САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра системного программирования

Калмыков Алексей Владимирович

Построение ослабленного GLR транслятора на основе анализа грамматики на избыточность

Бакалаврская работа

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

Научный руководитель:
аспирант кафедры системного программирования Григорьев С.В.

Рецензент:
ст. преп. Кириленко. Я.А.

Санкт-Петербург
2013

SAINT-PETERSBURG STATE UNIVERSITY
Mathematics & Mechanics Faculty

Chair of Software Engineering

Aleksei Kalmykov

Construction of relaxed GLR translator based on grammar redundancy analysis

Bachelor's Thesis

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

Scientific supervisor:
postgraduate student of chair of software engineering S.V. Grigoriev

Reviewer:
senior lecturer I.A. Kirilenko

Saint-Petersburg
2013

Оглавление

1. Введение

Среди задач реинжиниринга одна из самых частых - перевод исходного кода системы на другой язык программирования. Поскольку исходные коды могут насчитывать миллионы строк, данный процесс автоматизируется при помощи построения специальных трансляторов. Для создания транслятора обычно требуется описание синтаксиса и семантики обоих языков. Но в реинжиниринге часто можно встретить следующую ситуацию: корректно компилирующийся и работающий код не корректен с точки зрения стандартной грамматики. Это происходит из-за использования разработчиками недокументированных возможностей компиляторов. Например, в оригинальном компиляторе некоторая синтаксическая конструкция была обязательной, однако в последствии разработчики модифицировали компилятор так, что она стала необязательной. Это приводит к тому, что код разработчиков становится некорректен с точки зрения оригинального компилятора. Зачастую, не существует никакой возможности узнать информацию об этих модификациях от разработчиков или из документации.

Кроме того, для многих старых языков часто создавались самодельные расширения и диалекты, которые не документировались или документация которых была утрачена. Проведение ручного анализа и изменение грамматики является очень трудоемкой и дорогой операцией. В связи с этим возникает необходимость по грамматике и корректной программе проводить анализ на наличие недокументированных возможностей, а так же автоматически модифицировать существующий транслятор. В результате можно получить транслятор, который принимает исходный язык и некоторое его расширение, которые позволят производить корректную трансляцию из исходного языка в целевой.

Так же встречаются случаи, когда исходный код программы состоит из нескольких различных диалектов языка, у которых есть существенные различия (обязательность/ необязательность некоторых ключевых слов), которые вызывают конфликты. Такие ситуации тяжело проанализировать вручную, учитывая тот факт, что иногда информация о

реально используемом диалекте оказывается недоступной, не говоря о том, насколько трудоемко ручное исправление таких ошибок и правка синтаксического анализатора. Существует возможность того, что исходный код программы может быть вообще принципиально некорректным с точки зрения синтаксиса. Это может быть вызвано рядом причин, например тем, что данный код компилировался со специальными ключами, в так называемом специальном режиме, в котором допускается наличие ошибок, однако компилятор эти ошибки игнорирует.

Возможности по исправлению ошибок могут быть полезны не только для анализа старых языков. С одной стороны, в современных языках было бы полезно выявлять необязательные синтаксические конструкции (например в языке C необязательна открывающая скобка в условии оператора “if”). Это может быть полезным как для реализации системы автодополнения в средах разработки программирования (автоматическая подстановка символов, если известно, что эти символы должны быть там однозначно) так и для автоматического исправления ошибок (добавление пропущенных скобок, разделителей ‘;’) [3][4]. С другой стороны, это может оказаться полезным при разработке новых языков. Путём разумного удаления необязательных конструкций можно существенно упростить синтаксис языка и сделать его более удобным и понятным для программиста.

Проект YaccConstructor[1] — модульный программный продукт [8], предназначенный для разработки трансляторов для нужд реинжиниринга. В процессе использования данного проекта возникла необходимость создания ослабленных анализаторов. В проекте YaccConstructor существует несколько видов генераторов анализаторов. В данной дипломной работе будет рассматриваться генератор RNGLR анализаторов [7].

RNGLR анализатор (англ. Right-nulled GLR analyzator) [5] — одна из модификаций GLR[2] алгоритма анализа. Его ключевое отличие от

[1]Проект YaccContructor - http://code.google.com/p/recursive-ascent . Дата обращения: 31.05.2013
[2]GLR Parser - http://en.wikipedia.org/wiki/GLR_parser. Дата обращения: 31.05.2013

LALR[3] анализаторов заключается в том, что таблицы GLR анализатора допускают множество результирующих состояний, тогда как таблицы LALR позволяют лишь один переход из состояния. Это позволяет GLR анализаторам разрешать неоднозначности типа сдвиг/свертка и свертка/свертка[4].

Целью данной дипломной работы является реализация алгоритма построения ослабленного анализатора, его последующая интеграция в проект YaccConstructor и его апробация на примерах на языке T-SQL[5].

[3]LALR Parser - http://en.wikipedia.org/wiki/LALR_parser . Дата обращения: 31.05.2013

[4]Shift/reduce and reduce/reduce conflicts - http://140.120.7.20/OpenSystem2/SoftwareTools/node136.html . Дата обращения - 31.05.2013

[5]Язык Transact-SQL - http://en.wikipedia.org/wiki/Transact-SQL . Дата обращения: 31.05.2013

2. Постановка задачи

Искомый алгоритм должен обладать следующими свойствами. Во-первых, алгоритм не должен менять результаты анализа корректных цепочек. Во-вторых, для любой некорректной цепочки, которая может быть принята ослабленным анализатором, результат ее анализа должен совпадать с результатом анализа некоторой корректной цепочки. В-третьих, алгоритм не должен требовать вмешательство пользователя для достижения результата, то есть: не требовать внесения изменений в исходыный код, который предполагается анализировать, не должен требовать никаких модификаций и преобразований грамматики исходного языка и не требовать вмешательства пользователя на всех этапах работы алгоритма. Анализатор, в котором используется искомый алгоритм, будем в дальнейшем называть ослабленным.

Таким образом, сформулируем цели данной дипломной работы:

1. Разработка алгоритма построения ослабленного RNGLR анализатора на основе анализа грамматики на избыточность, удовлетворяющем следующим требованиям:
    1.1. Сохранение неизменными результатов анализа корректных цепочек.
    1.2. Для любой некорректной цепочки, которую сможет принять ослабленный анализатор, результат ее анализа должен совпадать с результатом анализа некоторой корректной цепочки.
    1.3. Сохранение неизменной входной грамматики.
    1.4. Алгоритм должен автоматически модифицировать существующий анализатор, так, чтобы он стал ослабленным.
2. Внедрение данного алгоритма в проект YaccConstructor.
3. Проведение тестировнаия алгоритма.

Структура дипломной работы. В главе 3 дан обзор существущих алгоритмов коррекции ошибок и построения ослабленных анализато-

ров, в главе 4 дается описание реализации алгоритма построения ослабленного анализатора и описание интеграции этого алгоритма в проект YaccConstructor.

3. Обзор существующих решений

3.1. Возможные методы решения

Задача построения ослабленного анализатора сводится к задаче обработки некорректных цепочек. Существует 3 основных способоа решения данной задачи:

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

Обнаружение ошибок позволит анализатора показать, что во входной строке есть ошибка, однако не позволяет эту ошибку обработать. Однако в условиях поставленной задачи, это является недостаточным.

Восстановление после ошибок предоставляет возможность не только находить ошибки, но и получать некоторый диагностический вывод об этой ошибке, например, сообщать пользователю об ожидаемом символе если он был пропущен. Однако данный метод может привести к серььезным изменениям внутеннего состояния транслятора, а это, в свою очередь, может привести к выполнению некоторых семантических действий связанных с соответствующим правилом грамматики в таком порядке, который невозможен при корректном входе.

Наиболее подходящим методом является исправление ошибок. Под исправлением ошибок подразумевается набор методов, в которых входная строка преобразуется так, что становится синтаксически корректной. При этом внутреннее состояние анализатора не изменяется. Поскольку от искомого решения требуется, что для любой некорректной цепочки, принимаемой ослабленным анализатором, результат будет совпадать с результатом анализа некоторой корректной цепочки, данный метод является наиболее подходящим.

Рассмотрим подробнее основные классы методов исправления ошибок [2].

3.1.1. Локальная коррекция

Сначала, алгоритм ищет ошибку. Как только она находится, вычисляется специальное множество acceptable-set[6], в которое входят допустимые символы для текущего состояния. После этого, алгоритм пропускает оставшиеся символы во входной строке до тех пор, пока не найдется символ из множества acceptable-set. Все методы данного класса алгоритмов отличаются способом вычисления множества acceptable-set и тем, как они изменяют состояние анализатора.

Основным преимуществом этого класса алгоритмов является быстродействие и автоматизация. Недостатком является то, что алгоритм не являтся эффективным, поскольку ищет самый первый символ из множества acceptable-set. В случае с GLR анализаторами, данный алгоритм будет порождать очень большое количество неверных выводов, что приведет к значительной потере производительности.

3.1.2. Глобальная коррекция

Основное отличие данного метода от метода локальной коррекции заключается в том, что после того, как алгоритм нашел ошибку, он ищет не первый символ во входной строке, который входит во множество acceptable-set, а наиболее подходящий. Для этого алгоритм анализирует текущий контекст, и на основе него пропускает символы во входной строке, до тех пор, пока не встретится наиболее подходящий.

Преимуществом данного алгоритма является высокая эффективность, поскольку алгоритм выбирает наиболее подходящий символ, а не первый попавшийся. К недостаткам относится низкая производительность, поскольку вычисление наиболее подходящего символа по контексту является затратной операцией. Подробнее рассматривается в работе [1].

[6]Acceptable-set - http://www.cs.vassar.edu/čs331/accepted-sets.html . Дата обращения - 31.05.2013

3.1.3. Специальные методы

Специальными методами называется такой класс алгоритмов, которые приходится дополнительно подстраивать под каждую решаемую задачу, то есть они не являются автоматизированными. Однако, как можно будет заметить далее, небольшая модификация алгоритма может привести к практически полной автоматизации.

Рассмотрим подробнее метод пустых ячеек.

3.2. Метод пустых ячеек

В таблице анализатора содержатся пустые ячейки. Если анализатор обращается к таким ячейкам, это означает, что произошла ошибка. В данном методе, пустым ячейкам сопоставляются специальные методы обработки ошибок. В общем случае, для каждой ячейки надо сопоставлять специальную процедуру, которую должен описывать программист. Это очень трудозаратно и требует от программиста большого внимания.

Однако, в условиях поставленной задачи,существует возможность автоматизировать данный метод. Такой результат был получен в дипломной работе Андрея Ефимова 2008 года [6].

Рассмотрим результаты данной работы подробнее.

3.2.1. Построение ослабленного LALR транслятора

В алгоритме, предложенном в дипломной работе Андрея Ефимова был предложен алгоритм построения ослабленного LALR анализатора на основе метода пустых ячеек. Идея алгоритма формулируется следующим образом:

Если в некотором состоянии i анализатор принимает только символ t, либо символ t и символ конца строки, и переходит в состояние j, то все пустые ячейки этого состояния могут быть заменены на push t; go to j. При этом все пустые ячейки в состояниях, в которых определена свертка для символа t, должны быть заменены на такие же свертки.

Данный алгоритм не модифицирует грамматику, не требует вмешательства программиста, не зависит от входной грамматики и не модифицирует входную строку.

Однако, данный алгоритм был реализован только для LALR анализаторов. В данной дипломной работе предлагается его модификация для работы с RNGLR анализаторами.

4. Описание решения

4.1. Особенности реализации алгоритма

Реализацию алгоритма автоматической модификации GLR анализатора на основе анализа грамматики на избыточность можно разделить на 2 этапа:

1. модификация таблицы анализатора;
2. модификация генератора RNGLR парсера.

Рассмотрим каждый из этих этапов более подробно.

4.1.1. Модификация таблицы анализатора

Сначала требуется произвести автоматическую модификацию таблицы анализатора. Поскольку в данной дипломной работе производится лишь модификация, а не создание анализатора, то на входе у нас есть готовая таблица анализатора. В отличии от LALR таблиц, в GLR таблицах возможно наличее более одного действия в одной ячейке таблицы анализатора, причем в различных коминациях (то есть любое количество опреаций shift и/или любое количество опреаций reduce). Таким образом получается, что алгоритм Ефимова нельзя применить к GLR анализаторам, не внеся каких-либо изменений.

В оригинальном алгоритме говорится, что если в состоянии i анализатор принимает только символ t и переходит в состояние j, то все пустые ячейки этого состояния заменяются на push t; goto j. Пустые ячейки для сверток заполняются сверткой, если она единственна в данном состоянии. Возникает вопрос: если в состоянии t есть единственная операция shift, и остальные ячейки пустые, и в этом же состоянии оперелены некоторое количество сверток, то неизвестно как обрабатывать данную ситуацию. В ходе данной дипломной работы было выяснено, что оптимальным решением в плане производительности является: независимо обрабатывать операции свертки и переносов и выполнять

их параллельно, если в одной ячейке анализатора присутствуют обе операции.

4.1.2. Модификация генератора RNGLR парсера

Для реализации алгоритма требутеся произвести небольшую модификацию RNGLR парсера. В алгоритме вводится новое действие *push t, goto j*, которое не является стандартным для RNGLR парсера. Данное действие называется *проталкиванием символа t*, и оно означает добавление на стек символа t и переход в состояние j. Метод перехода в новое состояние не требуется реализовывать, так как он уже реализован в парсере. Добавление символа на стек является тривиальным действием.

4.2. Интеграция в YaccConstructor

В проекте YaccConstructor представлены два модуля, которые модифицировались в процессе данной дипломной работы - *RNGLRGenerator* и *RNGLRParser*.

Модуль *RNGLRGenerator* отвечает за генерацию таблицы анализатора. Основным файлом модуля является файл *RNGLRGenerator.fs*. В нем происходит сбор всей небходимой информации и из него запускается генерация всех необходимых данных для создании таблицы анализатора. На вход данному модулю идет набор параметров, задаваемых из коммандной строки, необходимых для создания таблицы анализатора. Для создания ослабленного анализатора была добавлена опция коммандной строки "-attended=true/false", которая отвечает за построение ослабленного анализатора. Никакой дополнительной информации пользователю задавать не нужно. Описание самой таблицы анализатора просходит в файле *Tables.fs* в типе *Tables*. Сама таблица анализатора разбита на 2 части: на таблицу gotos, отвечающую за shift, и таблицу reduces, отвечающую за reduce, заполняемые числами. Поскольку данный тип не предоставляет возможности записать в таблицу команду *push t; goto j;* был создан специальный тип RelaxedTables, описываю-

щий таблицу ослабленного анализатора. Состояния, которые удовлетворяют условию алгоритма вычисляются и записываются в массивы attendedPushes и attendedReduces для записи команд *push t; goto j* и *reduce i* соответственно.

Модуль *RNGLRParser* отвечает за автоматическую генерацию RNGLR парсера. В его конфигурацию задается булевый флаг, отвечающий за то, является ли анализатор ослабленным или нет. Если при анализе входной строки в состоянии t встречается ошибка и при этом известно, что транслятор ослабленный, то:

1. Если хотя бы одно из значений attendedPushes[t] и attendedReduces[t] не пусто, то:

    1.1. Если не пусто значение только attendedPushes[t], произвести действие *push i;goto j*, где пара *(i,j)* это значение attendedPushes[t]

    1.2. Если не пусто значение только attendedReduces[t], произвести действие *reduce i*, где *i* это значение attendedReduces[t]

    1.3. Если оба значение не пусто, выполнить оба действия, разветвив действия анализатора

2. Если оба значения пусты, сообщить об ошибке

Таким образом связка из модулей *RNGLRGenerator* и *RNGLRParser* предоставляет возможность автоматической генерации ослабленного анализатора, отвечающем требованиям, представленным в Главе 2.

5. Заключение

5.1. Результаты

В ходе данной дипломной работы было изучена работа Андрея Ефимова и проведен обзор других существующих решений для решения задачи построения ослабленного анализатора для GLR анализаторов.

Основными результатами данной дипломной работы являются:

- результаты работы Андрея Ефимова использованы в контексте GLR анализаторов;
- реализован алгоритм автоматической модификации ослабленного GLR анализатора на основе анализа грамматики на избыточность;
- данный алгоритм интегрирован в модульный программный продукт для нужд реинжиниринга YaccConstructor;
- проведено тестирование алгоритма

5.2. Исходные коды

Исходные коды алгоритм доступны в проекте YaccConstructor , в git-ветке Relaxed_RNGLR

Список литературы

[1] Bruke M., G.Fisher. A practial method for LR and LL syntactic error diagnosis and recovery. — ACM Transactions on Programming Languages and Systems, April 1987. — URL: http://dl.acm.org/citation.cfm?id=22720.

[2] Grosch J. Efficient and Comfortable Error Recovery in Recursive Descent Parsers. — Structured Programming, 1989. — URL: ftp://www.cocolab.com/products/cocktail/doc.pdf/ell.pdf.

[3] Lennart K. Providing Rapid Feedback in Generated Modular Language Environments. — Proceedings of the 24th International Conferemce on Object-Oriented Programming, systems, Languages and Applications, 2009. — URL: http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2009-020.pdf.

[4] Lennart K. Natural and Flexible Error Recovery for Generated Modular Language Environments. — ACM Transactions on Programming Languages and Systems, 2012. — URL: http://dl.acm.org/citation.cfm?id=2400678.

[5] Scott E., Johnstone A. Right Nulled GLR Parser. — ACM Transactions on Programming Languages and Systems, July 2006. — URL: http://people.via.ecp.fr/~stilgar/doc/compilo/parser/Right%20Nulled%20GLR%20Parsers.pdf.

[6] А. Ефимов. Построение ослабленного LALR-транслятора на основе анализа грамматики на избыточность. — Дипломная работа кафедры Системного программирования Математико-Механического факультета, 2008. — URL: http://se.math.spbu.ru/SE/diploma/2008/Efimov_dip.pdf.

[7] Д. Авдюхин. Создание генератора GLR трансляторов для .NET. — Курсовая работа кафедры Системного программирования Математико-Механического факультета, 2012. —

URL: http://se.math.spbu.ru/SE/YearlyProjects/2012/YearlyProjects/2012/445/445_Avdyukhin_report.pdf.

[8] К. Улитин. Разработка архитектуры для генератора синтаксических анализаторов. — Курсовая работа кафедры Системного программирования Математико-Механического факультета СПбГУ, 2010. — URL: http://se.math.spbu.ru/SE/YearlyProjects/2010/YearlyProjects/2010/445/Ulitin_report.pdf.