САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Математико-механический факультет

Кафедра системного программирования

ИНСТРУМЕНТ РЕИНЖИНИРИНГА СПЕЦИФИКАЦИЙ ТРАНСЛЯЦИЙ

Дипломная работа студента 545 группы

Константина Андреевича Улитина

|  |  |  |
| --- | --- | --- |
| Научный руководитель | ……………… / подпись / | с.н.с. НИИ ИТ СПбГУ Я. А. Кириленко |
| Рецензент | ……………… / подпись / | ведущий инженер-программист Н. М. Тимофеев |
| “Допустить к защите” заведующий кафедрой, | ………………  / подпись / | д.ф.-м.н., проф. А. Н. Терехов. |

Санкт-Петербург

2011

SAINT PETERSBURG STATE UNIVERSITY

Mathematics and Mechanics Faculty

Software Engineering Department

TRANSLATION SPECIFICATION REENGINEERING TOOL

Graduate paper by

Konstantin Ulitin

545 group

|  |  |  |
| --- | --- | --- |
| Scientific advisor | ……………… | Senior Staff Scientist J. A. Kirilenko |
| Reviewer | ……………… | Lead Developer N. M. Timofeev |
| “Approved by” Head of Department | ……………… | PhD, Professor A. N. Terekhov |

Saint Petersburg

2011

Оглавление

[Введение 5](#_Toc295142294)

[Выбор базовой технологии 7](#_Toc295142295)

[DMS Software reengineering toolkit 7](#_Toc295142296)

[TXL 8](#_Toc295142297)

[Stratego/XT 8](#_Toc295142298)

[YaccConstructor 9](#_Toc295142299)

[Результаты обзора 9](#_Toc295142300)

[Реализация 11](#_Toc295142301)

[Типы узлов дерева разбора 11](#_Toc295142302)

[Особенности внутреннего представления 13](#_Toc295142303)

[Разрешение конфликтов 13](#_Toc295142304)

[Правила с одинаковыми именами 13](#_Toc295142305)

[Целевой язык 13](#_Toc295142306)

[Типы возвращаемых значений синтаксических конструкций 13](#_Toc295142307)

[Порядок вычисления атрибутов 14](#_Toc295142308)

[Преобразования внутреннего представления 15](#_Toc295142309)

[AddEOF 15](#_Toc295142310)

[ReplaceLiterals 15](#_Toc295142311)

[ExpandBrackets 15](#_Toc295142312)

[BuildAST 16](#_Toc295142313)

[ExpandEnbfStrict 16](#_Toc295142314)

[Модульность спецификации трансляции 16](#_Toc295142315)

[LeaveLast 17](#_Toc295142316)

[MergeAlter 18](#_Toc295142317)

[Парсеры спецификаций трансляции 18](#_Toc295142318)

[AntlrFrontend 18](#_Toc295142319)

[FsYaccFrontend 18](#_Toc295142320)

[Генераторы спецификаций трансляции 19](#_Toc295142321)

[FsYaccPrinter 19](#_Toc295142322)

[SqlMigration 20](#_Toc295142323)

[Claret 21](#_Toc295142324)

[Заключение 22](#_Toc295142325)

[Примеры использования преобразования BuildAST 23](#_Toc295142326)

[Список использованных источников 24](#_Toc295142327)

# Введение

В программном обеспечении, так или иначе работающем с некоторым формальным языком, вместо реализации синтаксического анализатора или транслятора вручную часто пользуются специализированным программным обеспечением — генераторами синтаксических анализаторов. Такие инструменты автоматически порождают алгоритм, выполняющий синтаксически-управляемую трансляцию по заданной спецификации трансляции. В этом случае спецификация трансляции становится таким же артефактом разрабатываемого программного комплекса, как и исходный код.

Каждый генератор анализаторов имеет свой язык описания грамматики, алгоритм генерации, и порождает транслятор, работающий по строго предопределённому алгоритму на некотором классе языков. Помимо этого, инструменты отличаются и другими характеристиками, включающими скорость работы сгенерированного анализатора, возможность восстановления после ошибок и другие. Более подробный обзор можно найти в работе [1].

Часто одна характеристика генератора анализаторов зависит от другой. В частности, многие из них привязаны к классу алгоритма построения анализатора. Кроме класса принимаемых грамматик, от него зависит скорость работы сгенерированного анализатора: как правило, парсеры на основе рекурсивного спуска медленнее, чем табличные. Но разрабатывать грамматику проще с использованием рекурсивного спуска, так как наглядно видно, как пытался парсер разобрать строку.

Соответственно, не может быть универсального средства, подходящего для большинства задач. Поэтому на начальном этапе проекта стоит нелегкая задача выбора инструмента, максимально подходящего в данный момент. Такая широта требований объясняет большое число[[1]](#footnote-1) существующих на данный момент генераторов синтаксических анализаторов. Многие из них уже не поддерживаются, но могут представлять ценность для некоторых проектов. В процессе развития грамматики работа с выбранным изначально генератором может быть неудобна или даже невозможна.

Жизненный цикл ни одного программного продукта не обходится без использования стороннего программного обеспечения. Такими приложениями являются как минимум компилятор или интерпретатор, и чаще всего — множество сторонних модулей или библиотек. Это выгодно как с точки зрения экономии времени, так и улучшения качества готового продукта, так как сторонние библиотеки протестированы огромным числом пользователей, и критичность их недостатков можно оценить заранее. Бывает так, что разработка относительно простого продукта сводится к собиранию его из сторонних компонент, их настройке, интеграции и тестированию совместимости. Все вышесказанное применимо и к генераторам синтаксических анализаторов: происходит использование другого приложения для получения кода приложения вместо написания транслятора вручную.

Выбирать конкретный инструмент приходится до его использования, и быстро перейти на альтернативу не представляется возможным по следующим причинам.

* У каждого из них свой формат входных файлов. Это еще осложняется тем, что поддерживаются разные конструкции языка спецификации трансляции.
* Транслятор, сгенерированный по грамматике одного класса алгоритмом для грамматик другого класса, не всегда корректен.
* Возможно, используются разные лексеры, разные целевые языки, разный порядок вычисления атрибутов.

При разработке грамматики для конкретного инструмента возможен выход за рамки его возможностей, или же просто грамматика становится плохо структурированной и сложной для понимания и соответственно дальнейшей модификации. В этом случае целесообразно перейти на другой инструмент, но по указанным выше причинам это потребует дополнительных трудозатрат.

Применительно к информационным системам процесс модернизации, применяемый в таком случае, называют реинжинирингом. Технологиями автоматизированного реинжиниринга, в частности, source-to-source трансляцией на более современный язык программирования, пользуются для преобразования системы с одновременным улучшением архитектуры. При реинжиниринге всегда стоит цель улучшить не только функциональные характеристики, но и обращают внимание на такой важный аспект, как сопровождаемость новой системы.

В рамках данной работы были поставлены следующие задачи.

1. Обосновать выбор технологии, на основе которой строится решение.
2. Реализовать инструмент автоматизированного реинжиниринга спецификаций трансляций, который бы позволял производить преобразования между форматами сторонних генераторов синтаксических анализаторов.
3. Снабдить разработчика функциональностью для более удобной разработки грамматики.
4. Провести апробацию разработанного инструмента.

# Выбор базовой технологии

Задача преобразования спецификации трансляции из формата одного инструмента в формат другого также решается с помощью генератора синтаксических анализаторов, поскольку является задачей трансляции из одного формального предметно-ориентированного языка в другой. В основе большинства языков спецификации трансляции лежит форма Бэкуса-Наура, поэтому их структуры схожи между собой. Однако работа такого транслятора не сводится к простой замене управляющих конструкций. Как уже говорилось, в разных генераторах анализаторов поддерживаются разные возможности языка спецификации трансляции, например, в YARD есть поддержка макроправил [2], которые являются специфическими конструкциями именно этого языка. Поэтому от такого транслятора требуются более сложные преобразования. И, захотев написать спецификации трансляций между форматами каких-то 5 инструментов, нам придется написать 20 спецификаций, по одной на каждую пару.

Другой подход заключается в том, чтобы разбить это отношение многие-ко-многим на много-к-одному и один-ко-многим. То есть, ввести некоторое внутреннее представление, которое бы связывало исходную и преобразованную спецификацию трансляции. Предпочтительно, чтобы это внутреннее представление поддерживало максимально возможное число синтаксических конструкций сторонних инструментов — тогда не придется восстанавливать высокоуровневые конструкции при генерации спецификации трансляции в случае, если они уже и так присутствовали в исходной. Также, в этом случае мы сможем повторно использовать логику, приводящую семантику одного инструмента к семантике другого. Для упомянутого примера нужно написать 10 трансляторов (5 из спецификации трансляции во внутреннее представление и 5 обратно). Соответственно, выдвигаются следующие требования к системе генерации синтаксических анализаторов, на основе которой будет разработан инструмент:

1. внутреннее представление спецификации трансляции, поддерживающее большинство синтаксических конструкций сторонних генераторов анализаторов, либо возможность завести свой тип;
2. возможность описывать свои трансформации внутреннего представления;

## DMS Software reengineering toolkit

Инструмент реинжинигинга программного обеспечения DMS не является классическим генератором анализаторов, а предназначен для более узкой цели, а именно для преобразования исходного кода, возможно в другой язык, с некоторыми преобразованиями. Задача такого класса называется program transformation. Часто ставится условие, что полученный код должен быть семантически эквивалентен исходному.

Язык описания парсера — классическая форма Бэкуса-Наура. Преобразования над деревом разбора задаются декларативным или императивным путем. Первый – написание шаблона модифицируемых узлов синтаксического дерева и соответствующих изменений. Например,

transform autoinc: \X = \X + 1 -> \X++ if NoSideEffects(\X)

Второй — просто описание процедуры, применяемой для каждого узла на собственном языке инструмента — PARLANSE. Имеется готовая библиотека для анализа и оптимизации кода для популярных языков программирования.

Дерево разбора строится автоматически по грамматике, которая не может быть атрибутной. Узлы соответствуют правилам в грамматике. Соответственно, необходимо преобразовать деревья разбора от спецификаций трансляций разных генераторов анализаторов в единый тип.

## TXL

TXL изначально расшифровывался как Turing eXtender Language, то есть язык для описания расширений языка Turing, но в настоящий момент это, как и DMS, система Program transformation. Транслятор строится из спецификации парсера и трансформаций, описанных в функциональном стиле.

rule tranformThisVarRefs

replace [reference] 'this '. Id [id] Comps [repeat component]

where not Id [is\_staticclass\_var]

by 'self. Id Comps

end rule

## StrategoXT

Система сочетает в себе язык описания преобразований Stratego с инструментами для построения парсеров и структурной печати текста XT. Дерево вывода представляется с помощью аннотированных термов(ATerm) вместо простого текстового представления, что позволяет производить сложные преобразования.

desugar : While(e, stm) -> If(e, DoWhile(stm, e))

## YaccConstructor

Генератор синтаксических анализаторов, позволяющий выбирать подходящие для задачи парсер спецификации трансляции и генератор [3]. Основан на модульной архитектуре, в основу которой положено типизированное внутреннее представление грамматики. Кратко структуру приложения можно изобразить так:

Frontend

Разбор грамматики в формате 1

Main

Внутреннее представление

Generator

Печать в формат 2

или

порождение анализатора

Преобразования

Такая архитектура, помимо порождения спецификаций трансляций в форматах различных инструментов, позволяет реализовывать собственные генераторы анализаторов, предоставляя необходимую для этого инфраструктуру.

## Результаты обзора

По архитектуре и возможностям системы program transformation DMS, TXL и Stratego/XT близки друг к другу. В принципе, они подходят для решения поставленной задачи. Возможности инструментов по изящному описанию преобразований в функциональном стиле позволяют быстрее их реализовывать. С другой стороны, система, построенная вокруг популярного языка программирования общего назначения, получает преимущества в виде огромного числа библиотек и возможностей самого этого языка. Так, язык F#, на котором написана большая часть приложения YaccConstructor, предоставляет удобные возможности для работы со списками и деревьями, которые являются основными структурами данных, представляющими дерево разбора.

Также, у инструмента имеется свой язык спецификации трансляции YARD, обладающий довольно широкими возможностями, и который следует развивать дальше параллельно с развитием внутреннего представления.

Таким образом, он и был взят за основу будущего инструмента.

Сводные результаты обзора приведены в таблице:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | DMS | TXL | Stratego/XT | YaccConstructor |
| Возможности входного языка | БНФ без атрибутов, с предикатами | РБНФ, неоднозначные грамматики | Можно использовать любой фронтенд | Любой фронтенд для среды .NET |
| Представление AST | Неявно заданный тип, узлы соответствуют правилам | Неявно заданный тип | ATerm | Типизированный IL |
| Легкость преобразования | Декларативный и императивный способы | Функциональный стиль, можно менять тип узлов | Стратегии, состоящие из правил | Реализация преобразований на F# или C# |
| Лицензия | Коммерческое приложение с закрытым исх. кодом | Бесплатен, код открыт для старых версий | LGPL | GPL |
| Pretty printing | Встроенные конструкции | Автоматически | GPP | Библиотека StructuredFormat |

# Реализация

В основу инструмента положено типизированное внутреннее представление спецификации трансляции. В исходном коде оно представлено типом размеченного объединения языка F#, которое уберегает разработчика от некоторых ошибок на этапе компиляции. Оно представляет собой отражение грамматики из входного файла, выраженное в терминах языка программирования. Инструмент должен поддерживать форматы разных генераторов анализаторов, поэтому внутреннее представление должно быть максимально всеобъемлющим и поддерживать как можно большее число синтаксических конструкций. В то же время одни и те же конструкции могут иметь слегка разный смысл и поведение, поэтому необходимо четко прописать семантику внутреннего представления, чтобы оно взаимно однозначно сопоставлялось файлу со спецификацией трансляции. В ином случае, трансляция спецификаций трансляций между форматами разных инструментов может не сохранять изначально заданный язык.

## Типы узлов дерева разбора

В данный момент внутреннее представление спецификации трансляции в инструменте YaccConstructor представлено типом

module Definition = begin

type info = { fileName: string}

type t<'patt,'expr> = {

info : info;

head : 'expr option; //текст до грамматики

grammar : Rule.t<'patt,'expr> list;//грамматика

foot : 'expr option //текст после грамматики

}

end

Definition.t и некоторые другие типы, обсуждаемые дальше, параметризованы типами ‘patt и ‘expr.

‘patt – тип предиката элемента последовательности.

‘expr – тип атрибута правила. Обычно он является семантическим действием и представляется кодом на F#.

Поля head и foot содержат код, дополняющий семантические действия в атрибутах правил. Это могут быть вспомогательные функции, открытие внешних используемых библиотек. Грамматика представлена списком правил, каждое из которых имеет вид

module Rule = begin

type t<'patt,'expr> = {

name : string;

args : 'patt list; //Аргументы правила, которые можно передавать в

//семантические действия как L-атрибуты

body : Production.t<'patt,'expr>;

\_public : bool; //Является ли нетерминал стартовым

metaArgs: 'patt list //Параметры правила – их можно подставлять в тело правила как обычные нетерминалы

}

end

Более подробное описание метаправил и примеры их применения можно найти в работе [2].

Тело правила представлено типом Production.t

module Production = begin

type IRuleType = interface end

type elem<'patt,'expr> = {

omit:bool; //Если true - не включать узел в AST

rule:(t<'patt,'expr>); //Правило

binding:'patt option; //Привязка S-атрибута для использования в сем. действии

checker:'expr option //Предикат – если условие не выполнено, правило

}

and t<'patt,'expr> =

|PAlt of (t<'patt,'expr>) \* (t<'patt,'expr>)//Альтернатива

|PSeq of (elem<'patt,'expr>) list \* 'expr option //Список элементов и собственный S-атрибут

|PToken of Source.t //Токен

|PRef of Source.t \* 'expr option //Ссылка по имени на другое правило, возможно с подстановкой L-атрибутов

|PMetaRef of Source.t \* 'expr option \* 'expr list // Ссылка по имени на метаправило, возможно с подстановкой L-атрибутов и списком подставляемых правил-параметров

|PLiteral of Source.t //Литерал – описание терминала прямо в спецификации трансляции, например 'if'

|PMany of (t<'patt,'expr>) //expr\* Повторение элемента 0 или более раз

|PSome of (t<'patt,'expr>) //expr+ Повторение элемента 1 или более раз

|POpt of (t<'patt,'expr>) //expr? Повторение элемента 0 или 1 раз

end

Далее будут подробно рассмотрены особенности конструкций в плане построения дерева и вычисления атрибутов.

## Особенности внутреннего представления

### Разрешение конфликтов

Семантика IL не указывает, в какую сторону разрешать конфликты при построении AST. На данный момент это может быть решено только с помощью предикатов, так как изначально инструмент строился для GLR-анализатора, который строит все возможные деревья вывода.

### Правила с одинаковыми именами

Присутствие во внутреннем представлении правил с одинаковыми именами возможно, но нужно следить за семантикой конкретного инструмента. Могут появиться после объединения нескольких файлов спецификаций трансляции в один. Требуется их преобразовывать для приведения в соответствие семантике выходного формата. Для этого в инструменте есть преобразования LeaveLast и MergeAlter.

### Целевой язык

На данный момент полагается, что семантические действия спецификаций трансляций, с которыми работает инструмент, написаны на F#. Это ограничивает класс возможно поддерживаемых трансляторов, но не теряется возможность проводить реинжиниринг грамматики без атрибутов, предназначенные для построения AST.

### Типы возвращаемых значений синтаксических конструкций

Упрощенно процесс трансляции можно представить как вычисление атрибутов построенного AST. Вопрос заключается в том, как объединить семантические действия, привязанные к некоторым узлам, в единую работающую программу. Разработчик спецификации трансляции должен знать точно, как будет работать код, который он написал в атрибутах.

Семантические действия возвращают некоторые значения, которые преобразуются в зависимости от типа узла, их содержащего.

|  |  |
| --- | --- |
| PAlt(a,b) | a или b, в зависимости от того, какой вариант попал в дерево разбора |
| PSeq(elem\_list, Some(action\_code)) | результат action\_code |
| PSeq(elem\_list, None) | в зависимости от генератора |
| PToken(token) | значение, привязанное к токену лексером |
| PRef(rule\_name, l\_attrs) | значение из поддерева rule\_name |
| PMetaRef(metarule\_name, l\_attrs, rule\_params) | --//-- |
| PLiteral(literal) | ничего(будет ошибка, если сделать привязку к этому элементу) |
| PMany(production) | production list – список, возможно пустой, значений, возвращаемых production, длины, равной количеству разобранных элементов |
| PSome(production) | то же, что PMany, только список непустой |
| POpt(production) | Some(значения, возвращаемого production), если элемент попал в дерево разбора,  None иначе |

### Порядок вычисления атрибутов

Одна из самых сложных задач генератора синтаксических анализаторов заключается в корректном вычислении атрибутов. Необходимо предоставлять пользователю возможность писать в атрибутах не только чистые функции, но и функции с побочными эффектами. Хотя это и часто плохой стиль программирования, бывает необходимо использовать внешнюю память при работе транслятора. То, что часто функции в спецификациях трансляции не являются чистыми, подтверждается примерами реальных грамматик. Сложность для транслятора они представляют тем, что при попытке разобрать, например, неподходящую альтернативу, нужно возвращать предыдущее состояние парсера и откатывать все побочные эффекты, что представляется практически нереализуемым. Поэтому код исполняется только после полного построения AST.

В большинстве генераторов синтаксических анализаторов порядок вычисления атрибутов на построенном дереве вывода одинаков и является обратной нумерацией, начатой от стартового правила. Сыновья обходятся слева направо: для инструментов, поддерживающих L-атрибуты, такой порядок необходим, остальные просто придерживаются его в силу естественности.

Соответственно, при преобразованиях спецификации трансляции важно не только сохранять эквивалентность грамматик, но и порядок вычисления атрибутов.

## Преобразования внутреннего представления

### AddEOF

Добавляет к стартовым правилам терминал, обозначающий конец входного потока лексем EOF.

В некоторых случаях бывает удобно включить в спецификацию трансляции этот токен, когда лексер передает его парсеру наравне с остальными. Иногда бывает проще подогнать спецификацию трансляции под лексер, чем наоборот. Например, в FsYacc принято добавлять его в грамматику. В спецификациях трансляции, написанных на языке YARD, терминал конца входной последовательности не обозначают. Поэтому для трансляции из формата FsYacc в YARD удобно было бы добавить этот терминал.

Стоит сказать, что простое добавление токена EOF в конец всех стартовых правил не всегда решает задачу. Если стартовое правило рекурсивно, то есть вызывается из самого себя или какого-то другого правила, то разбор не будет произведен, потому что парсер будет ожидать EOF в середине файла. В этом случае приходится создавать вспомогательное правило.

Пример в формате YARD:

+start: start NUMBER | /\*empty\*/;

преобразуется в

+yard\_start: start EOF; start: start NUMBER | /\*empty\*/;

### ReplaceLiterals

Заменяет все вхождения литералов на токены. Позволяет задавать формат имени генерируемых токенов.

Данное преобразование полезно, когда целевой формат не поддерживает описание литералов в спецификации трансляции. В этом случае быстрее всего заменить литералы на токены и задать их в лексере.

### ExpandBrackets

Раскрывает все сгруппированные подправила из альтернатив PAlt и последовательностей PSeq. То есть, после применения преобразования не должно остаться скобок.

Такое требование к грамматике выдвигает FsYacc, у которого очень простая структура правила — альтернативы из последовательностей терминалов и нетерминалов.

Преобразование заменяет каждое подправило ссылкой на новое сгенерированное правило, которое представляет те продукции, которые были сгруппированы в исходном правиле.

### BuildAST

Заменяет семантические действия спецификации трансляции на код, строящий дерево вывода.

Применяется в основном для отладки грамматики для тех инструментов, которые без кодирования атрибутов не строят дерево вывода, а только проверяют входную строку на принадлежность языку.

Результирующее дерево имеет простой тип

type AST =  
 | Node of string \* AST list  
 | Leaf of string

, пригодный для отладки как грамматик в прикладных задачах, так и самого инструмента. Для наглядного представления полученного дерева вывода можно применять методы, печатающие его в форматы описания графов dot или gml. Примеры таких деревьев представлены в приложении «Примеры использования преобразования BuildAST».

### ExpandEnbfStrict

Преобразует грамматику так, чтобы она не содержала конструкций РБНФ.

Такого же результата можно было добиться, используя преобразования ExpandEBNF и ExpandMeta, описанные в работе [2], но преимущество этого метода в том, что он создает меньше новых правил, и полученная грамматика легче читаема и более удобна для отладки.

## Модульность спецификации трансляции

Возможность задавать спецификацию трансляции в нескольких файлах является способом разграничения уровней абстракции, структурирования кода, позволяет повторно использовать модули в различных системах. Современные системы ввиду своей масштабности и сложности требуют этого и от исходных кодов генераторов синтаксических анализаторов. Однако такие способы организации кода как процедурное и объектно-ориентированное программирование не полностью применимы к коду спецификаций трансляций.

Представим примитивную ситуацию: мы хотим добавить в грамматику, описывающую сумму ряда чисел, действие умножения. Нужно расширить грамматику

+start: NUMBER (PLUS NUMBER)\* ;

до грамматики

+start: mult\_part (PLUS mult\_part)\* ;

mult\_part: NUMBER (STAR NUMBER)\* ;

Как видно, требуется способ заменить в первом правиле токены NUMBER на нетерминалы mult\_part, и добавить новое правило mult\_part. Для реальных случаев наподобие этого нужен достаточно сложный язык описания расширений правил. Один из способов это сделать и соответствующий язык описаны в работах [4] и [5].

В простых случаях может быть достаточно просто возможности заменять одно правило другим, с тем же именем. Этот подход, например, реализован в инструменте ANTLR [14]. Другой ­— если встречается правило с таким же именем, то продукция второго правила добавляется к первому в качестве альтернативы. В теории контекстно-свободных языков такой способ естественно получается из определения контекстно-свободной грамматики. Правила определены множеством, и в теории порядок не имеет значения, хотя на практике иногда важен. Такой подход тоже встречается в генераторах синтаксических анализаторов, например в Menhir.

Инструмент YaccConstructor позволяет задавать спецификацию трансляции в нескольких файлах, в том числе различных форматов, и выбирать, какой из двух выше описанных способов применять при их объединении. При этом с каждым файлом работает парсер соответствующего формата, а объединение происходит уже внутренних представлений. Выбор способа разрешения конфликтов правил с одинаковыми именами реализован в соответствующих преобразованиях внутреннего представления.

### LeaveLast

Оставляет последнее по порядку правило из правил с одинаковыми именами.

Дает возможность переопределения правил из базового модуля новыми правилами. Таким способом, можно было бы расширить грамматику действием умножения в рассмотренном выше примере. Но данный способ неустойчив к изменениям грамматики в базовом файле.

### MergeAlter

Объединяет правила с одинаковыми именами знаком альтернативы.

Может быть полезно во многих случаях, но тоже неустойчиво к изменениям грамматики в базовом файле.

## Парсеры спецификаций трансляции

Парсеры спецификаций трансляции, так называемые фронтенды, преобразуют исходный код спецификации трансляции, заданный в формате некоторого инструмента, во внутреннее представление YaccConstructor. При этом получившееся внутреннее представление должно быть максимально подобно исходному коду. Работу по его трансформации для приведения в формат другого инструмента выполняют уже преобразования внутреннего представления. Далее описываются реализованные в рамках работы модули.

### AntlrFrontend

Парсер грамматики в формате ANTLR.

Язык F# не поддерживается инструментом ANTLR, поэтому на данный момент фронтенд можно использовать только для реинжиниринга грамматик без атрибутов. Решение поддерживать формат было принято ввиду популярности инструмента и большого числа написанных спецификаций трансляций для него.

Реализован компонент с помощью генератора синтаксических анализаторов FsYacc в связке с лексером FsLex. Особенностью стало то, что в ANTLR спецификация лексера задается в том же файле, что и парсера, поэтому во внутреннее представление в комментарий в заголовке выносятся все лексемы, которые необходимо описать в лексере целевого инструмента.

Фронтенд был протестирован на грамматиках языка C, CSS 2.1, URL.

### FsYaccFrontend

Парсер спецификации трансляции в формате FsYacc.

FsYacc — реализация классического генератора Yacc для языка F#. Как и Yacc, имеет очень простой язык. По большей части повторяет возможности ocamlyacc до такой степени, что в FsYacc можно использовать спецификации трансляции, написанные для ocamlyacc. Как правило, в качестве лексера к нему используется FsLex.

В YaccConstuctor парсер FsYacc написан на языке Yard, который транслируется в FsYacc с помощью самого инструмента, а именно фронтенда YardFrontend, генератора FsYaccPrinter и необходимых преобразований.

## Генераторы спецификаций трансляции

Задача генератора спецификации трансляции заключается в текстовом выводе внутреннего представления в формат некоторого инструмента. Для этого подаваемое на вход генератора внутреннее представление должно удовлетворять ограничениям конкретного формата, например, не должно быть метаправил и возможно конструкций РБНФ. Общая для всех генераторов проблема форматирования печати решается с помощью соответствующей библиотеки Microsoft.FSharp.Text.StructuredFormat, входящей в FSharp.PowerPack. Она обладает достаточными для задачи возможностями по выводу списков, управлению отступами, заданию максимальной длины строки и другими, что существенно облегчает работу.

### FsYaccPrinter

Генератор спецификации трансляции из внутреннего представления в формат FsYacc.

Так как формат FsYacc поддерживает минимум конструкций, перед использованием данного генератора необходимо применить к внутреннему представлению ряд преобразований.

**Применение**

Благодаря гибкой модульной архитектуре и универсальности внутреннего представления можно найти различные способы применения инструмента.

* Реинжиниринг спецификации трансляции, то есть однократное ее кардинальное улучшение в рамках текущей задачи. Это может быть ее трансляция в формат более подходящего инструмента или исследование и автоматизированная модификация спецификации в том же формате. Применение инструмента позволяет решить проблему неожиданно большого числа конфликтов в грамматике для LALR(1)-анализатора путем перевода ее на GLR-анализатор, позволяющий работать с неоднозначными грамматиками. Таким же образом могут быть решены и другие непредвиденные проблемы, связанные с выбранным генератором анализаторов.
* Использование грамматики из открытого источника вместо написания собственной. Инструмент облегчает ее интеграцию в проект, предоставляя возможность транслировать ее в требуемый формат, более удобно исследовать и поддерживать в дальнейшем.
* Использование собственного генератора синтаксических анализаторов как модуля системы вместе с одним из фронтендов.
* Использование выбранного генератора анализаторов, но разработка спецификации трансляции на другом, более богатом языке с постоянной автоматической трансляцией кода в его формат.

Реальные проекты, в которых инструмент был применен, описаны ниже.

### SqlMigration

В пилотном проекте по автоматической трансляции кода из T-SQL в PL/SQL YaccConstuctor был выбран в качестве приложения для разработки парсера T-SQL. Грамматика разрабатывается на языке YARD и автоматически транслируется инструментом в формат FsYacc, которым и строится транслятор. При таком подходе благодаря применению конструкций YARD, которые не поддерживаются FsYacc, в особенности конструкций РБНФ и внутренних альтернатив, удается в несколько раз сократить грамматику и этим упростить разработку. На текущий момент исходный файл содержит 200 строк, а сгенерированный — 800.

Приложение использовалось в демонстрационных целях, трансляция производилась небольшого подмножества конструкций T-SQL, и поэтому не возникло неудобств с использованием дерева разбора, строящимся атрибутами, генерируемыми преобразованием BuildAST. Это освободило разработчика от необходимости писать их вручную.

Также для структурирования кода применялась возможность описывать грамматику в нескольких модулях.

При дальнейшем развитии проекта планируется использовать грамматику, написанную на YARD для демонстрационного проекта. Дополнительный плюс к выбору YARD-а в качестве языка, на котором будет разрабатываться парсер — минимизация последствий срабатывания риска. То есть, если в процессе разработки грамматики становится сложно вручную разрешать конфликты в классе LALR(1), или даже окажется, что язык SQL шире, чем LALR(1), то можно будет использовать более мощный генератор, например, GLR или основанный на парсер-комбинаторах, нужно будет лишь реализовать генератор спецификации трансляции в требуемый формат.

### Claret

Claret [6] — система статического анализа языка C. Планируется применение YaccConstructor для разработки парсера на основе грамматики, взятой из открытого источника и автоматизированной ее модификации. К сожалению, на данный момент инструмент полностью применить не получилось, так как он требует некоторых доработок, в частности, привязку к токену его типа и автоматический поиск правил, которые могут быть объединены конструкциями РБНФ.

# Заключение

В ходе выполнения данной дипломной работы были получены следующие результаты.

1. Обоснован выбор базовой технологии для построения инструмента реинжиниринга спецификаций трансляций.
2. Реализована возможность чтения грамматики в формате инструмента ANTLR, FsYacc печать в форматы YARD, FsYacc. Реализованы необходимые для этого трансформации: раскрытие внутренних подправил, добавление токена конца файла к стартовому правилу, замена литералов токенами, раскрытие конструкций РБНФ.
3. Для удобства разработки грамматики реализована простая поддержка возможности задания спецификации трансляции в нескольких файлах, выбор, объединять в альтернативу или оставлять последнее из правил с одинаковыми именами. Также, реализовано преобразование, генерирующее семантические действия для построения AST.
4. Инструмент успешно применен в проекте SqlMigration.

Дальнейшее развитие проекта может включать в себя поддержку форматов других генераторов синтаксических анализаторов и исследование способов совместить их. Другим направлением развития может быть развитие собственного языка спецификации трансляций, а именно конструкции расширенных регулярных выражений, перестановок, развитие поддержки модульной структуры грамматики, например, на основе работы [5]. Будет продолжаться работа над GLR-генератором [7]. Также, на основе проекта представляет интерес разработать дополнение к среде разработки Visual Studio, позволяющее более удобно разрабатывать транслятор и интегрировать его в основное приложение.

Исходные коды проекта, описание и документацию можно найти на сайте [8].

# Примеры использования преобразования BuildAST

|  |  |
| --- | --- |
|  | Дерево вывода для грамматики  +s: e;  e: NUMBER | NUMBER PLUS e ; |
|  | Дерево вывода для грамматики  +s: NUMBER (PLUS NUMBER)\* ;  Эта грамматика задает тот же язык, но записана короче и дерево вывода содержит меньше вспомогательных узлов. Такое дерево, отображающее конструкции РБНФ, возможно получить и генератором, их не поддерживающим. Это задается порядком применения преобразований. |

# Список использованных источников

1. **Чемоданов, И.С. и Дубчук, Н.П.** Обзор современных средств автоматизации создания синтаксических анализаторов. *Системное программирование. СПб.: Изд-во С.-Петерб. ун-та.* 2006 г., стр. 286-316.

2. **Чемоданов, Илья.** *Генератор синтаксических анализаторов для решения задач автоматизированного реинжиниринга программ.* 2007.

3. **Улитин, К.А.** *Разработка архитектуры для генератора синтаксических анализаторов.* 2010.

4. **Бреслав, А.А.** *Применение принципов MDD и аспектно-ориентированного программирования к разработке ПО, связанного с формальными грамматиками.*

5. —. *Средства повторного использования формальных грамматик и их применение для создания диалектов.*

6. Claret home page. [В Интернете] http://code.google.com/p/claret/.

7. **Григорьев, С.В.** *Генератор синтаксических анализаторов для неоднозначных контекстно-свободных грамматик.* 2010.

8. YaccConstructor home page. [В Интернете] http://code.google.com/p/recursive-ascent/.

9. **Ахо, А., и др., и др.** *Компиляторы: принципы, технологии и инструментарий.* 2008.

10. **Богданов, В.Л. и Гордеев, В.С.** Практический опыт написания синтаксического анализатора языка программирования Кобол. [авт. книги] Под ред. проф. А.Н. Терехова и А.А. Терехова. *Автоматизированный реинжиниринг программ.* 2000, стр. 64-75.

11. **Syme, Don.** *Expert F#.* 2007.

12. **Clint, Paul, Lammel, Ralf и Verhoef, Chris.** *Toward an engineering discipline for GRAMMARWARE.*

13. **Мартыненко, Б.К.** *Языки и трансляции.* 2004.

14. *Репизиторий, документация, примеры грамматик ANTLR.* [В Интернете] http://antlr.org.

15. *Спецификация FsYacc.* [В Интернете] http://fsharppowerpack.codeplex.com/wikipage?title=FsYacc.

16. *Current Parsing Techniques in Software Renovation Considered Harmful.* **Brand, Mark van den, Sellink, Alex и Verhoef, Chris.** IWPC '98.

17. *TXL Home Page.* [В Интернете] http://www.txl.ca/.

18. *Stratego Program Transformation Language.* [В Интернете] http://strategoxt.org/.

19. *DMS Software Reengineering Toolkit.* [В Интернете] http://www.semanticdesigns.com/Products/DMS/DMSToolkit.html.

1. Например, страница <http://en.wikipedia.org/wiki/Comparison_of_parser_generators> приводит около 150. [↑](#footnote-ref-1)