y2K Data Expansion); перестройка кода (Interactive and off-line Code Restructuring); подстройка под диалект языка, API или идиомы (Conversion Regarding Language Dialect, API, Idioms); передокументирование программных продуктов (Software Re-documentation); метрическая оценка качества программных продуктов (Metrics-based Software Assessment).

В данной работе рассматривается процесс реинжиниринга на начальном этапе, а точнее на этапе инвентаризации программного продукта (подсчет некоторых количественных характеристик). Инвентаризация (от позднелат. Ivetarium) — опись имущества [7]. Так, например, сложность программы можно оценить метрикой Холстеда [18], вычисляемой на основе анализа числа строк и синтаксических элементов исходного кода программы, или же метрикой Чепина, основанной на оценке информационной прочности отдельно взятого программного модуля, а так же показателем цикломатической сложности Маккейба [8]. Эти метрики принято называть традиционными [17], так как они основаны на императивном подходе к языку, они применяются к измерению сложности структурных систем с 1976 [8]. В современных условиях большинство программных проектов создается на основе объектно-ориентированного подхода, в связи с чем существует значительное количество метрик, позволяющих получить оценку сложности объектно-ориентированных проектов, например метрики для подсчета глубины вложенности классов и связанности между классами объектов. Велось множество споров о возможности применения традиционных метрик к объектно-ориентированному коду, но в статье [17] показывается, что применение этих метрик позволяет вполне корректно оценить сложность приложений.

Традиционно для целей оценки исходного кода ПО используют синтаксические анализаторы, построенные по полной грамматике конкретных языков [26]. Так, например, существует множество реализаций генераторов синтаксических анализаторов на основе контекстно-свободных грамматик. Но, к сожалению, не для всех языков имеется возможность построить такую грамматику. В частности, устаревшие языки (например, COBOL, PL) разрабатывались параллельно с развитием теории трансляции языков, в то время, когда многие аспекты теории были еще не изучены, и они не соответствуют многим стандартам, принятым позже. Следовательно, в таких случаях приходится отказаться от возможности использования подобных инструментов.

Другая проблема построения анализаторов по полной грамматике возникает при столкновении с различными диалектами одного и того же языка, каждый из которых имеет свою специфику (дополнительные ключевые слова и т.п.) — приходится переписывать сконструированный анализатор, подгоняя его под каждый отдельный диалект. Например, существуют десятки различных диалектов Cobol'a, многие из которых имеют взаимно противоречащие особенности, поэтому не представляется возможным создать «универсальный» синтаксический анализатор Cobol'a.

Однако, стоит все же заметить, что для анализа многих метрик требование знания полной грамматики или наличия корректного ввода не обязательно, и зачастую даже излишне. Например, для подсчета максимальной глубины вложенности классов не обязательно знать строение while-цикла. Возникает желание иметь возможность строить анализатор для подсчета метрик, основываясь лишь на некоторой части нужной грамматики.

2 Постановка задачи

Искомое решение описанной проблемы должно основываться на предположении о возможной избыточности грамматики в условиях вычисления конкретной метрики. Предлагаемое решение позволит, в определенных случаях, строить анализатор исходного кода ПО по неполной (незавершенной) грамматике.

Неполная грамматика (в контексте нашей работы) — грамматика, в которой существуют нетерминалы, не имеющие правил вывода..

Стоит отметить, что искомое решение должно обладать следующими важными свойствами. Во-первых, оно должно сохранить неизменной результат вычисления метрики анализатором, построенным по полной грамматике входного языка. Во-вторых, результат анализатора, построенного по любой неполной грамматике, которая будет удовлетворять поставленным условиям, должен совпадать с результатом анализа на основе некоторой полной грамматике. Кроме того, желательно, чтобы построение такого анализатора для соответствующих метрик выполнялось автоматически.

Структура дипломной работы. В главе 3 дан обзор имеющихся методов анализа кода по неполной грамматике языка. В главе 4 приведен обзор наиболее часто встречающихся метрик анализа исходного кода программных продуктов. В главе 5 приведено описание реализуемого инструмента на основе генератора анализаторов ANTLR. В главе 6 рассматривается реализация инструмента. В приложении А показан пример реализации нечеткого разбора в нотации ANTLR, в приложении В – пример использования островных грамматик в нотации ANTLR. Результаты работы сформулированы в части 7.

3 Методы анализа

Одна из целей данной работы — показать возможность подсчета более-менее сложных метрик, требующих дерево синтаксического разбора, поэтому мы изначально отказываемся от применения исключительно лексического подхода для их вычисления.

Для реализации синтаксического анализа, основанного на неполных грамматиках, существует несколько подходов:

- нечеткий разбор (fuzzy parsing) [12];
- островные грамматики (island grammars) [19];
- скелетные грамматики (skeleton grammars), их надстройка – толерантные (tolerant grammars) [15];
- устранение ошибок (error repair) [20].

Каждый из этих подходов требует свой уровень детализации грамматик, представленный на рисунке №1.

Рисунок №1. Спектр подходов к построению анализаторов.

Все эти подходы основаны на дифференциации исходного кода на значимые и незначимые конструкции. Рассмотрим далее каждый из подходов более подробно.

3.1 Нечеткий разбор

Нечеткий разбор представляет собой синтаксический анализ над выбранными порциями ввода, с целью получения частичной модели исходного кода. Ключевая идея – идентифицировать «опорные терминалы» (anchor terminals), которые запускают применение контекстно-свободных правил вывода. Таким образом, ввод пропускается до тех пор, пока не найден опорный терминал b, затем контекстно-свободный анализатор пытается применить правило, начинающееся с b.

Пусть G = (V_N, V_T, R, S) – контекстно-свободная грамматика, V_N – множество нетерминалов, V_T – множество терминалов, R – множество правил, S – начальный нетерминал. Тогда язык, сгенерированный из G, L(G), определен как множество строк s $s \in V_T^*$, где s получены из S, путем применения правил из R. В нашем контексте можно сказать, что s соответствует программе на языке L(G).

Нечеткий парсер P(G) для данного языка L(G) – синтаксический анализатор, распознающий некоторые подстроки из множества строк L(G). Формально его можно описать четверкой: (V_N' , V_T' , R' , A), где A называют опорным символом (anchor symbol), $A \subseteq V_T'$, $V_T' \subseteq V_T$. Опорный символ помечает начало подстроки, распознаваемой F(G). Таким образом, каждая строка $s \in L(G)$, которая содержит хотя бы один якорь, частично принимается нечетким парсером. Для каждого опорного символа a существует правило

$r_a \in R'$, левая часть которого состоит из одного нетерминала $n_a \in V_N'$, называемого опорным нетерминалом, а правая определяет продукцию, совпадающую с подстрокой в s, начинающуюся с a. Получаем, что все опорные нетерминалы в V_N' вкупе с соответственными правилами из R' определяют набор «подграмматик» исходной грамматики G.

Язык $L_F(G)$, частично принимаемый F(G), формально может быть описан так:

$L_F(G) = \{ s \in L\ G \mid s = w1aw2 \ \& \ w1 \in V_T^* \ \& \ w2 \in V_T^* \ \& \ a \in A \ \& \ S \rightarrow^* s$ с применением правил из R $\}$

Иначе говоря, F(G) распознает лишь части языка L(G), посредством неструктурированного множества правил. В сравнении с парсерами, покрывающими весь язык L(G), который рассматривает каждый терминальный символ при условии корректности ввода, нечеткий парсер продолжает бездействовать до тех пор, пока его сканер не встретит опорный символ. Вслед за этим парсер пытается распознать n_a, применив правило r_a. Затем парсер снова ничего не делает, пока не встретит другой опорный символ или не достигнет конца ввода.

Для практических целей, если порядок анализа определен в форме рекурсивного спуска, строгое применение правила r_a, для распознанной подстроки, начинающейся с a, не обязательно. Здесь программист может уходить от строгого применения грамматических правил и рассматривать частные оптимизации. Например: пропустить множество токенов или применить стратегию обработки ошибок для парсеров, анализирующих незавершенные программные системы.

Нечетки разбор был применен в коммерческой среде разработки C++, Sniff [24]. Он умеет обрабатывать незавершенные программные системы, содержащие ошибки, путем извлечения описания структур, таких как классы, методы, функции и описания переменных, игнорируя тела функций и методов.

Также этот вид разбора используется в cscope [4], source navigator [1] и некоторых других продуктах.

3.2 Островные грамматики

Подход, связанный с использованием островных грамматик является надстройкой над техникой нечеткого разбора.

Островная грамматика [19] — это грамматика, состоящая из детализированных продукций, описывающих определенные конструкции интереса (острова), и свободных продукций, соответствующие остальному.

Формально это определение будет выглядить следующим образом образом.

Определение. Пусть дан язык L_0 и контекстно-свободная грамматика $G = (V_N, V_T, P, S)$, такие что $L(G)=L_0$. Множество конструкций интереса $I \subset V_T^{¿}$, $\forall i \in I : \exists s_1, s_2 : s_1 i s_2 \in L(G)$. Островная грамматика $G_I = (V_{NI}, V_{TI}, P_I, S_I)$ для языка L_0 обладает следующими свойствами:

1. $L(G) \subset L(G_I)$ - G_I порождает расширение языка L(G);
2. $\forall i \in I : \exists v \in V_{¿} : v \rightarrow^{¿}, \exists s_3, s_4 : s_3 i s_4 \notin L(G) \wedge s_3 i s_4 \in L(G_I)$.

Данный подход не требуют полного дерева разбора и может быть применен для локальных трансформаций, таких как: простые структурные модификации, нормализации условий, изменения каких-либо стандартов кодирования.

Островные грамматики не требуют использования какого-то специального формализма описания грамматик или техник разбора.

Рассмотрим пример островной грамматики, основанный на LR-разборе и SDF

формализме [25].

```
Module Layout
lexical syntax
[\ \t\n] -> LAYOUT

module Water
imports Layout
context-free syntax
Chunk* -> Input
Water -> Chunk
lexical syntax
~[\ \t\n]+ -> Water {avoid}

module DataParts
lexical syntax
[0][1-9] -> Level
[1-4][0-9] -> Level
[A-Z][A-Z0-9\-]* -> DataName

module DataFields
imports Water DataParts
context-free syntax
Level DataName -> Chunk
```

Листинг №1.

Данная грамматика определяет структуру объявлений данных в программе Cobol'a. Парсер, полученный из этой грамматики может выделять все объявления в данной ему строке на Cobol'е. Например, 3 объявления данных "01 REC1", "03 FLD1", и "03 FLD2" будут распознаны для следующей программы:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. TST.
DATA DIVISION.
WORKING-STORAGE
01 REC1.
03 FLD1 PIC 99.
03 FLD2 PIC S9(4) USAGE COMP.
PROCEDURE DIVISION.
...
```

Листинг №2.

Продукции в модуле DataParts определяют лексические структуры Level и DataName. В показанном фрагменте 2 Level-числа: 01 и 03; 3 имени данных: REC1, FLD1, и FLD2. Продукция

```
LevelDataName -> Chunk
```
Листинг №3.

в модуле DataFields определяет контекстно-свободный синтаксис конструкций интереса (острова), объявления данных. Оставшаяся продукция в модуле Layout и Water нужны для пропуска разметки и ненужной части ввода (вода). В Частности, продукция

```
~[\ \t\n]+ -> Water {avoid}
```
Листинг №4.

определяет каждый токен без пробелов, табуляций, перевода строки и может быть разобран как «вода», в то время как avoid означает слабый приоритет продукции. Таким образом, «острова» Level и DataName строго предпочтительней «воды». Данная грамматика очень сжата, так как она описывает лишь структуру конструкций интереса.

Используя данный подход, хочется быть уверенным, что в случае описаеия полной исходной грамматики для извлечения конструкций интереса, мы получим такой же результат, что и в случае использования островной грамматики.

Далее, в примере, под исходной грамматикой полагается грамматика VS Cobol II. Исходная продукция объявления данных выглядит следующим образом:

```
Level-number
data-name-or-filler?
Data-description-entry-clauses
"."
            -> Data-description-entry
```
Листинг №5.

Можно заметить, что «островная» продукция для DataField в действительности покрывает лишь некоторую часть исходной. В частности, точка прерывания «.» пропущена в описании «островной» продукции. Также, конструкция Data-description-entry (в исходной) заменяется на «острова» DataField, для простоты представления. Возникает вопрос: позволяет ли островная грамматика допускать ошибочно-позитивные и ошибочно-негативные результаты, при попытке распознать конструкции интереса?

Ошибочно-позитивный результат – подстрока разбирается островной грамматикой как остров, но не соответствует конструкции интереса. Пример ошибочно-позитивного фрагмента:

```
01 FLD1 PIC 99.
01 FLD2 PIC 99 VALUE 42 COMP-3.
```
Листинг №6.

В самом деле, «42 COMP-3» соответствует островной продукции

```
Level DataName ->Chunk
```
Листинг №7.

но не является Data-description-entry в исходной. Это может быть исправлено путем отбрасывания «COMP-3» как имени данных. В SDF это может быть сделано следующим образом:

```
"COMP-3" -> DataName {reject}
```

Листинг №8.

В конечном счете, все, или почти все, зарезервированные слова в Cobol'е должны быть отброшены таким образом. Это место является склонным к ошибкам, если обработано вручную и без обращения к исходной грамматике. Стоит отметить, что бывают и другие случаи возникновения ошибочно-позитивных результатов.

Ошибочно-отрицательный результат – подстрока, являющаяся надлежащей конструкцией интереса в исходной грамматике, пропускается островной грамматикой как «вода».

Давайте предположим, что конструкции интересов – описания данных (не просто имена). Тогда следующий фрагмент содержит 2 ошибочно-отрицательных результата:

```
01 REC1.
03 PIC X.
03 FILLER PIC X.
```

Листинг №9.

Подстрока «03 PIC X.» сопоставляется с правилом исходной грамматики Data-description-entry, как вход без имени данных. Тогда как островная грамматика не распознает данную специальную форму. Также, «03 FILLER PIC X.» - не распознается (здесь предполагается, что слово FILLER было зарезервировано как имя данных для избегания ошибочно-положительных результатов).

Позволять островной грамматике принимать ошибочно-положительные или ошибочно-отрицательные результаты – не лучшая тактика. Чтобы запретить такое поведение, каждый раз когда обнаружена ошибка данного плана необходимо накладывать дополнительные ограничения на островную грамматику, путем добавления правил, приоритетов и другого рода ограничений. Например, мы можем переделать островную грамматику для определения данных так, что она станет более уступчивой по отношению к исходной:

```
Data-description-island -> Chunk
Level (DataName|"FILLER")? Water* "."
        -> Data-description-island
```

Листинг №10.

Нетерминал Data-description-island может быть напрямую отображен на нетерминал Data-description-entry исходной. Чтобы усилить правильность контекста для ввода определеных данных, требуется больше использовать совместные структуры.

Расмотрим алгоритм построения анализатора на основе островных грамматик на примере MANGROVE.

MANGROVE – генератор source model extractors, основанных на островных

грамматиках. Для представления грамматик был выбран SDF формализм описания синтаксиса, поддерживаемый обобщенным LR-разбором (рисунок №2).

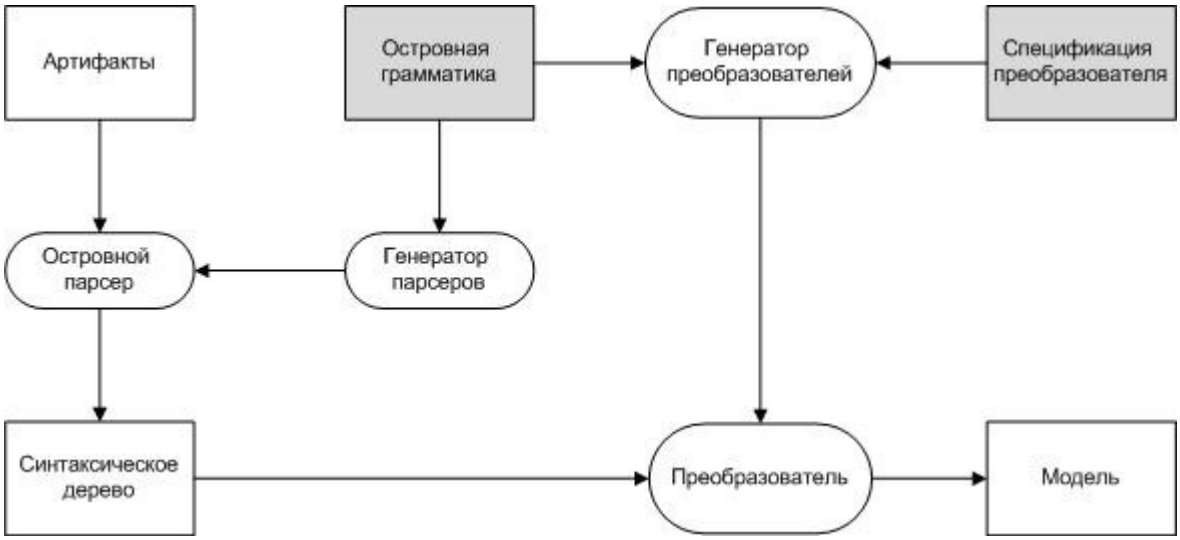


Рисунок №2. Архитектура MANGROVE.

Схема получения анализатора основана на двух типах входных данных (серые прямоугольники). Первый определяет островную грамматику, описывающую синтаксис распознаваемых конструкций (используется для генерации островного парсера). Второй определяет связь этих конструкций с желаемой моделью. Вместе они используются для генерации преобразователя (extractor), который читает выход островного парсера и конвертирует его в модель, по которой уже строится желаемый анализатор.

С помощью данного генератора легко строится анализатор, для подсчета цикломатической сложности, что было показано в работе [19].

3.3 Скелетные грамматики

Подход разбора исходного кода на основе скелетной грамматики основывается на той же идеи определения конструкций интереса, что и предыдущий (с островными грамматиками), но требует более подробного описания грамматики входного языка.

Весь процесс получения полной грамматики (скелетной) показан на рисунке №3.

Рисунок №3. Получение скелетной грамматики из исходной.

Он состоит из четырех этапов.

Первый — выбор правил вывода для описания конструкций интереса. В дальнейшем, правила вывода будем называть продукциями, а правила вывода конструкций интереса — контрактными продукциями.

Второй — дополнение контрактных продукций исходными для достижения конструкций интереса из начального нетерминала. Этот шаг называется «дополнение

корнем» (root completion).

Третий — все неопределенные или частично определенные нетерминалы дополняются «свободными» конструкциями по-умолчанию. Этот шаг называется «дополнение по умолчанию» (default completion).

Четвертый — толерантность грамматики может быть ослаблена или усилена, благодаря введению дополнительных ограничений. Этот шаг называется «ограничения по умолчанию» (default restriction).

Опишем процесс получения скелетной грамматики из исходной на примере (в данном примере подразумевается, что у нас есть полная исходная грамматика языка).

1. Контрактные продукции

Типичной операцией реинжиниринговых пакетов для языка Cobol'a [23] является исключение jump-предложения NEXT SENTENCE.

```
IF A>B THEN
MOVE 1 TO B
ELSE
NEXT SENTENCE
DISPLAY "WILL NEVER BE REACHED".
DISPLAY "START OF NEXT SENTENCE".
```

Листинг №11.

В приведенном примере (листинг №11), конструкция NEXT SENTENCE может быть переведена в CONTINUE с удалением мертвого кода. После чего получим:

```
IF A>B THEN
MOVE 1 TO B
ELSE
CONTINUE.
DISPLAY "START OF NEXT SENTENCE".
```

Листинг №12.

Для проведения данной трансформации нам необходимо в исходном коде отлавливать следующую конструкцию интересов:

```
context-free syntax
"CONTINUE" -> Statement
"NEXT" "SENTENCE" -> Statement
Statement* -> Statement-list
variables
"Stat*" [0-9]* -> Statement
```

Листинг №13.

Для демонстрации результата, в следующем листинге приведена скелетная грамматика для нашего примера.

```
contract.1 "CONTINUE" -> Statement
contract.2 "NEXT" "SENTENCE" -> Statement
contract.3 Statement* -> Statement-list
```

```
root.1 Statement-list "." -> Sentence
root.2 Label-name "." Sentence* -> Paragraph
root.3 Paragraph-without-header? Paragraph* -> Paragraphs
root.4 Section-header "." Paragraphs -> Section
root.5 Section-without-header Section* -> Sections
root.6 Proc-division-header Sections -> Proc-division
root.7 Id-division? Env-division? Data-division? Proc-division? -> Program
```

```
default.1 Token-start-verb Token-stat* "." -> Statement
default.2 Integer | User-defined-word -> Label-name
default.3 Sentence+ -> Paragraph-without-header
default.4 Label-name "SECTION" Integer? -> Section-header
default.5 Paragraphs -> Section-without-header
default.6a "PROCEDURE" "DIVISION" Using-phrase "." Declaratives? -> Procedure-division-
header
default.6b "USING" User-defined-word* -> Using-phrase
default.6c "DECLARATIVES" "." Token-excl-end* "END" "DECLARATIVES" "." -> Declaratives
default.7 "IDENTIFICATION" "DIVISION" "." Token-excl-env-data-procedure* -> Id-division
default.8 "ENVIRONMENT" "DIVISION" "." Token-excl-data-proc* -> Env-division
default.9 "DATA" "DIVISION" "." Token-excl-proc* -> Data-division
```

Листинг №14. Продукции полученной грамматики. Вспомогательные тоукены продукций опущены.

Продукции распределены по группам: contract — контрактные продукции, root — полученные на шаге «дополнение корнем», default — на шаге «дополнение по-умолчанию».

2. Дополнение корнем

Следующим шагом является добавление правил, с помощью которых контрактные продукции выводятся из начального нетерминала, что обеспечивает рассматривание конструкций интереса только в нужных частях программы, то есть позволяет избегать ошибочно-положительных результатов. В нашем примере мы должны соединить Statement и Program, что делается с помощью тривиального алгоритма: из исходной грамматики транзитивно выбираются все продукции, необходимые для достижения начального нетерминала.

```
• Обозначения:
D(P) – нетерминалы, определенные в продукциях P.
U(P) – нетерминалы, использующиеся в продукциях P.
N(P) — объединение U(P) и N(P).
```

```
• Вход:
– Исходная грамматика G_B = (N_B, T, P_B, s).
– Контрактные продукции PC подмножество P_B.
```

• Выход:
– Грамматика от корня $G_R = (N_R, T, P_R, s)$.

• Алгоритм:
1. $P_R := P_C$ (инициализация).
2. Повторяем шаги (a) и (b) пока возможно.
   (a) Выбираем продукцию p принадлежащую $P_B \setminus PR$,
   где p имеет вид $n \rightarrow u\ n'\ v$, n' принадлежит $D(P_R)$ и n не принадлежит $D(P_R)$.
   (b) $P_R := P_R$ объединить {p}.
3. $N_R := N(P_R)$

Листинг №15.

3. Дополнение по умолчанию

Некоторые нетерминалы в получившейся грамматике могут оказаться полностью, или частично не определенными. И контрактные продукции, и продукции включенные на шаге «дополнение корнем» могут содержать подобного рода нетерминалы. Следующий шаг нужен для назначения таким нетерминалам «заглушек».

В нашем примере, в продукциях по-умолчанию (заглушек) (листинг №14), используются нетерминалы, начинающиеся с Token-... . Их определения получаются автоматически, путем перечисления всех токенов, которые могут появиться в данном контексте. Таким образом, эти нетерминалы определяют корень «водной» части дерева разбора. Благодаря продукциям по умолчанию, полученная грамматика способна распознать следующий фрагмент:

```
ACCEPT CURRTIME FROM DATE
CONTINUE
ADD A TO B
```

Листинг №15.

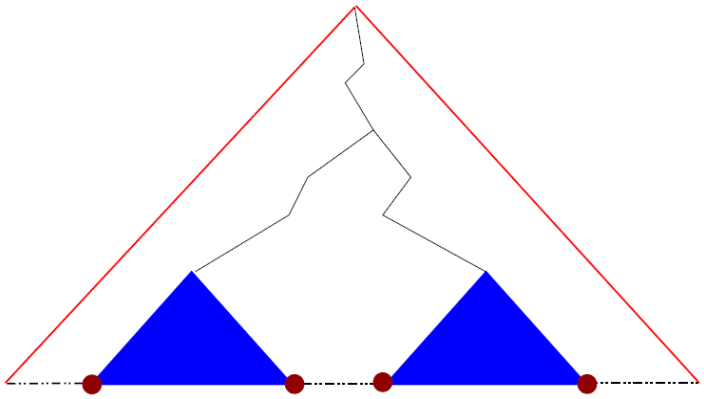
Здесь содержится 3 предложения: первое и последнее разбираются с помощью продукции по-умолчанию для предложений, в то время как CONTINUE, в середине, разбирается контрактной продукцией. Полагается, что продукции по умолчанию имеют меньший приоритет, чем остальные.

От того, каким образом будут получаться продукции по-умолчанию, будет зависеть устойчивость нашего разбора.

Наивный подход заключается в определении продукций по умолчанию следующим образом: Token*. Для нашего примера:

```
default.1 Token* -> Statement
...
default.9 Token* -> Data-division
```

Листинг №16.



```
"DATA" "DIVISION" "."
Token-excl-proc*
-> Data-division
```

Листинг №17.

Предполагается, что нетерминал Token-excl-proc определен для всех возможных токенов (включая «PROCEDURE»). Продукция выражает, что содержимое data division обязательно начинается с «DATA» «DIVISION» «.». Это довольно свободное определение, но оно является безопасным, так как находится в соответствии с исходной грамматикой. То есть, любой может убедиться, что последовательность токенов «DATA» «DIVISION» «.» больше нигде не появляется, кроме как в начале data division. Другой пример — продукция Statement:

```
Token-start-verb Token-stat* -> Statement
```

Листинг №18.

Здесь, Token-start-verb содержит все начальные операторы предложений Cobol'a, такие как «ACCEPT», «ADD» и другие. Token-stat опеределяет все токены, которые могут появиться в предложении Cobol'a. Из него булут исключены начальные символы и «.».

Иформация о синхронизации, которая используется в примере, может быть получена из исходной грамматики путем системного анализа. С этой целью для нетерминала должны быть подсчитаны несколько множеств токенов из теории разбора ([5], [6]):

- firstG(n) — первые токены в строках, получаемых из n;
- lastG(n) — последние токены в строках, получаемых из n;
- followG(n) — токены, следующие за n в сентенциальной форме;
- precedeG(n) — токены, предворяющие n в сентенциальной форме.
- prefixG(n) — токены, где-либо встречающиеся в префиксе n;
- postfixG(n) — токены, где-либо встречающиеся в постфиксе n;
- bodyG(n) — токены, где-либо встречающиеся в строке, получаемой из n, игнорируя firstG(n) и lastG(n).

Используя эти множества можно определить безопасные продукции по умолчанию. Например, для нетерминала n, мы можем определить его продукцию по-умолчанию как «f t* l», при следующих условиях:

- из n что-то выводится в G (непустое порождение);
- firstG(n) = {f} (определенное начало);
- lastG(n) = {l} (определенный конец);
- f не принадлежит prefixG(n) (уникальное начало);
- l не принадлежит postfixG(n) (уникальный конец);
- l и f не принадлежит bodyG(n) (начало и конец не содержатся в теле);
- t = bodyG(n) (все возможные токены).

Используя приведенные критерии, продукции по умолчанию могут предоставляться систематично (иногда приходится принимать во внимание более одного токена).

4. Ограничения по-умолчанию

На данной стадии мы уже получили полную устойчивую грамматику. Остается один нерешенный вопрос: возможность ошибочно-отрицательных результатов (с ошибочно-положительными мы боремся на этапе «дополнение корнем»). Здесь предлагается ввести дополнительные ограничения на продукции по умолчанию. Например исключить из них токены, задействованные в конструкциях интереса.

Данный подход был автоматизирован и применялся при реструктурировании кода программ на COBOL'e [15].

3.4 Устранение ошибок

Подход основывается на задаче обработки некорректных цепочек, которые решают с помощью алгоритмов реакции на ошибку.

Различают 3 основных способа [3]:

1. Обнаружение ошибок (error detection)
2. Восстановление после ошибок (error recovery)
3. Исправление ошибок (error correction)

Наименьшее, что можно потребовать от транслятора, это показать наличие синтаксической ошибки во входной строке (то есть, что строка не принадлежит языку, описанному данной грамматикой). Это называется обнаружением ошибок.

Для некоторых приложений этого может быть достаточно. Но, как правило, необходимо узнать о всех ошибках во входной строке. В этом случае транслятор должен суметь продолжить работу, изменив свое внутреннее состояние. Такая обработка называется восстановлением после ошибок. В общем случае, компилятор указывает позицию, в которой обнаружена ошибка. Также вывод может включать некоторое диагностическое пояснение, например, «в данной позиции ожидается ';'».

Главный недостаток анализатора с восстановлением после ошибок – невозможность построить дерево синтаксического разбора при наличии ошибок во входной строке. При обнаружении ошибок изменение внутреннего состояния может привести к выполнению семантических действий, связанных с правилом грамматики, в том порядке, который был бы невозможен при синтаксически корректном входе. Это может привести к появлению наведенных ошибок.

Одним из вариантов решения этой проблемы является игнорирование семантических действий в том случае, если была обнаружена ошибка. Но это не всегда приемлемо. Более подходящим вариантом является использование различных методов исправления ошибок ([14], [22]). При исправлении ошибок анализатор преобразует входную строку в

синтаксически корректную, удаляя, добавляя или изменяя терминальные символы. Стоит заметить, что такое преобразование не всегда приводит входную строчку к той, которая подразумевалась пользователем. Поэтому к исправлению, как правило, допускаются только простые ошибки [11]: например, отсутствие точки с запятой.

Этот подход подробно разбирался в дипломной работе Ефимова Андрея Александровича на тему «Построение ослабленного LALR-транслятора на основе анализа грамматики на избыточность» [3], поэтому подробнее этот подход мы рассматривать не будем.

Решать поставленную задачу мы решили с использованием скелетных грамматик. На вход нашему инструменту ожидается грамматика от корня.

4 Метрики

Метрики являются атрибутами программных продуктов, которые могут описать множество вещей, включая сложность, усилия, качество, надежность и многое другое. В наше время их существует большое разнообразие, однако в статье Тима Мензиса [21] показывается, что для каждого отдельного проекта только некоторые из них являются определяющими, причем для каждого – свои. В статье говориться, что слепая вера в любую единственную метрику не является хорошей практикой, и следует рассмотреть все доступные метрики в данном проекте. В таком случае будет больше информации при выборе определяющих, что будет вести к информированным и научным решениям вместо безосновательной веры и пустых предположений.

Далее приведен обзор наиболее часто встречающихся метрик.

Метрики сложности программ принято разделять на 3 основные группы [2]:

- метрики размера программ;
- метрики сложности потока управления программ;
- метрики сложности потока данных программ.

Метрики первой группы базируются на определении количественных характеристик, связанных с размером программы, и отличаются относительной простотой. Метрики этой группы ориентированы на анализ исходного текста программ. Поэтому они могут использоваться для оценки сложности промежуточных продуктов разработки.

Метрики второй группы базируются на анализе управляющего графа программы. Управляющий граф программы, который используют метрики данной группы, может быть построен на основе алгоритмов модулей. Поэтому метрики второй группы могут применяться для оценки сложности промежуточных продуктов разработки.

Метрики третьей группы базируются на оценке использования, конфигурации и размещения данных в программе. В первую очередь это касается глобальных переменных.

4.1 Размерно - ориентированные метрики (показатели оценки объема)

LOC-оценка (Lines Of Code)

Количество строк исходного кода (Lines of Code – LOC, Source Lines of Code – SLOC) является наиболее простым и распространенным способом оценки объема работ по проекту.

В зависимости от того, каким образом учитывается сходный код, выделяют два основных показателя SLOC:

a. количество «физических» строк кода – SLOC (используемые аббревиатуры LOC, SLOC, KLOC, KSLOC, DSLOC) – определяется как общее число строк исходного кода, включая комментарии и пустые строки (при измерении показателя на количество пустых строк, как правило, вводится ограничение – при подсчете учитывается число пустых строк, которое не превышает 25% общего числа строк в измеряемом блоке кода).
b. Количество «логических» строк кода – SLOC (используемые аббревиатуры LSI, DSI, KDSI, где «SI» - source instructions) – определяется как количество команд и зависит от используемого языка программирования. В том случае, если язык не допускает размещение нескольких команд на одной строке, то количество «логических» SLOC будет соответствовать числу «физических», за исключением числа пустых строк и строк комментариев. В том случае, если язык программирования поддерживает размещение

нескольких команд на одной строке, то одна физическая строка должна быть учтена как несколько логических, если она содержит более одной команды языка.

Метрика стилистики и понятности программ

Иногда важно не просто посчитать количество строк комментариев в коде и просто соотнести с логическими строчками кода, а узнать плотность комментариев. То есть код сначала был документирован хорошо, затем – плохо. Или такой вариант: шапка функции или класса документирована и комментирована, а код нет.

Fi = SIGN (Nкомм. i / Ni – 0,1)
Общая оценка – F = Σfi

Суть метрики проста: код разбивается на n-равные куски и для каждого из них определяется Fi.

Для подсчета этих двух метрик достаточно лексического анализа, поэтому мы не будем включать их в список рассматриваемых.

Метрики Холстеда

Метрика Холстеда относится к метрикам, вычисляемым на основании анализа числа строк и синтаксических элементов исходного кода программы.

- Основу метрики Холстеда составляют четыре измеряемые характеристики программы: NUOprtr (Number of Unique Operators) — число уникальных операторов программы, включая символы-разделители, имена процедур и знаки операций (словарь операторов);
- NUOprnd (Number of Unique Operands) — число уникальных операндов программы (словарь операндов);
- Noprtr (Number of Operators) — общее число операторов в программе;
- Noprnd (Number of Operands) — общее число операндов в программе.

На основании этих характеристик рассчитываются оценки:

- Словарь программы (Halstead Program Vocabulary, HPVoc): HPVoc = NUOprtr + NUOprnd;
- Длина программы (Halstead Program Length, HPLen): HPLen = Noprtr + Noprnd;
- Объем программы (Halstead Program Volume, HPVol): HPVol = HPLen log2 HPVoc;
- Сложность программы (Halstead Difficulty, HDiff): HDiff = (NUOprtr/2) × (NOprnd / NUOprnd);
- На основе показателя HDiff предлагается оценивать усилия программиста при разработке при помощи показателя HEff (Halstead Effort): HEff = HDiff × HPVol.

4.2 Метрики сложности потока управления программ

Метрики цикломатической сложности по Мак-Кейбу

Показатель цикломатической сложности является одним из наиболее распространенных показателей оценки сложности программных проектов. Данный

показатель был разработан ученым Мак-Кейбом в 1976 г. [MCC], относится к группе показателей оценки сложности потока управления программой и вычисляется на основе графа управляющей логики программы (control flow graph). Данный граф строится в виде ориентированного графа, в котором вычислительные операторы или выражения представляются в виде узлов, а передача управления между узлами – в виде дуг.

Показатель цикломатической сложности позволяет не только произвести оценку трудоемкости реализации отдельных элементов программного проекта и скорректировать общие показатели оценки длительности и стоимости проекта, но и оценить связанные риски и принять необходимые управленческие решения.

Упрощенная формула вычисления цикломатической сложности выглядит следующим образом:

C = e – n + 2, где e – число ребер, а n – число узлов на графе управляющей логики.

Как правило, при вычислении цикломатической сложности логические операторы не учитываются.

В процессе автоматизированного вычисления показателя цикломатической сложности, как правило, применяется упрощенный подход, в соответствии с которым построение графа не осуществляется, а вычисление показателя производится на основании подсчета числа операторов управляющей логики (if, switch и т.д.) и возможного количества путей исполнения программы.

Цикломатическое число Мак-Кейба показывает требуемое количество проходов для покрытия всех контуров сильносвязанного графа или количества тестовых прогонов программы, необходимых для исчерпывающего тестирования по принципу «работает каждая ветвь».

Показатель цикломатической сложности может быть рассчитан для модуля, метода и других структурных единиц программы.

Метрика Майерса

Г. Майерс предложил расширение метрики Мак-Кейба. Суть подхода Г. Майерса состоит в представлении метрики сложности программ в виде интервала [Z(G), Z(G)+h], где Z(G) — метрика Мак-Кейба, а h – показатель сложности предикатов. Для простого предиката h=0, а для n-местных предикатов h=n-1. Такая метрика позволяет различать программы, представленные одинаковыми графами, отличающихся сложностью условных конструкций.

Данная метрика является надстройкой над метрикой Мак-Кейба и реализацией особо отличаться не будет.

Метрика Джилба

Одной из наиболее простых, но, как показывает практика, достаточно эффективных оценок сложности программ является метрика Т. Джилба, в которой логическая сложность программы определяется как насыщенность программы выражениями типа IF-THEN-ELSE. При этом вводятся две характеристики: CL - абсолютная сложность программы, характеризующаяся количеством операторов условия; cl - относительная сложность программы, характеризующаяся насыщенностью программы операторами условия, т. е. cl определяется как отношение CL к общему числу операторов.

Используя метрику Джилба, есть смысл дополнить ее еще одной составляющей, а именно характеристикой максимального уровня вложенности оператора CLI, что позволило не только уточнить анализ по операторам типа IF-THEN-ELSE, но и успешно применить метрику Джилба к анализу циклических конструкций.

Данную метрику можно назвать упрощением метрики Мак-Кейба, так как из всех

ветвлений рассматривает только конструкцию IF-THEN-ELSE.

Метрика граничных значений

Большой интерес представляет оценка сложности программ по методу граничных значений.

Введем несколько дополнительных понятий, связанных с графом программы.

Пусть G=(V,E) - ориентированный граф программы с единственной начальной и единственной конечной вершинами. В этом графе число входящих вершин у дуг называется отрицательной степенью вершины, а число исходящих из вершины дуг - положительной степенью вершины. Тогда набор вершин графа можно разбить на две группы : вершины, у которых положительная степень <=1; вершины, у которых положительная степень >=2.

Вершины первой группы назовем принимающими вершинами, а вершины второй группы - вершинами отбора.

Для получения оценки по методу граничных значений необходимо разбить граф G на максимальное число подграфов G', удовлетворяющих следующим условиям : вход в подграф осуществляется только через вершину отбора; каждый подграф включает вершину (называемую в дальнейшем нижней границей подграфа), в которую можно попасть из любой другой вершины подграфа. Например, вершина отбора, соединенная сама с собой дугой-петлей, образует подграф.

Число вершин, образующих такой подграф, равно скорректированной сложности вершины отбора. Каждая принимающая вершина имеет скорректированную сложность, равную 1, кроме конечной вершины, скорректированная сложность которой равна 0. Скорректированные сложности всех вершин графа G суммируются, образуя абсолютную граничную сложность программы. После этого определяется относительная граничная сложность программы :

`S0=1-(v-1)/Sa,` где S0 - относительная граничная сложность программы; Sa - абсолютная граничная сложность программы; v - общее число вершин графа программы.

4.3 Метрики сложности потока данных программ

Метрики этой группы требуют анализ потока данных, что требует знание областей объявления, использования и изменения операндов программы, то есть необходимо описывать практически полную грамматику языка. Поэтому мы приведем примеры метрик данной группы, но рассматривать их применительно к нашей задаче не будем.

Метрики Чепина

Существует несколько ее модификаций. Рассмотрим более простой, а с точки зрения практического использования – достаточно эффективный вариант этой метрики.

Суть метода состоит в оценке информационной прочности отдельно взятого программного модуля с помощью анализа характера использования переменных из списка ввода-вывода.

Все множество переменных, составляющих список ввода-вывода, разбивается на четыре функциональные группы.

1. Множество «P» – вводимые переменные для расчетов и для обеспечения вывода. Примером может служить используемая в программах лексического анализатора переменная, содержащая строку исходного текста программы, то есть сама переменная не модифицируется, а только содержит исходную информацию.
2. Множество «M» – модифицируемые или создаваемые внутри программы переменные.
3. Множество «C» – переменные, участвующие в управлении работой программного

модуля (управляющие переменные).

4. Множество «T» – не используемые в программе (“паразитные”) переменные. Поскольку каждая переменная может выполнять одновременно несколько функций, необходимо учитывать ее в каждой соответствующей функциональной группе.

Далее вводится значение метрики Чепина:

Q = a1P + a2M + a3C + a4T, где a1, a2, a3, a4 – весовые коэффициенты.

Весовые коэффициенты использованы для отражения различного влияния на сложность программы каждой функциональной группы. По мнению автора метрики наибольший вес, равный трем, имеет функциональная группа C, так как она влияет на поток управления программы. Весовые коэффициенты остальных групп распределяются следующим образом: a1=1; a2=2; a4=0.5. Весовой коэффициент группы T не равен нулю, поскольку “паразитные” переменные не увеличивают сложности потока данных программы, но иногда затрудняют ее понимание. С учетом весовых коэффициентов выражение примет вид:

Q = P + 2M + 3C + 0.5T.

Метрика Спена

Определение спена основывается на локализации обращений к данным внутри каждой программной секции. Спен - это число утверждений, содержащих данный идентификатор, между его первым и последним появлением в тексте программы. Следовательно, идентификатор, появившийся n раз, имеет спен, равный n-1. При большом спене усложняется тестирование и отладка.

4.4 Объектно-ориентированные метрики

Как отмечалось ранее, вводят отдельный класс метрик для оценивания объектно-ориентированного кода. Далее представлены основные его представители.

Метрики Чидамбера и Кемерера [9]

Взвешенная насыщенность класса WMC (Weighted methods per class) - суммарная сложность всех методов класса: WMC=СУММАci, i=1...n, где ci - сложность i-го метода, вычисленная по какой либо из метрик (Холстеда и т.д. в зависимости от интересующего критерия), если у всех методов сложность одинаковая, то WMC=n.

Метрика является обобщением традиционных метрик на классах.

Глубина дерева наследования DIT (Depth of Inheritance tree) - длина самого длинного пути наследования, заканчивающегося на данном модуле. Чем глубже дерево наследования модуля, тем может оказаться сложнее предсказать его поведение. С другой стороны, увеличение глубины даёт больший потенциал повторного использования данным модулем поведения, определённого для классов-предков.

Для многих современных ООП языках существует огромгое количество подключаемых библиотек, что затрудняет возможность подсчета полной глубины вложенности, но позволяет рассматривать вложенность относительную, то есть вложенность классов, непосредственно описанных в предоставленных исходных файлах.

Количество детей NOC (Number of children) - число модулей, непосредственно насле-

дующих данный модуль. Большие значения этой метрики указывают на широкие возможности повторного использования; при этом слишком большое значение может свидетельствовать о плохо выбранной абстракции.

Данная метрика не сильно отличается по реализации от предыдущей, поэтому ее мы не будем рассматривать.

Связность объектов CBO (Coupling between object classes) - количество модулей, связанных с данным модулем в роли клиента или поставщика. Чрезмерная связность говорит о слабости модульной инкапсуляции и может препятствовать повторному использованию кода.

Ответ на класс RFC (Response for a class) - RFC=|RS|, где RS - ответное множество класса, то есть множество методов, которые могут быть потенциально вызваны методом класса в ответ на данные, полученные объектом класса. То есть RS=(({M}({R_i}), i=1...n , где M - все возможные методы класса, R_i - все возможные методы, которые могут быть вызваны i-м классом. Тогда RFC будет являться мощностью данного множества. Чем больше RFC, тем сложнее тестирование и отладка.

Недостаток сцепления методов LCOM (Lack of cohesion in Methods). Для определения этого параметра рассмотрим класс C с n методами M1, M2, ... ,Mn, тогда {I1},{I2},...,{In} - множества переменных, используемых в данных методах. Теперь определим P - множество пар методов, не имеющих общих переменных; Q - множество пар методов, имеющих общие переменные. Тогда LCOM=|P|-|Q|. Недостаток сцепления может быть сигналом того, что класс можно разбить на несколько других классов или подклассов, так что для повышения инкапсуляции данных и уменьшения сложности классов и методов лучше повышать сцепление.

Метрика Мартина [16]

Прежде чем начать рассмотрение метрик Мартина необходимо ввести понятие категории классов. В реальности класс может достаточно редко быть повторно использован изолированно от других классов. Практически каждый класс имеет группу классов, с которыми он работает в кооперации, и от которых он не может быть легко отделен. Для повторного использования таких классов необходимо повторно использовать всю группу классов. Такая группа классов сильно связана и называется категорией классов. Для существования категории классов существуют условия:

- Классы в пределах категории класса закрыты от любых попыток изменения все вместе. Это означает, что, если один класс должен измениться, все классы в этой категории с большой вероятностью изменятся. Если любой из классов открыт для некоторой разновидности изменений, они все открыты для такой разновидности изменений.
- Классы в категории повторно используются только вместе. Они настолько взаимозависимы и не могут быть отделены друг от друга. Таким образом, если делается любая попытка повторного использования одного класса в категории, все другие классы должны повторно использоваться с ним.
- Классы в категории разделяют некоторую общую функцию или достигают некоторой общей цели.

Ответственность, независимость и стабильность категории могут быть измерены путем подсчета зависимостей, которые взаимодействуют с этой категорией. Могут быть определены три метрики :

Центростремительное сцепление Ca. Количество классов вне этой категории, которые зависят от классов внутри этой категории.

Центробежное сцепление Ce. Количество классов внутри этой категории, которые зависят от классов вне этой категории.

Нестабильность I = Ce / (Ca+Ce). Эта метрика имеет диапазон значений [0,1]. I = 0 указывает максимально стабильную категорию. I = 1 указывает максимально не стабильную категорию.

Можно определять метрику, которая измеряет абстрактность (если категория абстрактна, то она достаточно гибкая и может быть легко расширена) категории следующим образом:

Абстрактность A = nA / nAll, где nA — количество_абстрактных_классов_в_категории, nAll — общее_количество_классов_в_категории. Значения этой метрики меняются в диапазоне [0,1]. 0 — категория полностью конкретна, 1 — категория полностью абстрактна.

Для реализации выбранного метода, от метрики требуется возможность (и достаточность) построения скелетной грамматики. Таким образом данный подход применим к подсчету следующих метрик: метрика Джилба, взвешенная насыщенность класса, глубина дерева наследования, метрика Мак-Кейба, ответ на класс, метрика Мартина.

5 Архитектура

В качестве основы для нашего инструмента был выбран ANTLR [13].

ANTLR (от англ. Another Tool For Language Recognition — «ещё одно средство распознавания языков») — генератор анализаторов, позволяющий автоматически создавать лексические и синтаксические анализаторы на одном из целевых языков программирования (C++, Java, C#, Python, Ruby) по описанию LL(*)-грамматики. Является Open Source продуктом, написанным на языке Java (имеет порты на Python, JavaScript, C# и ActionScript).

Данный инструмент был выбран в связи с его популярностью и богатыми возможностями. Так, например, он предоставляет удобные средства для восстановления после ошибок и сообщения о них. В приложении к диплому приведены примеры реализации нечеткого разбора и островных грамматик на основе инструмента ANTLR.

Метод восстановления после ошибок в своей реализации активно использует понятие LocalFollow [10].

LocallFollow(A,B) для нетерминала A, встречающегося в правой части правила вывода нетерминала B есть FIRST от следующего за A нетерминала в этом правиле.

При генерации парсера ANTLR создает LocallFollow для всех возможных пар нетерминалов. В нашей реализации они будут играть роль токенов синхронизации.

В стандартной реализации ANTLR его реакцией на неопределенный нетерминал будет сообщение об ошибке и прекращение построения анализатора. Мы модифицируем процесс генерации анализатора так, что в случае возникновения такой исключительной ситуации генерируется заглушка для неопределенного нетерминала.

Алгоритм конструкции по умолчанию (заглушки) изображен на рисунке №5.

Рисунок №5. Алгоритм конструкции по умолчанию.

Пропуск одного токена, в случае не удачи, необходим для избежания зацикливания(когда этот токен принадлежит множеству LocallFollow).

Для генерации анализаторов на различных целевых языках программирования в ANTLR используется Java StringTemplate Engine.

StringTemplate — инструмент для генерации исходного кода, веб-страниц, почтовых сообщений или любого другого форматированного выходного текста. Его характеризующая черта — строгое разделение модели и представления.

В контексте такой связки, наша заглушка будет описана в нотации StringTemplate для языка Java.

6 Реализация

Для тестирования нашей реализации мы взяли метрику Джилба и грамматику Java 1.4.

Подсчет метрики Джилба можно было реализовать двумя способами:

1. Описать семантические правила и возвращаемые значения в нотации ANTLR.
2. С помощью ANTLR получить TreeWalker и описать его действия в каждом узле.

В своей реализации мы воспользовались первым способом и дополнили грамматику Java семантическими правилами для возвращения количества ветвлений.

В процессе реализации мы столкнулись со следующими сложностями.

1. Не удалось существенно сократить грамматику языка Java при построении грамматики от корня. Это произошло благодаря большой вложенности конструкций в языке Java. К примеру, в правиле
   'if' parExpression statement ('else' statement)?
   нельзя откинуть нетерминал `parExpression`, так как возможны выражения вида:

```
if((new SomeClass()
    {
     public void method(int q){if(true)this.temp=q;}
    }).temp==1)
 {
  System.out.println("123");
 }
```

Листинг №19.

2. Неопределенные нетерминалы возвращают значения.
   Ситуация похожа на попытку вернуть значение у правила, которое его не возвращает — ANTLR сообщает, что такое невозможно. В нашей реализации неопределенный нетерминал не может возвращать значений.
3. Ненужное выполнение семантических действий.
   При попытки разобрать следующее правило могут выполниться все семантические действия в нем содержащиеся. В ANTLR есть специальное свойство backtracking, позволяющее заглядывать вперед без выполнения этих действий.
4. Ошибочное принятие следующего правила разбора.
   Если пропускаемая конструкция полностью содержит в себе следующую, то наш алгоритм примет ее за заглушку и пойдет дальше. Этого не произойдет если на вход подается грамматика от корня.

7 Результаты работы

1. Изучены следующие подходы анализа кода по неполной грамматике: нечеткий разбор, островные грамматики, скелетные грамматики, восстановление ошибок.
2. Выявлены требования для неполной грамматики, обладая которыми, ее возможно разбирать, используя подход на основе скелетных грамматик.
3. Изучены часто используемые метрики анализа кода, такие как метрика Мак-Кейба, метрика Джилба, метрики Крамера и др.
4. Описан инструмент для генерации анализаторов по скелетной грамматике.
5. Проведена модификация инструмента ANTLR для целевого языка Java.
6. Инструмент протестирован на построении анализатора программы на языке Java для подсчета метрики Джилба.
7. Модификация инструмента задокументирована, для последующего ее использования и доработки.

А Нечеткий разбор

Получаемый анализатор выводит в консоль извещение о встрече описанных конструкций.

```
lexer grammar FuzzyJava;
options {filter=true;}

IMPORT
        :       'import' WS name=QIDStar WS? ';'
        ;
RETURN
        :       'return' (options {greedy=false;}: .)* ';'
        ;
CLASS
        :       'class' WS name=ID WS? ('extends' WS QID WS?)?
                ('implements' WS QID WS? (',' WS? QID WS?)*)? '{'
      {System.out.println("found class "+$name.text);}
        ;
METHOD
   :   TYPE WS name=ID WS? '(' ( ARG WS? (',' WS? ARG WS?)* )? ')' WS?
      ('throws' WS QID WS? (',' WS? QID WS?)*)? '{'
       {System.out.println("found method "+$name.text);}
   ;
FIELD
   :   TYPE WS name=ID '[]'? WS? (';'|'=')
       {System.out.println("found var "+$name.text);}
   ;
STAT:   ('if'|'while'|'switch'|'for') WS? '(' ;

CALL
   :   name=QID WS? '('
       {/*ignore if this/super */ System.out.println("found call "+$name.text);}
   ;
COMMENT
   :   '/*' (options {greedy=false;} : . )* '*/'
       {System.out.println("found comment "+getText());}
   ;
SL_COMMENT
   :   '//' (options {greedy=false;} : . )* '\n'
       {System.out.println("found // comment "+getText());}
   ;
STRING
        :       '"' (options {greedy=false;}: ESC | .)* '"'
        ;
```

```
CHAR
        :       '\'' (options {greedy=false;}: ESC | .)* '\''
        ;
WS  :   (' '|'\t'|'\n')+
    ;
fragment
QID :   ID ('.' ID)*
        ;
fragment
QIDStar
        :       ID ('.' ID)* '.*'?
        ;
fragment
TYPE:   QID '[]'?
    ;
fragment
ARG :   TYPE WS ID
    ;
fragment
ID  :   ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*
    ;
fragment
ESC     :       '\\' ('"'|'\''|'\\')
        ;
```

Листинг №20.

В Островная грамматика

Пример само вложенности языков: Simple с комментариями в стиле javadoc (вложенный язык Javadoc), Javadoc с действиями в фигурных скобках(вложенный язык Simple).

```
grammar Simple;

tokens {
          RCURLY='}';
}

@lexer::members {
public static final int JAVADOC_CHANNEL = 1;
public static int nesting = 0;
}

program : (variable)*
          (method)+
        ;
variable: 'int' ID ('=' expr)? ';'
        ;
method  : 'method' ID '(' ')' {System.out.println("enter method "+$ID.text);}
          block
        ;
block   : '{'
            (variable)*
            (statement)+
          '}'
        ;
statement
        : ID '=' expr ';' {System.out.println("assignment to "+$ID.text);}
        | 'return' expr ';'
        | block
        ;
expr    : ID
        | INT
        ;
ID      : ('a'..'z'|'A'..'Z')+ ;
INT     : ('0'..'9')+ ;
WS      : (' '|'\t'|'\n')+ {$channel=HIDDEN;}
        ;
LCURLY  : '{' {nesting++;}
        ;
RCURLY  : '}'
          {
```

```
        if ( nesting<=0 ) {
            emit(Token.EOF_TOKEN);
            System.out.println("exiting embedded simple");
        }
        else {
            nesting--;
        }
        }
      ;
JAVADOC : '/**'
        {
          // создание нового Javadoc-лексера и Javadoc-парсера
          System.out.println("enter javadoc");
          JavadocLexer j = new JavadocLexer(input);
          CommonTokenStream tokens = new CommonTokenStream(j);
          tokens.discardTokenType(JavadocLexer.WS);
          JavadocParser p = new JavadocParser(tokens);
          p.comment();
          // возвращает токен JAVADOC парсеру но в другом канале,
          // благодаря чему, он им не анализируется.
          $channel = JAVADOC_CHANNEL;
        }
      ;
```

```
grammar Javadoc;

comment : ( author )* ;

author  : '@author' ID {System.out.println("author "+$ID.text);} ;

ID      : ('a'..'z'|'A'..'Z')+
        ;
SIMPLE  :   '{'
          {
          System.out.println("enter embedded Simple escape");
          SimpleLexer lex = new SimpleLexer(input);
          CommonTokenStream tokens = new CommonTokenStream(lex);
          SimpleParser parser = new SimpleParser(tokens);
          parser.statement();
          }
          {$channel=HIDDEN;}
        ;
END     : '*/' {emit(Token.EOF_TOKEN);}
        {System.out.println("exit javadoc");}
        ;
```

```
WS      : (' '|'\t'|'\n')+
        ;
```

Листинг №21.

Список литературы

1: Анализатор исходного кода source navigator, 2008. http://sourcenav.sourceforge.net/
2: Новичков Александр. Метрики кода и их практическая реализация в IBM Rational ClearCase// . . . P. .
3: Ефимов Андрей Александрович. Построение ослабленного LALR-транслятора на основе анализа грамматики на избыточность// . 2008. P. .
4: Программа для навигации по исходному коду cscope, 2009. http://cscope.sourceforge.net/
5: А. Ахо, Дж. Ульман. Теория синтаксического анализа, перевода и компиляции. Том 1, Синтаксический анализ. МИР. 1978. P. 611.
6: А. Ахо, Дж. Ульман. Теория синтаксического анализа, перевода и компиляции. Том 2, Компиляция. МИР. 1978. P. .
7: Энциклопедический словарь экономики и права, . http://dic.academic.ru/dic.nsf/econ_dict
8: T. J. McCabe. A complexity measure. IEEE Trans. Software Eng.. 1976. P. .
9: Shyam R. Chidamber, Chris F. Kemerer . A Metrics Suite for Object Oriented Design// . ACM New York, NY, USA. 1991. P. 197 - 211.
10: Rodney W. Topor. A note on error recovery in recursive descent parsers// ACM SIGPLAN Notices. ACM. 1982. P. 37-40.
11: M. Burke, G. Fisher. A practical method for LR and LL syntactic error diagnosisand recovery// ACM Transactions on ProgrammingLanguages and Systems. . 1987. P. 164–197.
12: Rainer Koppler. A systematic approach to fuzzy parsing// . John Wiley & Sons, Inc.. 1997. P. .
13: ANTLR home site, 2010. http://www.antlr.org
14: P. Degano, C. Priami. Comparison of syntactic error handling in LR parsers// Software—Practice and Experience. . 1995. P. 657–679.
15: S.~Klusener and R.~L{\"a}mmel. Deriving tolerant grammars from a base-line grammar// . IEEE Computer Society Press. 2003. P. 10.
16: R.C. Martin. Designing Object-Oriented C++ Applications Using the Booch Method. Prentice-Hall. 1995. P. 528.
17: David P. Tegarden, Steven D. Sheetz, David E. Monarchi. Effectiveness of Traditional Software Metrics for Object-Oriented Systems// . . 1992. P. .
18: M. H. Halstead. Elements of Software Science. . 1977. P. .
19: Leon Moonen. Generating Robust Parsers using Island Grammars// . IEEE Computer Society. 2001. P. 13.
20: David T. Barnard and Richard C. Holt. Hierarchic Syntax Error Repair for LR Grammars// . . 1982. P. .
21: Tim Menzies, Justin S. Di Stefano, Mike Chapman, Ken McGill. Metrics That Matter// . 27th NASA SEL workshop on Software Engineering . 2002. P. 51.
22: D. Grune, C. Jacobs. Parsing Techniques - A Practical Guide. Ellis Horwood Limited. 1990. P. .
23: N.P. Veerman. Revitalizing modifiability of legacyassets// Proc. Conference on Software Maintenanceand Reengineering (CSMR'03). IEEEPress. 2003. P. 19–29.
24: Walter R. Bischofberger . Sniff—A Pragmatic Approach to a C++ Programming Environment// ACM SIGPLAN OOPS Messenger. ACM, New York, NY, USA. 1993. P. 229.
25: J. Heering, P. R. H. Hendriks, P. Klint, J. Rekers. The syntax definition formalism SDF — reference manual—// ACM SIGPLAN Notices. ACM, New York, NY, USA. 1989. P. 43 - 75.
26: Wikipedia, . http://ru.wikipedia.org/wiki/