САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра системного программирования

Авдюхин Дмитрий Алексеевич

Язык описания трансляции для средств реинжиниринга информационных систем

Дипломная работа

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

Научный руководитель:
ст. преп. Я.А. Кириленко

Рецензент:
Руководитель группы в ООО “ИнтеллиДжей Лабс” С.Д. Шкредов

Санкт-Петербург
2013

SAINT-PETERSBURG STATE UNIVERSITY
Mathematics & Mechanics Faculty

Software Engineering Chair

Avdyukhin Dmitry

Translation definition language for informational system reengineering tools

Graduation Thesis

Admitted for defence.
Head of the chair:
professor A.N. Terekhov

Scientific supervisor:
sen. lect. I.A. Kirilenko

Reviewer:
Team lead at "JetBrains" S.D. Shkredov

Saint-Petersburg
2013

Содержание

1 Введение **4**

2 Постановка задачи **6**

3 Обзор существующих решений **7**

3.1 Существующие генераторы парсеров 7

3.2 YaccConstructor 12

3.2.1 Yard 13

3.2.2 RNGLR генератор 15

3.2.3 Преобразования 17

4 Реализация **18**

4.1 Спецификация языка описания трансляций 18

4.2 Изменение синтаксиса существующих конструкций языка Yard 18

4.2.1 Модульность 19

4.2.2 Опции 21

4.2.3 Типы терминалов 22

4.2.4 Приведение к надтипу 22

4.2.5 Разрешение неоднозначностей 23

4.2.6 Расширение существующих конструкций языка Yard 24

4.3 Генератор GLR парсеров 25

4.3.1 Поддержка Yard 25

4.3.2 Оптимизация структур, используемых при разборе 26

4.3.3 Оптимизация SPPF 27

4.3.4 Средства отладки 28

5 Заключение **29**

6 Литература **30**

1 Введение

Одна из важных задач при автоматизированном реинжиниринге программного обеспечения [68] – это разработка синтаксического анализатора входного языка. Она решается современными инструментами, которые предоставляют возможность автоматического создания анализатора на основе спецификации трансляции, задаваемой пользователем. Большинство таких инструментов позволяют описывать контекстно-свободные грамматики, в итоге порождая $LALR(k)$ [16] или $LL(k)$ [39] парсеры.

Однако при реинжиниринге систем возникает проблема, связанная с отсутствием подходящей документации языка.

- Для описания грамматик устаревших языков часто используют не форму Бэкуса-Наура [66], используемую во многих современных инструментах, а синтаксические диаграммы, обладающими бóльшими возможностями.
- Даже имеющаяся контекстно-свободная грамматика языка очень часто является неполной и неоднозначной, что негативно проявляется во время разбора.
- Для некоторых языков (ярким примером является Cobol [51]) существует множество диалектов, и не всегда известно, какой именно диалект использовался при создании системы. Более того, часто в процессе разработки системы сам язык изменяется, дополняется. При этом эти дополнения очень часто остаются незадокументированы, и всплывают уже в процессе синтаксического разбора. Таким образом, встает задача определения диалекта и осуществления соответствующего ему разбора.

Жизненный цикл синтаксического анализатора в проекте по реинжинирингу обладает своими особенностями. На этапе предпродажной подготовки необходимо в кратчайшие сроки реализовать прототип инструмента, позволяющий продемнострировать возможность создания конечного инструмента. В связи с этим, важным требованием является возможность быстрого создания простой версии парсера, обрабатывающего основные конструкции языка. Важным является и то, что нет смысла полностью описывать входной язык, поскольку многие конструкции попросту не встретятся в исходном коде. Как следствие, разработка грамматики происходит итеративно, разбираемое множество строк постепенно расширяется до тех пор, пока не покроет весь код.

Также важно иметь возможность собрать как можно больше информации о том, какую часть кода полученный анализатор может разобрать, а какую нет, чтобы минимизировать число запусков анализатора и иметь лучшее представление об объеме оставшихся работ. Для этого инструмент должен предоставлять возможность продолжать разбор после синтаксических ошибок, что часто требует использования дополнительных конструкций при описании трансляции, поскольку при трансляции нужна нетривиальная обработка некорректного кода.

Таким образом, к языку спецификации трансляций предъявляются следующие требования:

- возможность быстрого изменения описания;
- наличие средств переиспользования кода;
- наличие средств для работы с диалектами;

- возможность описания действий по восстановлению после ошибок;

- быстрота разработки базовой версии синтаксического анализатора.

Поскольку многие из этих требований зависят от алгоритма разбора порождаемых инструментов, помимо разработки языка необходима реализация соответствующего ему генератора трансляторов.

Многие существующие инструменты не удовлетворяют предъявленным требованиям. Цель данной работы – провести их анализ, предложить и реализовать наиболее удобный и выразительный язык спецификации трансляций и сопутствующий инструментарий.

2 Постановка задачи

В рамках данной работы были поставлены следующие задачи.

- Провести анализ существующих инструментов, предназначенных для решения задач реинжиниринга, оценить возможности предоставляемых ими средств описания трансляций.
- Разработать спецификацию языка описания трансляций:
  - сочетающего в себе преимущества инструментов, выявленные в ходе анализа;
  - удовлетворяющего предъявляемым к языку требованиям.
- Разработать генератор трансляторов на основе этого языка.
- Реализовать преобразования, приводящие описание трансляции к виду, принимаемому генератором.

3 Обзор существующих решений

3.1 Существующие генераторы парсеров

Далее будет проведен анализ генератор анализаторов в соответствии с наличием в них различных конструкций, урощающих создание грамматики. При этом некоторые возможности не рассматриваются, так как являются базовыми и встречаются практически во всех инструментах. Среди таких возможностей можно выделить следующие.

- Возможность записи грамматики в BNF-нотации.
- Наличие комментариев.
- Поскольку часто вместе с терминалом связана некоторая сопутствующая информация (например, для терминала NUMBER это может быть соответствующее число), для них необходима возможность указывать тип этой информации.
- Наличие синтезируемых атрибутов – их значение в узлах дерева разбора определяется значениями атрибутов в дочерних узлах. Обычно каждому правилу вывода соответствует функция, отображающее узел дерева разбора в некоторое значение. Таким образом реализуется возможность описания трансляции.
- Возможность описать некий код, который будет просто скопирован в полученный транслятор. Часто он содержит код общего использования, такие как открытия модулей или объявление функций.

Многие инструменты также обладают возможностями описания лексического анализатора, однако эта особенность не рассматривается, поскольку существуют отдельные инструменты, строящие лексеры. Кроме того, многие генераторы LL парсеров предоставляют возможности для описания откатов при разборе. Они также не рассматриваются, поскольку предназначены для решения конфликтов в LL анализаторах, которые не возникают при использовании более мощных алгоритмов разборов.

Далее будут описаны возможности языков описания трансляций, существенно упрощающие процесс разработки грамматики.

- EBNF-нотация [66]. Внутри правил можно использовать сгруппированные альтернативы, то есть допускается запись вида

  ```
  a : b (c|d|e) f;
  ```

  Также вводятся в использования символы '*', '+', '?', которые, примененные к некоторой конструкции в правой части правил, означают, что это констркция встречается: для '*' – 0 или более раз, для '+' – 1 или более раз, для '?' – 0 или 1 раз.

- Явное использование литералов. В правой части правил можно указывать не терминалы, а конкретные строки (например, ключевые слова или знаки препинания):

  ```
  retstat : 'return' expr ';' ;
  ```

- Предикаты. Это условия, невыполнение которых означает, что разбор строки по данному правилу невозможен. Они часто используются в LL–анализаторах для разрешения неоднозначностей. На примере инструмента ANTLR [3]:

```
expr:
    {istype()}? ID '(' expr ')' // ctor-style typecast
  | {isfunc()}? ID '(' expr ')' // function call
  ;
```

  В данном примере в зависимости от того, является входная строка вызовом функции или конструктора, разбор произойдет по соответствующему правилу и создастся соответствующий тип в дереве разбора.

- Именованные связывания: при вычислении синтезируемых атрибутов можно обращаться к дочерним узлам по именам. Альтернативой является обращение по номеру узла в правиле, что усложняет понимание, отладку и написание трансляторов, потому что количество дочерних узлов может превышать десяток.

- Наследуемые атрибуты (или l-атрибуты) - при вычислении значения в узле используется значение, вычисленное в родительском или братском узле. В ANTLR они выглядят следующим образом:

```
add[int x] returns [int result] : '+=' INT {$result = $x + $INT.int;}
```

  В данном примере при вычислении значения в узле используется переданное значение x. Наследуемые атрибуты позволяют сделать описание грамматики более чистым, поскольку позволяют избежать использования глобальных переменных и зависимости от порядка вычислений.

- В таких инструментах, как yacc, есть возможность задавать ассоциативность и приоритеты операторов. Приоритеты гарантируют, что выражение $a * b + c$ разберется как $(a * b) + c$, а ассоциативность – $a/b/c$ как $(a/b)/c$.

- В других инструментах, например в Dypgen, есть возможность указывть приоритеты для целых правил, а не только для операторов.

```
%relation pp<pt
expr :
    | expr(<=pp) PLUS  expr(<pp) { $1 + $3 } pp
    | expr(<=pt) TIMES expr(<pt) { $1 * $3 } pt
    | ...
```

- Иногда грамматика представляется в виде нескольких модулей. На примере ASF+SDF [5]:

```
module A
  ...
  imports B C
  ...
```

В нем содержимое модулей B и C вставляется в модуль A (аналогично тому, как действует *#include* в языке C). Данный подход позволяет разбивать грамматику на относительно независимые части, что упрощает ее совместную разработку. Однако это слабая версия модульности, в которой нетерминалы, объявленные в разных модулях, считаются одинаковыми. Это не позволяет решать такую задачу, как композицию грамматик [43].

- В описании языка можно определять различные опции. Например, инструмент CppCC [12] предоставляет для этого следующий синтаксис:

```
<options_section> -> "OPTIONS" "{" ( <option_name> "=" <option_value> ) * "}"
```

  Это позволяет определять, например, следующие опции:

  – имя модуля, который будет сгенерирован;
  – CASE_SENSITIVE - производить ли чувствительный к регистру разбор;
  – DEBUG_PARSER - добавлять ли в парсер дебажный код.

  Также многие инструменты позволяют указывать опции в качестве аргументов командной строки.

- Dypgen предоставляет возможность динамического расширения грамматики. В процессе разбора грамматика может изменяться, в нее добавляются новые правила. Однако, данная функция не представляется полезной. Изменение грамматики может быть полезно при работе с различными диалектами – в случае, если разобранного кода достаточно, чтоб определить диалект, правила, соответствующие другим диалектам, можно не учитывать в разборе. Но добавление правил не решает эту задачу, и не проще изначального написания тех же правил в грамматике. Также она может использоваться для создания правил, зависящих от семантики, например $a^n b^n$, где n зависит от семантики. Однако такие случаи встречаются крайне редко, и практически всегда приближенные описания являются удовлетворительными.

- В ASF+SDF присутствует конструкция `{S1 S2}+`, эквивалентная `S1 (S2 S1)*`, то есть список из $S1$, разделенных $S2$. Она часто используется для представления аргументов функций и во многих случаях, когда есть множество данных, разделенных запятой. Также ее можно использовать при описании арифметических выражений.

- Правила, параметризованные другими правилами (макро-правила). Например, в Menhir [45] они объявляются следующим образом:

```
%public option(X): | { None } | x = X { Some x }
```

  В данном случае `option(COMMA)` эквивалентно `option_comma`, где

```
option_comma:  { None } | x = COMMA { Some x }
```

  Макро-правила, как и конструкции EBNF, позволяют переиспользовать общие концепции. Например, при помощи них легко записывается описанный ранее список с разделителем:

```
list(elem, sep): head=elem tail=(sep e=elem {e})* {head :: tail}
```

Однако, в этом и многих других инструментах аргументами являются только простейшие конструкции: литералы, нетерминалы (возможно, также с макро-аргументами), терминалы – что является неудобным ограничением.

В результате обзора выяснилось, что нет инструмента со всеми описанными возможностями. Кроме того, на конструкции языка часто наложены существенные ограничения – например, только атомарные конструкции в качестве аргументов макро-правил, часто невозможно использование посчитанных значений в промежуточных вычислениях. Ни один инструмент не предлагает встроенных средств для работы с диалектами. Все это послужило мотивацией для разработки собственного языка.

Информация о рассмотренных генераторах трансляторов кратко представлена в сводных таблицах. Во многом возможности языка описания трансляций связаны с алгоритмом разбора порождаемого инструмента. Например, булевы грамматики и грамматики, разбирающие выражения (PEG [9]) предоставляют неописанные ранее возможности, и для них существуют собственные алгоритмы анализа. Однако, практическая необходимость в них в реинжиниринге требует отдельных исследований, что послужило причиной того, что их возможности описания не рассматриваются в обзоре.

Было замечено, что LL-анализаторы систематически предоставляют более развитые средства для работы с семантикой, такие как предикаты и наследуемые атрибуты, чем прочие алгоритмы разбора, которые часто отсутсвуют у последних. Связано это в первую очередь с тем, что порядок обхода дерева при вычислении семантики совпадает с порядком построения дерева, и часто происходит одновременно с ним. Кроме того, в LL парсерах часто наблюдается естественный приоритет правил: альтернативы разбираются в том порядке, в котором они описаны в грамматике.

Учитывая эту зависимость между средствами описания трансляций и алгоритмом разбора, все инструменты были разбиты на 3 группы: порождающие $LL(k)$ и $LL(*)$ трансляторы, порождающие LR, $LALR$ и GLR трансляторы и все остальные (включая PEG и булевы грамматики). В обзор также включены инструменты, не предоставляющие средств для описания семантики.

Генераторы LL трансляторов:

Инструмент	Литералы	EBNF	Предикаты	l-атрибуты	Макро-правила	Приоритеты правил	Именованные связывания	Модульность/ Многофайловость
ANTLR [3]	+	+	+	+	–	+	+	+
PRECCX [49]	+	+	–	+	+	+	+	–/+
Coco/R [10]	+	+	+	+	–	+	+	–
Toy (TPG) [61]	+	+	+	+	–	+	+	–
LLnextgen [42]	+	+	+	+	–	+	+	–
LLGen [41]	+	+	+	+	–	–	+	–
JavaCC [32]	+	+	+	+	–	–	+	–

Parsec [48]	+	–	+	+	–	+	+	–
Depot4 [15]	+	+	–	–	–	+	+	+/–
JetPAG [34]	+	+	–	+	–	–	+	+
CppCC [12]	+	+	–	+	–	–	+	–
CompTools [27]	+	+	–	–	–	–	+	–
RDP [52]	+	+	–	+	–	–	+	–/+
More than Parsing [2]	+	+	–	–	+	–	–	–
LEPL [38]	+	+	+	–	–	–	–	–
AXE [6]	+	+	–	–	–	–	–	–
SLK [57]	+	+	–	–	–	+	–	–
HiLexed [28]	+	+	–	–	–	?	–	–
Oops [47]	+	+	–	–	–	–	–	–
Grammatica [26]	+ [1]	+	–	–	–	–	–	–
RPATK [53]	+	+	–	–	–	–	–	–
Simple Parser [56]	+	+	–	–	–	–	–	–

Генераторы LR трансляторов:

Инструмент	Литералы	EBNF	Предикаты	l-атрибуты	Макроправила	Приоритеты правил	Именованные связывания	Модульность/ Многофайловость
ASF+SDF [5]	+	+	–	–	–	+	+	+
Menhir [45]	–	+	–	–	+	+	+	+
Hime [29]	+	+	–	–	+	+	–	+
Dypgen [19]	+	+	–	+	–	+	+	–
LAPG [37]	+	+	–	+	–	+	–	–
Beaver [7]	–	+	–	–	–	+	+	–
Irony [31]	+	+	–	–	–	+	–	–
Elkhound [20]	+	–	–	–	–	+	+	–
Bison [8]	+	–	–	–	–	+	–	–
GPPG [24]	+	–	–	–	–	+	–	–
Yacc [62][2]	+	–	–	–	–	+	–	–
VisualParse++ [55]	+	–	–	–	–	+	–	–
JS/CC [35]	+	–	–	–	–	+	–	–
MSTA [46]	+	–	–	–	–	+	–	–

[1]Некоторые инструменты, например Elkhound, Grammatica позволяют использовать в грамматике литералы, однако для них должно быть отдельно описано отображение в обычные токены

[2]В том числе и различные реализации Yacc: BtYacc, byacc, jay . . .

CUP [14]	–	–	–	–	–	+	+	–
QLALR [50]	+	–	–	–	–	–	+	–
Sweet Parser [59]	+	–	–	–	–	+	–	–
Lemon [36]	–	–	–	–	–	+	+	–
CookCC [11]	–	+	–	–	–	+	–	–
Rie [44]	–	–	–	+	–	–	+	–
Frown [30]	+	?	–	?	–	+	+	–
Styx [60]	+	+	–	–	–	–	–	–
Eli [21]	+	+	–	–	–	–	–	–
Essense [23]	+	–	–	–	–	–	–	?
GOLD [25]	+	–	–	–	–	–	–	–
SableCC [54]	–	+	–	–	–	–	–	–
AnaGram [1]	–	–	–	–	–	–	+	–
CSP [13]	–	–	–	–	–	–	–	–
Dragon [18]	–	–	–	–	–	–	–	–

Генераторы, основанные на иных алгоритмах разбора:

Инструмент	Литералы	EBNF	Предикаты	l-атрибуты	Макроправила	Приоритеты правил	Именованные связывания	Модульность/ Многофайловость
N2 [71]	+	+	+ [3]	–	–	+	+	+
Spirit [58]	+	+	+	–	–	+	–	–
APG [4]	+	+	–	–	–	+	–	–
JFLAP [33]	–	+	–	–	–	–	–	–
LISA [40]	+	–	–	–	–	–	–	–

3.2 YaccConstructor

На кафедре системного программирования математико-механического факультета СПбГУ разрабатывается инструмент YaccConstructor [63], позволяющий создавать генераторы синтаксических анализаторов для задач реинжиниринга. Он имеет модульную структуру [69], позволяющую создавать анализаторы с различными языками описания грамматик и алгоритмами разбора. Для создания генератора анализаторов необходимы следующие компоненты.

- Парсер входной грамматики (фронтенд), преобразующий грамматику, описанную при помощи некоторого языка, во внутреннее представление.

[3] В грамматиках, разбирающих выражения, есть предикаты, отличающиеся от уже описанных. Они пытаются провести разбор определенного правила и в зависимости от успеха прерывают или продолжают разбор текущей ветви.

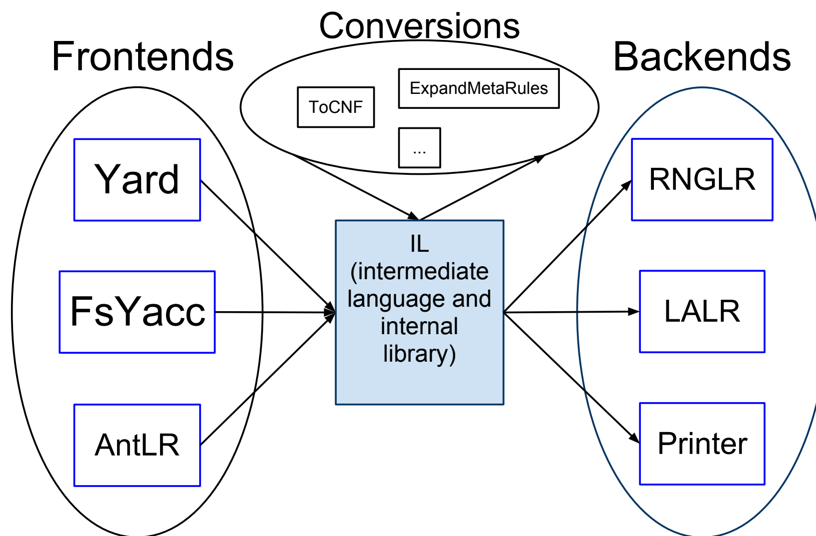

Рис. 1: Структура YaccConstructor

- Генератор (бэкенд), при помощи которого на основе внутреннего представления создаётся транслятор либо синтаксический анализатор или генерируется исходная грамматика на определенном языке, что позволяет порождать эквивалентное описание языка, принимаемое другим инструментом.
- Преобразования грамматики, которые приводят грамматику к эквивалентному виду, удовлетворяющему ограничениям, накладываемым генератором. Например, это может быть устранение левой рекурсии или раскрытие EBNF конструкций.

3.2.1 Yard

В рамках проекта разрабатывается язык описания трансляций Yard [70], предоставляющий следующие возможности.

- Синтезируемые атрибуты, именованные связывания, литералы:

  ```
  use_stmt: "use" dbname=database_name { new UseStmt(dbname) } ;
  ```

  В случае, если синтезируемый атрибут отсутствует, правилу сопоставляется семантика, соответствующая прямому произведению из значений его дочерних узлов. При этом некоторые из узлов могут не участвовать в нем, если перед ними стоят символ '–'. Это часто удобно в небольших выражениях, из которых нужно выбрать один элемент, поскольку позволяет избежать громождкой записи с синтезируемыми атрибутами.

  ```
  expr: -'(' expr -')'
  ```

  равносильно

```
expr: '(' e=expr ')' { e }
```

В случае, если имя связывания является конструкцией более сложной, чем идентификатор (например, при сопоставлении с шаблоном или при явном указании имень типа), имя заключается в угловые скобки:

```
a: ... <_,n,_>=b  <s : string>=c ...
```

- В начале и конце порождаемого генератором файла может находиться некоторый произвольный код. Для этого этот код пишется в фигурных скобках, соответственно, в начале или конце файла грамматики.

- Многофайловость:

```
include "Statements.yrd"
```

Все правила, содержащиеся в подключаемом файле, присоединяются к грамматике.

- EBNF нотация:

```
a: (b|c d)* e?;
```

При этом `a*` и `a+` в результате трансляции порождают списки из входящих в них значений, а `a?` - *Some* из начения `a` или *None*. Также возможно вычисление значений подпоследовательности:

```
stmt: ...(expr | c d)* e?;
```

- Макро-правила:

```
list<<item sep>> : hd=item tl=(-sep item)* { hd::tl };
```

В качестве аргументов им можно передавать только нетерминалы, терминалы и литералы.

- Предикаты:

```
someRule: a=INT =>{ a > 0 }=> "+" b=INT { a + b };
```

После того, как будет вычислено значение a, будет проведена проверка `a > 0`, и, если условие невыполнено, разбор по этому правилу не осуществляется.

- Наследуемые атрибуты:

```
a[x]: n=NUM {x * n};
b : a[2];
```

- Условная компиляция:

```
exec_literal:
  #if ms
    ("EXEC" | "EXECUTE")
  #elif pl
    "EXECUTE" "IMMEDIATE"
  #endif
    "(" LITERAL ")"
  ;
```

Инструменту в командной строке могут определяться опции (например, ms или pl). Аналогично работе препроцессора в языке C, код, следующий за `#if ms`, остается только в том случае, если опция ms была определена.

Благодаря этому возможно переиспользование большей части грамматики при работе с различными диалектами.

- Метки:

```
s: x (@l1(y z)| @l2(a b)) g;
```

У любой последовательности может стоять метка, указывающая на то, какому диалекту она соотвествует. В результате есть возможность определить, с каким диалектом языка идет работа.

Также меткам можно присваивать веса: `@l1[1.2]`. В результате у дерева разбора, в зависимости от того, какие альтернативы в нем присутствуют, появляется некоторый вес. Далее из всех деревьев можно выбрать одно с наибольшем весом, как наиболее соответствующее верному диалекту.

3.2.2 RNGLR генератор

Также в рамках проекта YaccConstructor был реализован генератор, порождающий GLR [64] анализаторы на языке F# [17]. За основу взят RNGLR [22] алгоритм разбора, поскольку обладает преимуществами перед остальными алгоритмами [65]. Порождаемые трансляторы действуют следующим образом.

- На вход им подается последовательность токенов или литералов. Все токены образуют алгебраический тип Token, который порождается генератором. С каждым токеном связана некоторая информация.
  - Для токена NUMBER это может быть соответствующее число, для IDENTIFIER – соответствующее имя идентификатора.
  - Также для нужд реинжиниринга оказывается необходимым для каждого токена знать его координаты в исходном файле, поскольку это помогает определять, где произошла синтаксическая ошибка. Кроме того, при дальнейшей обработке полученного дерева разбора преобразования над некоторыми узлами могут осуществляться с ошибкой, и пользователю необходимо знать координаты этих узлов.

  – Суммируя все сказанное, наиболее применимым на практике является случай, когда все токены хранят в себе следующие данные:
    * Строка в исходном файле, которая соответствует токену. Причем даже для чисел и прочих нестроковых данных оказывается удобным хранить их в виде исходной строки, поскольку это позволяет перенести их в сгенеренный файл без изменений
    * Координаты начала и конца токена. Причем в координатах полезными оказываются имя файла (в случае анализа системы, состоящей из нескольких файлов), смещение относительно начала файла (для работы с файлом как со строкой), строка и столбец (которые удобны для восприятия человеком).
  – В рамках проекта [67] была реализована библиотека для привязки, эффективно хранящая указанные значения. Поскольку полученное дерево может содержать сотни миллионов узлов, и в каждом хранится привязка, то объем этих данных может достигать нескольких гигабайт, что послужило причиной для создания оптимальной структуры.

- Далее по последовательности токенов строится Shared Packed Parse Forests (SPPF) [22] – структура, содержащая в сжатом виде все деревья, которые можно постороить. В случае ошибки будет выдан токен, на котором она произошла. Так как в токене хранится его позиция, то у пользоватекся есть возможность узнать, где она произошла.
- SPPF фильтруется – либо из него удаляются циклы, либо из всех вариантов разбора оставляется только один.
- Происходит трансляция оставшихся деревьев.

Данный генератор создавался как "родной" для Yard, то есть максимально поддерживающий его конструкции. Однако, он обладал рядом недостатков.

- Отсутствие непосредственной поддержки литералов. Все литералы приходилось заменять на текстовые эквиваленты (что делается одним из преобразований, но только для ключевых слов).
- Отсутсвие поддержки предикатов.
- Неоднозначность фильтрации - нельзя было предсказать, какое дерево останется
- Большой расход памяти. В результате разбора миллиона строк кода на Transact SQL полученный SPPF занимал 20 ГБ памяти.
- Большое время работы. На тесте, состоящем из 10^5 строк на Transact SQL, парсер, написанный на fsyacc завершал работу за 10 секунд. GLR алгорит работал 3 минуты. Поскольку алгоритм GLR анализа сложнее, и строит громоздкое промежуточное представление деревьев, ожидалось, что время его работы будет заметно больше по сравнению с fsyacc, который основан на $LALR(1)$ алгоритме разбора. Однако, подобная разница во времени работы является серьезным недостатком.

Перечисленные недостатки усложняют применение генератора на практике, в результате чего возникла необходимость его доработки.

3.2.3 Преобразования

Некоторые конструкции языка генератор не может обработать, поэтому в инструменте присутствуют преобразования, которые от них избавляются. Среди реализованных в рамках YaccConstructor можно выделить следующие:

- раскрытие EBNF;
- раскрытие макро-правил;
- замена литералов на терминалы (обычно добавлением префикса “KW_”);
- добавление синтезируемых атрибутов по умолчанию – в соответствием с тем, что говорилось в описании Yard в случае, если синтезируемый атрибут отсутствует.

4 Реализация

4.1 Спецификация языка описания трансляций

Поскольку язык Yard уже обладает широкими возможностями описания трансляций, было принято решение доработать его спецификацию вместо того, чтобы разрабатывать новый язык. Аналогично, реализацией языка также является доработанный фронтенд для Yard.

4.2 Изменение синтаксиса существующих конструкций языка Yard

В Yard был внесен ряд изменений, основанных на практическом применении инструмента, не вносящих новых возможностей, однако упрощающих описание трансляции.

- Ранее литералы писались в двойных кавычках: `"go"`. Этот синтаксис был заменен на `'go'`, поскольку читаемость не изменяется, но сам литерал вводится проще, поскольку не использует лишние клавиши.
- В процессе использования выяснилось, что символ ‘;’, ставящийся в конце правила, часто забывается, поэтому он стал опциональным. Однако, это приводит к неоднозначностям, поскольку символ ‘+’, обозначающий стартовое правило, также является одним из значащих символов в EBNF нотации.

  ```
  a : b
  +c : d
  ```

  в результате лексического разбора дает ту же последовательность, что и

  ```
  a : b+
  c : d
  ```

  Чтобы избавиться от неоднозначностей, стартовое правило обозначается не ‘+’, а атрибутом `[<Start>]`.
- Поскольку макро-правила используются достаточно часто, синтаксис с двойными угловыми скобками является громоздким. Поэтому решено было использовать одиночные угловые скобки. Однако, потребовалось изменить синтаксис для сложных связываний (которые также пишутся в угловых скобках), которые стали писаться внутри фигурных скобок. Такое решение позволяло избежать сложного лексического анализа, поскольку такие связывания разбираются также, как и наследуемые атрибуты, потому что также содержат обычный код на целевом языке.
- В случае, если EBNF конструкция применяется к сложному выражению, оно заключается в скобки. При этом наиболее используемой из таких конструкций является ‘?’, причем часто она применяется к сложным выражениям. В результате грамматика сильно загромождается выражениями `(...)?`, что плохо влияет на ее читаемость:

```
create_proc:
    KW_CREATE (KW_PROCEDURE | KW_PROC)
```

```
    (ident '.')? ident (';' DEC_NUMBER)?
    brace_opt<'(' comma_list<proc_formal_param>? ')'>
    ('with' (execute_as)+ )?
    ('for' 'replication')?
    'as' 'begin'? (proc_body_stmnt ';'?)* 'end'?
    ';'?
```

Как альтернатива этому выражению введена конструкция `[...]`. В результате пример будет выглядеть следующим образом:

```
create_proc:
    KW_CREATE (KW_PROCEDURE | KW_PROC)
    [ident '.'] ident [';' DEC_NUMBER]
    brace_opt<'(' [comma_list<proc_formal_param>] ')'>
    ['with' (execute_as)+]
    ['for' 'replication']
    'as' 'begin'? (proc_body_stmnt ';'?)* 'end'?
    ';'?
```

Однако, такой синтаксис уже используется для наследуемых атрибутов. Поскольку они используются довольно редко, для них был выбран громоздкий синтаксис с двойными угловыми скобками, ранее использовавшийся в макро-правилах.

4.2.1 Модульность

Для разбиения на файлы используется уже имеющееся выражение `include "имя_файла"`. Однако теперь каждый файл состоит из модулей. Разбиение на модули, в отличие от простого разбиения на файлы, позволяет структурировать грамматику, уменьшая число связей между различными ее частями и делая их более явными, тем самым несколько усложняя написание трудноотлаживаемого кода.

Бывает два типа модулей: обычные и безымянные. Обычный модуль имеет вид:

```
[<AllPublic>] - опционально;
module имя_модуля
[open модуль1, ... ,модульN]
правила*
```

Модуль имеет имя, которое позволяет дргим модулям ссылаться на него, то есть использовать правила, объявленные в нем. Для этого необходимо указать используемый модуль в блоке open вызывающего модуля. Например:

```
module A
open B, C // теперь в A доступны правила, которые предоставляют B и C.
```

Никакие два модуля не могут иметь одинаковое имя.

Однако, не все правила из модуля доступны. Те правила, которые видны из других модулей, будем называть открытыми. Соответсвенно, правила, которые не видны, будем именовать закрытыми. По умолчанию все правила закрыты, однако правило можно сделать открытым, указав перед ним модификатор `public`:

```
module A
    public a: X
    b: Y


module B
    open A
    c: a // OK - a - открытое правило
    d: b // Ошибка - b - закрытое правило
```

Также, в случае, если перед модулем указан атрибут `[<AllPublic>]`, все его правила станут открытыми, за исключением тех, перед которыми явно указан модификатор `private`.

Использование модификаторов доуступа упрощает разработку и поддержку грамматик по причинам, схожим с разработкой на объектно-ориентированных языках. Связей между составляющими грамматики становится меньше, и они все явно прописаны. Поэтому рекомендуется все открытые правила описывать в начале модуля, что позволяет легче определить, какие правила доступны.

Атрибут `[<AllPublic>]` был добавлен, поскольку в грамматике может возникнуть необходимость объявить модуль с общими, зачастую параметризованными правилами, которые могут в равной степени использоваться другими модулями. В остальных случаях его использование неоправдано.

Рассмотрим пример:

```
module A
open B, C
```

Внутри модуля A будут доступны те правила, которые объявлены в A, и открытые правила из B и C. Каждое правило объявляет некоторый нетерминал, который стоит в его левой части. Один и тот же нетерминал не может встречаться дважды в одной области видимости. Пользователь может получить сообщения об ошибке в следующих случаях.

- Если в каком-либо правиле из A в правой используется нетерминал, для которого не существует правила вывода, доступного в A.
- Какой-либо нетерминал в A объявлен дважды.
- Какой-либо нетерминал в A совпадает с каким-то доступным нетерминалом из B или C. Либо и в B, и в C есть доступный нетерминал с одним и тем же именем.

Кроме самого сообщения выдается информация о положении ошибки и модулях, с которыми ошибка связана.

Такая организация модулей позволяет достаточно легко решать задачу композиции грамматик – получение по двум грамматикам новой, которая принимает строки, разбираемые хотя бы одним из

языков. В обычном случае это осложнялось бы пересечением имен нетерминалов, что требовало бы нетривиальных исправлений грамматик. Теперь же в худшем случае требуется переименование некоторых модулей, что представляет собой гораздо более простую задачу. Данный подход предоставляет простую форму работы с диалектами – если у пользователя есть грамматики языка для некоторых диалектов, то он может взять их композицию и получить грамматику, которая принимает оба диалекта.

В одном файле может быть объявлено несколько модулей. Модуль заканчивается либо с концом файла, либо с началом следующего модуля. Среди модулей может быть один безымянный. Он должен быть первым модулем в главном файле (который подается на вход инструменту). Безымянный модуль отличается от обычных отсутствием заголовка (`module имя_модуля`). Поскольку у него нет имени, на него не могут ссылаться другие модули, и атрибут `[<AllPublic>]` также к нему не применим. Таким образом, безымянный модуль имеет следующую структуру:

```
[open модуль1, ... ,модульN]
правила*
```

Безымянный модуль полезен, если создается небольшая однофайловая, часто тестовая, грамматика. Также он может быть применим в случае большой грамматики, поскольку позволяет выделить главный модуль, на который никто не может ссылаться.

Поскольку большинство генераторов на вход принимают список правил, появилась необходимость реализовать преобразование, которое превращает модули в этот список. В случае наличия нетерминалов из разных модулей с одинаковыми именами их приходится переименовывать. По умолчанию в такой ситуации к нетерминалу приписывается префикс – имя модуля, в котором он объявлен. Если и в этом случае возникают пересечения, то в качестве суффикса указывается некоторое число – минимальное, чтобы разрешить неоднозначности.

4.2.2 Опции

Многим генераторам YaccConstructor и некоторым преобразованиям для работы требуются дополнительные опции. Ранее они передавались в качестве аргументов в командной строке:

```
YaccConstructor ... -g "RNGLRGenerator -token SourceText -pos Translator.SqlNodes.Position
                             -module Parser -o Parser.fs" ...
```

Такая запись является очень громоздкой; кроме того, в командную строку выносится информация, имеющая непосредственное отношение к самому описанию трансляции (типы токенов, типы позиций в исходном файле). Поэтому было решено добавить в исходный файл возможность для указания этих опций в самой грамматике.

```
options {
    opt1='val1'
    opt2='val2'
}
```

Для примера выше блок опций будет выглядеть следующим образом:

```
options {
    token=SourceText
    pos="xFormer.Translator.SqlNodes.Position"
    module=Parser
    o="Parser.fs"
}
```

4.2.3 Типы терминалов

Поскольку с каждым токеном связана некоторая информация, добавлена возможность указывать ее тип. Пример использования:

```
tokens {
    | NUMBER of double
    | SELECT
    | _ of string
}
```

`_ of string` означает, что тип всех токенов, для которых он не указан, будет `string`. Как ранее говорилось, в большинстве случаев достаточно, чтобы токены имели один и тот же тип, содержащий исходную строку и позицию в файле, поэтому данный способ описания обладает преимуществом перед стандартным подходом, когда для каждого токена явно указывается тип. До того, как был введен описанный синтаксис, генератору в качестве аргумента можно было передавать тип, однаковый для всех токенов, и даже этого часто было достаточно.

4.2.4 Приведение к надтипу

При описании трансляции в случае с языком F# очень часто встречается такой код:

```
public sql_stmt:
    s=execute_stmt { s :> Statement}
  | s=writetext_stmt { s :> Statement }
  | s=set_stmt  { s :> Statement }
  | s=set_local { s :> Statement }
  | s=declare_local { s :> Statement }
```

Связано это с тем, что в F# нет автоматического приведения к надтипу. В результате появляется много бессмысленного кода (правило может состоять их десятков альтернатив, и в каждой приходится осуществлять приведение), ухудшается читаемость. Для того, чтобы это решить, введена конструкция `{:> Надтип}`, применимая только к альтернативам. С ее помощью пример запишется следующим образом:

```
public sql_stmt: {:> Statement}
    execute_stmt
  | writetext_stmt
```

```
  | set_stmt
  | set_local
  | declare_local
```

4.2.5 Разрешение неоднозначностей

Вместо использования приоритетов правил был предложен следующий подход. В случае возникновения неоднозначностей из нескольких альтернатив выбирается та, которая в грамматике описана раньше. Например в данной грамматике

```
if_stmt:
    'if' expr 'then' stmt
  | 'if' expr 'then' stmt 'else' stmt
```

`else` всегда будет относиться к последнему if-у, поскольку в случае возникновения неоднозначности выбор будет сделан в пользу варианта, где на верхнем уровне нет `else`. Чтобы этого добиться, при возниконовении конфликта, когда одна и та же строка выводится из одного и того же нетерминала по разным правилам, генератор выбирает тот вариант, в котором правило описано в грамматике раньше.

Поскольку неоднозначная грамматика часто является громоздкой, а описанный подход не всегда позволяет легко или очевидно для пользователя разрешать неоднозначности, было решено добавить в язык низкоуровневые конструкции, позволяющие изменять построенные таблицы. Кроме того, в случае разбора GLR алгоритмом даже однозначной грамматики иногда возникают ситуации, когда возникающие в результате построения $LALR$-таблиц конфликты приводят к увеличению разбора, и модификация таблиц может позволить оптимизировать известные узкие места. Во многих случаях конфликты разрешаются, если известно, какой символ может или, наоборот, не может оказаться в данной позиции. Для этого введены следующие конструкции:

```
  /<token> - на данной позиции ожидается данный токен.
  /^<token> - на данной позиции не ожиадается данный токен
  /[<token1>; ... ; <tokenN>] - ожидается один из указанных токенов
  /^[<token1>; ... ; <tokenN>] - не ожидается ни один из указанных токенов
```

В результате пример выше можно записать в таком виде:

```
if_stmt:
    'if' expr 'then' stmt /^'else'
  | 'if' expr 'then' stmt 'else' stmt
```

Преимущество данного подхода заключается в том, что, во-первых, теперь порядок правил не важен и, во-вторых, неоднозначности не разрешаются алгоритмом, а попросту не возникают. В данном примере первое правило не будет применено, если после него стоит литерал `'else'`, в результате чего `'else'` всегда будет относиться к последнему `'if'`, что и требовалось.

Рассмотрим другой пример. В языке Transact SQL есть оператор `return`, который осуществляет выход из процедуры и при выходе из функции, возможно, возвращает значение: `return 2`. Также в Transact SQL есть конструкция `(SELECT ...)`, одна из версий которой осуществляет выборку из

таблицы и возвращает на ее основе некоторое значение. При этом она может являться как выражением, так и оператором. Это приводит к тому, что строка `return (SELECT ...)` может трактоваться и как выход из функции с возвратом значения (что является верным), и как два не связанных оператора. При этом конфликт возникает на уровне блока операторов, что делает его разрешение довольно сложным. При помощи введенной конструкции неоднозначность решается следующим образом:

```
return_stmt:
    'return' expr
  | 'return' /^'('
```

4.2.6 Расширение существующих конструкций языка Yard

Добавлена возможность указывать у нетерминала несколько l-атрибутов.

```
n<<a b>><<a c>>
```

Это является важной возможностью в процессе преобразования грамматики. Фактически, результатом трансляции нетерминалов с наследуемыми атрибутами становятся функции, аргументами которых являются атрибуты в том виде, в котором были записаны внутри `<<...>>`. Поэтому приведенный пример сгенерирует выражение

```
fun a b -> fun a c -> ...
```

Более простое выражение `n<<a b a c>>` было бы преобразовано в

```
fun a b a c -> ...
```

, что запрещено семантикой целевого языка, потому что содержит два одноименных аргумента a.

Ранее внутри EBNF конструкций могли содержаться произвольные выражения, однако у них был тот недостаток, что в них нельзя было использовать объявленные ранее семантические значения – l-атрибуты и связывания после того, как они определены, могут далее использоваться при вычислении значений элементов правила, стоящих правее них. Это привело к усложнению преобразования, раскрывающего EBNF. Оно действует следующим образом: накапливаются все семантические значения, из них формируются l-атрибут, который передается в раскрытое правило. Пример:

```
a<<u>>: v=x (y {u + v})*
```

раскроется в

```
a<<u>>: v=x many1<<u v>>
many1<<u v>> : {[]} | item=(y {u + v}) tail=many1<<u v>> { item::tail }
```

При этом, чтобы избежать лишних использований наследуемых атрибутов, проверяется, возможно ли использование семантических значений.

```
a<<u>>: v=x y*
```

раскроется в

```
a<<u>>: v=x many1
many1 : {[]} | item=y tail=many1 { item::tail }
```

Была предпринята попытка в качестве аргументов макро-правил передавать произвольные выражения. Однако, с ними описанное решение с l-атрибутами не работает, потому что макро-правила могут вызываться рекурсивно из самих себя. При этом количество передаваемых аргументов будет расти, будут создаваться новые правила, и этот процесс не сходится. На данный момент не было найдено надежного способа раскрывать макро-правила с произвольной семантикой, поэтому в их аргументах запрещается использование семантических значений, объявленных за пределами правила.

Пример правила, раскрытие которого породит некомпилирующийся код:

```
present<item>:  {false} | item {true}
a<<x>> : present<b<<x>> >
```

`present<b<<x>> >` будет раскрыто следующим образом:

```
yard_present:  {false} | b<<x>> {true}
```

В случае, если не объявлена глобальная переменная x, она не будет видна в данной области и порожденный код не будет компилироваться.

4.3 Генератор GLR парсеров

В качестве генератора парсеров был выбран описанный ранее генератор RNGLR трансляторов. Его преимуществом является то, что он максимально поддерживает предыдущую реализацию Yard. Кроме того, GLR алгоритм предоставляет средства, позволяющие бороться с конфликтами и упрощающие разработку транслятора. В связи с изменением языка описания пришлось серьезно доработать и генератор, который, кроме того, требовал различных улучшений.

4.3.1 Поддержка Yard

В генераторе были поддержаны следующие конструкции языка:

- Была реализована поддержка литералов. Литералы отличаются от токенов тем, что представляют собой фиксированную, заранее известную строку. Поэтому было решено, что единственное, что они могут хранить – координаты начала и конца. Возможно представить, что они могут хранить какие-либо еще данные, но этот случай очень редко имеет место в реинжиниринге. Кроме того, саму строку можно восстановить, взяв подстроку из исходного файла с соответствующими координатами.

  Каждому литералу соответствует свой тип токена:

  ```
  ``L <value>`` of <pos>*<pos>
  ```

  , где `<value>` - значение литерала, а `<pos>` – тип координат, который указывается пользователем при генерации. Также, как и токены, литералы можно создавать стандартным конструктором, однако этот способ неудобен. Поэтому введена функция `genLiteral (string, startPos, endPos)`, которая либо создает нужный тип токена, либо бросает исключение.

- Реализована поддержка координат при трансляции. В реинжиниринге важной оказывается возможность для каждого правила знать координаты его начала и конца в исходном файле. Поэтому в функции трансляции для правил дополнительно передаются их координаты. Координаты узла в дереве вычисляются на основе самого левого и самого правого непустого дочернего узла. В случае, если рассматривается ε-дерево, координаты вычисляются на основе этих зачений для ближайших непустых соседей.

- Раньше при фильтрации леса из всех деревьев раньше бралось самое первое. Так как алгоритм анализа и построения леса достаточно сложен, определение того, какое именно дерево будет выбрано, является очень сложной задачей. Поскольку подобная неопределенность может приводить к нестабильной работе и усложнять отладку, ее необходимо было решить. Для этого был реализован алгоритм фильтрации, действующий по принципу "longest match", то есть отбирающий из двух деревьев то, в котором первый отличающийся по координатом узел имеет большую конечную координату.

- Была реализована поддержка предикатов. Они генерируются в условные операторы, которые на этапе трансляции не добавляют к результату те деревья, для которых указанное условие не выполняется.

4.3.2 Оптимизация структур, используемых при разборе

В целях оптимизации была изменена структура графа, который строится в процессе разбора (Graph-structured stack [22], GSS). Вершина в графе соответствует паре – LALR состояние и номер токена, после рассмотрения которого они появились (будем называть этот номер уровнем вершины). На ребрах графа содержится информация о деревьях разбора.

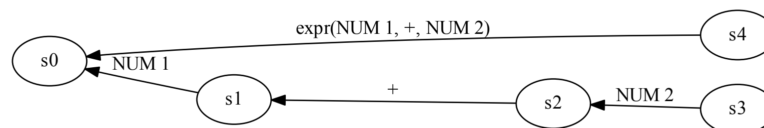

Рис. 2: Пример GSS

В каждой вершине графа для хранения ребер был использован станжартная структура List, позволяющая динамически добавлять элементы. Однако, она требует много памяти – на хранение структуры с всего одним ребром тратятся 80 байт. Из алгоритма разбора следует, что ребра добавляются только к вершинам на последнем уровне. Таким образом, для вершин на меньших уровнях достаточно статических массивов. Статический массив с одним ребром занимает уже 40 байт, то есть вдвое меньший объем. Более того, из почти всех вершин выходит ровно одно ребро, так что вместо массива ребер рассматривается структура данных, состоящая из одного ребра и массива остальных ребер, который в случае, если у вершины только одно исходящее ребро, равен `null`. Таким образом для вершин с несколькими ребрами ничего не меняется, а с одним ребром получется выигрыш еще в 24 байта.

Ребра, выходящие из вершин последнего уровня, складываются в динамический массив для каждого состояния. После того, как все ребра добавлены, полученные динамические массивы превращаются в описанную выше статическую структуру. Для более быстрого добавления ребер в динамический массив или поиска существующего в нем ребра решено применить эвристику: хранить ребра в порядке убывания уровня конечной вершины, поскольку большинство сверток происходят именно в таком порядке. На данный момент не найдено примера, на котором поиск занимает не константное время.

4.3.3 Оптимизация SPPF

SPPF [22] – структура, позволяющая хранить все возможные деревья разбора в сжатом виде. SPPF состоит из узлов, каждый из которых соответствует некоторой части входного потока и представляет собой одну из трех следующих конструкций.

- Множество семейств. Семейство представляет собой вывод некоторой подстроки входа из некоторого нетерминала. Таким образом, она состоит из номера правила, по которому происходит вывод, и массива дочерних узлов.
- Терминал. Все терминалы хранятся в отдельном массиве, и в качестве узла выступает его номер – целое неотрицательное число. Соответствует выводу одного терминального символа.
- ε-дерево - соответствует выводу пустой строки из какого-то нетерминала. Все ε-деревья предпосчитаны статически и хранятся отдельно, так что вместо самого дерева используется только ссылка на него – отрицательное число, вычисляемое по номеру нетерминала, выводящего данную пустую строку.

Если вывод правила содержит в конце несколько ε-деревьев, то они не фигурируют в дереве, в результате чего правило оказывается короче. Это важно, поскольку многие правила имеют вид

```
e1 : e2 (op e2)*
```

Причем в случае с арифметическими выражениями эти правила имеют большую вложенность (например, когда в качестве произвольного выражения выступает единственное число), что позволяет получить большой выигрыш по памяти.

Большинство узлов в дереве содержат только одно семейство, так что тут используется та же структура, что и с ребрами в GSS. Большинство семейств в реальной грамматике содержат не более 2 узлов. Поэтому для них рассматривается структура, аналогичная той, что использовалась для ребрами в GSS, с той разницей, что вместо одного обязательного элемента хранится 2 (которые равны null в случае отсутствия).

При вычислении семантики над полученным SPPF было замечено, что его рекурсивный обход приводит к переполнению стэка на глубоких деревьях. Однако, в случае, если лес не содержит циклов, можно упорядочить узлы так, чтобы дети всегда были рассмотрены раньше родителей. В результате трансляция представляет собой проход по массиву, где каждый элемент вычисляется на основе уже вычисленных. Более того, узлы дерева возможно отсортировать так, что в случае отсутствия неоднозначностей (либо после их фильтрации) узлы будут транслироваться слева направо В случае наличия циклов разделение на родителей и детей невозможно. Однако, в этом случае теряет смысл и трансляция, так что в этом случае трансляция останавливается, пользователю сообщается о наличии циклов.

4.3.4 Средства отладки

Для отладки транслятора были реализованы следующие возможности.

- Поскольку грамматика преобразуется, ее конечный вид неизвестен пользователю. Поэтому при генерации транслятора в лог печатаются все правила конечной грамматики, с нумерацией и в максимально простом виде.

- В случае, если пользователь умеет работать со стэком, то ему может помочь вывод построенного GSS, поскольку оттуда может быть видно, почему, например, не произошла такая-то свертка.

- В случае ошибки возвращается токен, на котором она произошла. Однако, этого может быть недостаточно для определения позиции ошибки, поскольку координаты могут к моменту создания парсера выставляться некорректно либо недостаточно точно. Иногда в таких случаях помогает возможность вывода нескольких последних разобранных токенов, что также было реализовано.

- Если разбор завершился успешно, то над полученным деревом можно запустить обход, который выведет информацию о всех неоднозначностях: начальная и конечная позиции, номера всех конфликтующих правил.

- Печать полученного дерева в dot-формате с подсветкой неоднозначных узлов красным цветом. Также в эти узлы добавляется символ '!', что позволяет находить их в дереве при помощи текстового поиска.

5 Заключение

В процессе работы были получены следующие результаты.

- Проведен анализ существующих генераторов парсеров на предмет возможностей описания трансляций, выделены основные особенности, упрощающие разработку транслятора.
- Разработана спецификация языка описания трансляций, сочетающая преимущества рассмотренных инструментов и добавляющая новые. Она получена путем доработки спецификации языка Yard.
- Получена практическая реализация языка за счет доработки фронтенда для языка Yard в инструменте YaccConstructor.
- Получен генератор трансляторов, предназначенный для этого языка, путем доработки RNGLR модуля.
- Доработаны преобразования грамматики, позволяющие раскрывать добавленные конструкции, приводя грамматику к виду, принимаемому RNGLR генератором.

Весь проект YaccConstructor можно найти на сайте https://code.google.com/p/recursive-ascent/. Автор принимал участие в проекте под учетной записью dimonbv.

6 Литература

Список литературы

[1] Anagram. URL: http://www.parsifalsoft.com

[2] *Angel Herranz, Pablo Nogueira*. More than parsing. In Francisco Javier Lopez Fraguas, editor, Spanish Conference on Programming and Languages (CEDI-PROLE'05), pages 193-202. Thomson Paraninfo, September 2005.

[3] ANTLR. URL: http://www.antlr.org/

[4] APG. URL: http://coasttocoastresearch.com/

[5] ASF+SDF. URL: http://www.meta-environment.org

[6] AXE. URL: http://www.gbresearch.com/axe

[7] Beaver. URL: http://beaver.sourceforge.net/

[8] Bison. URL: http://www.gnu.org/software/bison/

[9] *Bryan Ford*. Parsing Expression Grammars: A Recognition-Based Syntactic Foundation. // Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages, стр 111-122. 2004.

[10] Coco/R. URL: http://www.ssw.uni-linz.ac.at/coco/

[11] CookCC. URL: https://code.google.com/p/cookcc/

[12] CppCC. URL: http://cppcc.sourceforge.net/

[13] CSP. URL: http://csparser.sourceforge.net/

[14] CUP. URL: http://www.cs.princeton.edu/~appel/modern/java/CUP/

[15] Depot4. URL: http://www.tools.vlampe.de/depot4/Depot4.html

[16] *Deremer F.L.* Practical translators for LR(k) languages. // Ph.D. thesis, Massachusetts Institute of Technology, 1969.

[17] *Don Syme, Adam Granicz, Antonio Cisternino* Expert F# 3.0. // Издательство “Apress”, 2012. 638 с.

[18] Dragon. URL: http://www.lemke-it.com/litexec?request=pubdragondoc§ion=dragon_content.html&user=&lang=en

[19] Dypgen. URL: http://dypgen.free.fr/

[20] Elkhound. URL: http://scottmcpeak.com/elkhound/

[21] Eli. URL: http://eli-project.sourceforge.net/

[22] *Elizabeth Scott, Adrian Johnstone.* Right Nulled GLR Parsers, 2006.

[23] Essense. URL: http://s48.org/essence/

[24] GPPG. URL: http://gppg.codeplex.com/

[25] GOLD Parsin System. URL: http://goldparser.org/

[26] Grammatica. URL: http://grammatica.percederberg.net/

[27] *Gilbert B.* CompTools: A Compiler Generator for C and Java, 2004.

[28] HiLexed. URL: http://www.hilexed.org/

[29] Hime Parser Generator. URL: http://himeparser.codeplex.com/

[30] *Hinze R.* FROWN An LALR(k) Parser Generator. 2001.

[31] Irony. URL: http://irony.codeplex.com/

[32] JavaCC. URL: https://javacc.java.net/

[33] JFLAP. URL: http://www.jflap.org/

[34] JetPAG. URL: http://jetpag.sourceforge.net/

[35] JS/CC. URL: http://jscc.phorward-software.com

[36] Lemon. URL: http://www.hwaci.com/sw/lemon/lemon.html

[37] LAPG - Lexical Analyser and Parser Generator. URL: http://lapg.sourceforge.net/

[38] LEPL. URL: http://www.acooke.org/lepl/

[39] *Lewis P.M. II, Stearns R.E.* Syntax-Directed transduction. // ACM 15, 3, 465–488, July 1968.

[40] LISA. URL: http://labraj.uni-mb.si/lisa/

[41] LLGen. URL: http://www.cs.vu.nl/~ceriel/LLgen.html

[42] LLnextgen. URL: http://os.ghalkes.nl/LLnextgen/

[43] *M.G.J van den Brand, M.P.A. Sellink, C. Verhoef.* Current Parsing Techniques in Software Renovation Considered Harmful. // Proceedings of the sixth International Workshop on Program Comprehension, стр. 108-117, IEEE, 1998.

[44] *Masataka S., Harushi I., Ikuo N.* Rie, a Compiler Generator Based on a One-pass-type Attribute Grammar. // Software: practice and experience, vol. 25(3), 229–250 (march 1995).

[45] Menhir. URL: http://gallium.inria.fr/~fpottier/menhir/

[46] MSTA. URL: http://cocom.sourceforge.net/

[47] Oops. URL: http://www.cs.rit.edu/~ats/projects/oops3/

[48] Parsec. URL: http://legacy.cs.uu.nl/daan/parsec.html

[49] PRECCX. URL: http://preccx.sourceforge.net/

[50] QLALR. URL: http://qt-project.org/quarterly/view/qlalr_adventures

[51] *Ralf Lammel and Chris Verhoef.* Cracking the 500-Language Problem. // IEEE Software, 2001.

[52] RDP. URL: http://www.dcs.rhbnc.ac.uk/research/languages/projects/rdp.html

[53] RPA Toolkit. URL: http://rpatk.net/

[54] SableCC. URL: http://sablecc.org/

[55] *Sandstone technologies inc.* Parsing with sandstone's Visual Parse++. 2001.

[56] Simple Parser. URL: http://cdsoft.fr/sp/sp.html

[57] SLK. URL: http://slkpg.byethost7.com/

[58] Spirit. URL: http://boost-spirit.com/home

[59] Sweet Parser. URL: http://www.sweetsoftware.co.nz/parser_overview.html

[60] Styx. URL: http://www.speculate.de/

[61] Toy Parser Generator. URL: http://cdsoft.fr/tpg/

[62] Yacc и Lex. URL: http://dinosaur.compilertools.net/

[63] YaccConstructor. URL: http://code.google.com/p/recursive-ascent/

[64] *Tomita M.* Efficient Parsing for Natural Language, 1986.

[65] *Авдюхин Д.* Создание генератора GLR трансляторов для .NET, 2012. 17 с.

[66] *Ахо А., Сети Р., Ульман Дж.* Компиляторы: принципы, технологии, инструменты. // М.: Издательский дом "Вильямс", 2003. — 768 с.

[67] Библиотека привязки. URL: https://code.google.com/p/recursive-ascent/wiki/Source_Text_Utils

[68] *Терехов А.Н.* Автоматизированный реинжиниринг программ // Издательство Санкт-Петербургского государственного университета, 2000. 332 с.

[69] *Улитин К.А.* Разработка архитектуры для генератора синтаксических анализаторов, 2010. 15 с. // URL: http://recursive-ascent.googlecode.com/files/KonstantinUlitin_CompilerCompilerArchitecture.pdf

[70] *Чемоданов И.С.* Генератор синтаксических анализаторов для решения задач автоматизированного реинжиниринга программ, 2007. 37 с. // URL: http://recursive-ascent.googlecode.com/files/IlyaChemodanov_Yard.pdf

[71] *Чистяков В.Ю.* N2 - языковый фреймворк. // RSDN Magazine. 2012. http://www.rsdn.ru/article/nemerle/N2/N2-Project.rsdnml.xml