САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра системного программирования

Иванов Андрей Васильевич

Восстановление после ошибок в GLR-алгоритме

Курсовая работа

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

Санкт-Петербург
2013

Оглавление

Введение

При автоматизированном реинжиниринге программного обеспечения возникает задача синтаксического анализа различных языков программирования. При этом частой является ситуация, когда у разработчиков нет подходящего инструмента, который мог бы произвести этот анализ.

В связи с этим широкое распространение получили генераторы синтаксических анализаторов – инструменты, которые принимают на вход грамматику, описанную в определенной форме, и порождают на ее основе соответствующий ей парсер. Одна из особенностей разработки таких синтаксических анализаторов для задач реинжиниринга заключается в том, что документация отсутствует совсем или же содержит в себе неточности. Итеративность процесса разработки синтаксических анализаторов является другой особенностью. Как правило, при реинжиниринге не обязательно разбирать весь входной язык, а достаточно поддержать его подмножество, которое используется в обрабатываемой системе.

Таким образом, грамматика часто оказывается неполной, и на этапе синтаксического анализа случаются ошибки. После их обнаружения следует предпринять некие действия, чтобы продолжить разбор, поскольку это позволит выявить другие ошибки, а также провести дальнейший анализ разобранной части кода.

Наиболее мощным классом синтаксических анализаторов являются GLR-парсеры (Generalized LR)[2]. Их преимущество над LL(k) и LALR(k) алгоритмами заключается в том, что они рассматривают все возможные пути разбора. За счет этого GLR-парсеры могут работать с неоднозначными грамматиками и возвращать все возможные деревья разбора. Однако возникают сложности при восстановлении, потому что становится сложно определить, по каким ветвям стоит продолжить разбор. Например, если начать восстанавливать все ветви, которые «умерли», то может быть получено много лишних результатов и лишних сообщений об ошибках.

В рамках данной работы поставлена цель исследовать существующие подходы к восстановлению и реализовать восстановление после ошибок в GLR-парсере.

1. Постановка задачи

В рамках данной работы были поставлены следующие задачи:

- Исследовать возможные подходы к восстановлению после ошибок.
- Провести анализ алгоритмов, оценить их применимость к GLR, выбрать наиболее подходящий для практической реализации.
- Реализовать восстановление после ошибок в GLR-генераторе на основе выбранного алгоритма. Полученный инструмент должен уметь строить дерево разбора исходного кода, позволяющее продолжить обработку даже в случае наличия ошибок.
- Провести тестирование полученного инструмента.

2. Обзор существующих подходов

Существует ряд различных подходов к восстановлению после ошибок в синтаксическом анализе. В ходе работы рассмотрены следующие подходы.

В стратегии Panic mode [1] при описании грамматики программистом явным способом задается синхронизирующее множество. При возникновении ошибки ищется элемент этого множества, затем анализатор должен перейти в состояние, в котором принимается найденный символ. В случае GLR из стека постепенно удаляются состояния, пока на вершине стека не окажется нужное. Обычно синхронизирующее множество состоит из символов, завершающих конструкцию языка. Например, для языка Pascal - точка с запятой (;), ключевое слово end. Недостаток метода заключается в том, что не в каждом языке можно найти такие разделители. Примером может служить язык SQL.

В стратегии Error Token [5] в случае возникновения ошибки генерируется специально зарезервированный токен ошибки (error token). Он вставляется перед символом, на котором случилась ошибка, в рассматриваемый вход. Если в грамматике содержатся правила, распознающие этот токен, то разбор продолжается следующим образом. Со стека снимаются состояния до тех пор, пока error token не станет достижим. После того, как такое состояние найдено, происходит сдвиг. Следом начинается поиск символа, который в соответствующем правиле идет за error token'ом. Когда такой символ находится, происходит свертка или сдвиг и разбор продолжается.

В стратегии Follow-set [1] никаких изменений грамматики не требуется. Когда анализатор встречает ошибку, он вычисляет по своему состоянию некоторое множество допустимых в этом состоянии символов (acceptable-set). Затем пропускаются все символы входной цепочки, пока не встретится символ из этого множества. Потом состояние анализатора изменяется таким образом, чтобы первый не пропущенный символ стал допустимым.

В стратегии locally-least error recovery [1] при обнаружении ошибки процесс восстановления происходит за счет редактирования входного потока. Во входную строку могут добавиться терминальные и нетерминальные символы, после чего вход может стать корректным. Также токен, на котором случилась ошибка, может быть удалён или быть заменен на какой-то другой символ. При этом каждое действие с любым символом имеет конечную стоимость, которая определяется программистом, который пишет грамматику. Выбирается вариант корректировки, который имеет наименьшую суммарную стоимость, после чего парсер продолжает свою работу.

Стратегия Suffix grammar [1] основывается на следующей идее, что множество всех суффиксов любого языка можно описать грамматикой, которая называется суффиксной. В случае возникновения ошибки генерируется суффиксная грамматика и происходит сдвиг символа. Оставшийся входной поток является или суффиксом исходного

языка или же содержит другую ошибку, которая будет обрабатываться таким же образом. Но недостаток данного метода в том, что суффиксная грамматика получается очень большой.

В качестве реализуемого алгоритма выбран подход, использующий в своей работе error token. Его преимущество над остальными подходами в том, что error token позволяет сохранить семантику, что позволяет проводить дальнейшую обработку дерева, содержащего ошибки, и разбирается как обычный токен. Также этот алгоритм позволяет влиять на процесс восстановления через явное изменение грамматики.

3. Реализация

3.1. YaccConstructor

На кафедре Системного Программирования СПбГУ ведется разработка инструмента YaccConstructor [6], который позволяет создавать синтаксические анализаторы для платформы .NET. Основным языком разработки проекта является язык F# [3]. В рамках проекта был создан RNGLR-модуль [4], который является генератором GLR-парсеров. Описываемый в данной работе алгоритм восстановления после ошибок реализован как часть этого модуля.

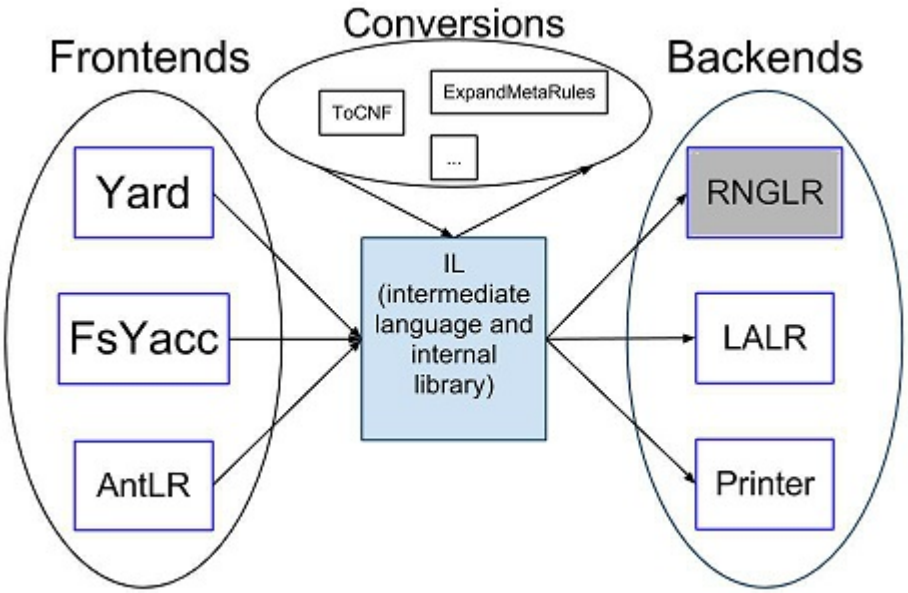

Рис. 1: Архитектура YaccConstructor

3.2. Граф разбора

Поскольку реализация алгоритма тесно связана с графом разбора, используемом в RNGLR-модуле, то рассмотрим структуру этого графа более подробно.

Каждая вершина имеет три поля:

- Состояние - стандартное состояние парсера при LALR(1)-анализе.
- Уровень – номер токена, который рассматривался в момент, когда вершина создавалась.
- Исходящие ребра.

Каждое ребро имеет два поля:

- Дерево. Рёбра создаются в результате сдвига или свёртки. При этом полученное в результате одного из этих действий дерево ассоциируется с ребром.
- Вершина, в которую ребро направлено.

3.3. Алгоритм

Алгоритм восстановления начинает работу в одном из двух случаев. Пусть при просмотре токена не случилось ни одного сдвига и ни одной свёртки. В данном случае отмирают все ветви разбора. Парсер распознаёт это состояние как ошибочное и переходит в режим восстановления.

Второй случай связан с ситуацией, когда парсер завершает свою работу в недопускающем состоянии - состоянии, из которого невозможно свернуться к стартовому нетерминалу. В качестве примера рассмотрим следующее арифметическое выражение: “1+“. После просмотра последнего токена парсер останавливается в состоянии, из которого невозможно построить дерево разбора.

Реализованный алгоритм восстановления работает следующим образом. При обнаружении ошибки текущим токеном делается error. Он добавляется в грамматику вне зависимости от того, присутствуют в грамматике правила, содержащие его, или нет. Если в исходной грамматике он отсутствовал, то алгоритм сообщит о текущей ошибке и завершит работу.

В противном случае в текущем графе разбора начинается поиск состояния, из которого можно выполнить сдвиг по error’у. Поиск осуществляется поиском в ширину от вершин, находящихся на последнем уровне. Поскольку важно сохранить информацию о пропущенных токенах, то с каждой вершиной ассоциирована последовательность символов, которые были сняты со стека.

После того, как искомое состояние найдено, происходит свёртка или сдвиг по error’у. Полученное в результате одного из этих действий состояние делается текущим, и из него высчитывается множество токенов, которые позволят продолжить восстановление. Состоит это множество из тех символов, по которым из текущего состояния можно снова сдвинуться или свернуться, что в свою очередь определяется после просмотра goto и reduce таблиц.

В дальнейшем во входном потоке происходит поиск токена, который является элементом из вычисленного множества. После того, как такой находится, происходит свёртка всех пропущенных токенов в error, после чего разбор продолжается в обычном режиме.

Заключение

В ходе работы получены следующие результаты.

- Изучены существующие подходы к восстановлению после ошибок. В результате выбран алгоритм, использующий в своей работе error token.
- Реализовано восстановление после ошибок в RNGLR-модуле YaccConstructor'a.
- Реализован набор тестов, демонстрирующий работоспособность реализованного алгоритма
- Принято участие в конференции "Технологии Microsoft в теории и практике программирования". Тезисы опубликованы в сборнике материалов конференции.

Весь проект можно найти на сайте https://code.google.com/p/recursive-ascent/, автор принимал участие в проекте под учетной записью ivanovandrew2004.

Список литературы

[1] Grune D. Parsing Techniques - A Practical Guide / Ed. by C. Jacobs D. Grune. — 1990.

[2] McPeak Scott, C.Nekula George. A fast, practical GLR parser generator. — URL: http://www.scottmcpeak.com/elkhound.

[3] Syme Don, Granicz Adam, Cisternino Antonio. Expert F# 2.0. — Apress, 2010. — ISBN: 978-1-4302-2431-0. — URL: http://www.apress.com/9781430224310.

[4] Д.А. Авдюхин. Создание генератора GLR трансляторов для .NET. — URL: http://se.math.spbu.ru/SE/YearlyProjects/2012/YearlyProjects/2012/445/445_Avdyukhin_report.pdf.

[5] Руководство по Bison. — URL: http://www.linux.org.ru/books/GNU/bison/bison_9.html#SEC90.

[6] Сайт разработки инструмента YaccConstructor. — URL: http://code.google.com/p/recursive-ascent/.