САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Математико-механический факультет

Кафедра системного программирования

Рагозина Анастасия Константиновна

Апробация инструмента YaccConstructor на примере разработки парсера языка Transact-SQL

Курсовая работа

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

Санкт-Петербург
2013

Оглавление

Введение

Разработка синтаксических анализаторов при решении задач реинжиниринга программного обеспечения имеет ряд особенностей, из-за которых появляются дополнительные требования к применяемым для этого инструментам. Основные отличия, которые нужно учитывать при разработке синтаксических анализаторов для нужд реинжиниринга:

- Даже официальная документация часто содержит ошибки и не отражает всех возможностей языка. Для многих диалектов, особенно “доморощенных”, документация может вообще отсутствовать.

- Разработка ведётся итеративно. Нет необходимости реализовывать парсер, который поддерживает все конструкции языка. Достаточно обработать исходный код конкретной системы. При этом важно понимать, что он может меняться в процессе разработки.

- Время, необходимое для создания первой версии инструмента, обрабатывающей большинство конструкций, содержащихся в исходном коде, должно быть максимально уменьшено.

Одним из основных моментов при разработке синтаксических анализаторов является минимизация затрат на перенос грамматики из документации в инструмент. При этом отдельно нужно обратить внимание на то, чтобы отличия результирующей грамматики от приведённой в документации были минимальны. Это позволит в дальнейшем существенно сократить время, требуемое для поддержки новых конструкций. В ходе работы выяснилось, что результирующая грамматика не имеет критичных отличий от грамматики, описанной в документации.

На практике этого можно достичь использованием языка спецификации трансляции с хорошей синтаксической выразительностью и инструментальной поддержкой и генератора синтаксических анализаторов, который реализует GLR алгоритм анализа.

На кафедре Системного Программирования ведётся проект YaccConstructor [3], целью которого является создание инструмента для разработки синтаксических анализаторов для реинжиниринга ПО, учитывающего указанные выше особенности и практический опыт по разработке средств автоматизации реинжинирнга программного обеспечения.

Целью работы является апробация инструмента YaccConstructor и языка спецификации трансляций YARD, входящего в него, на примере разработки синтаксического анализатора языка Transact-SQL [2].

1. Постановка задачи

В рамках данной работы были поставлены следующие задачи.

- Реализовать парсер подмножества языка Transact-SQL. Разработку нужно вести итеративно, нет необходимости описывать все конструкции языка, нужно обработать только те, которые присутствуют в обрабатываемой системе. В качестве обрабатываемой системы был заранее зафиксирован набор стандартных системных процедур MS-SQL сервера.
- В ходе работы необходимо воспроизвести процесс разработки, характерный для реинжиниринга:
  - разработка грамматики ведется по доступной документации [2], в которой могут присутствовать ошибки или неточности.
  - цель — разбор конкретного исходного кода.
- Необходимо постараться оценить затраты на перенос грамматики из документации в инструмент. Это важно для оценки времени разработки парсера и проекта в целом.
- Понять насколько исходная грамматика отличается от результирующей. Это важно для того, чтобы оценить простоту понимания результирующей грамматики.

2. Язык YARD

YARD — язык спецификации трансляций, разрабатываемый на кафедре системного программирования в рамках проекта, посвященного созданию инструмента YaccConstructor, описанного выше. На рисунке(рис. 1), отображающем структуру инструмента YaccConstructor, выделен язык Yard, как одна из составляющих этого инструмента.

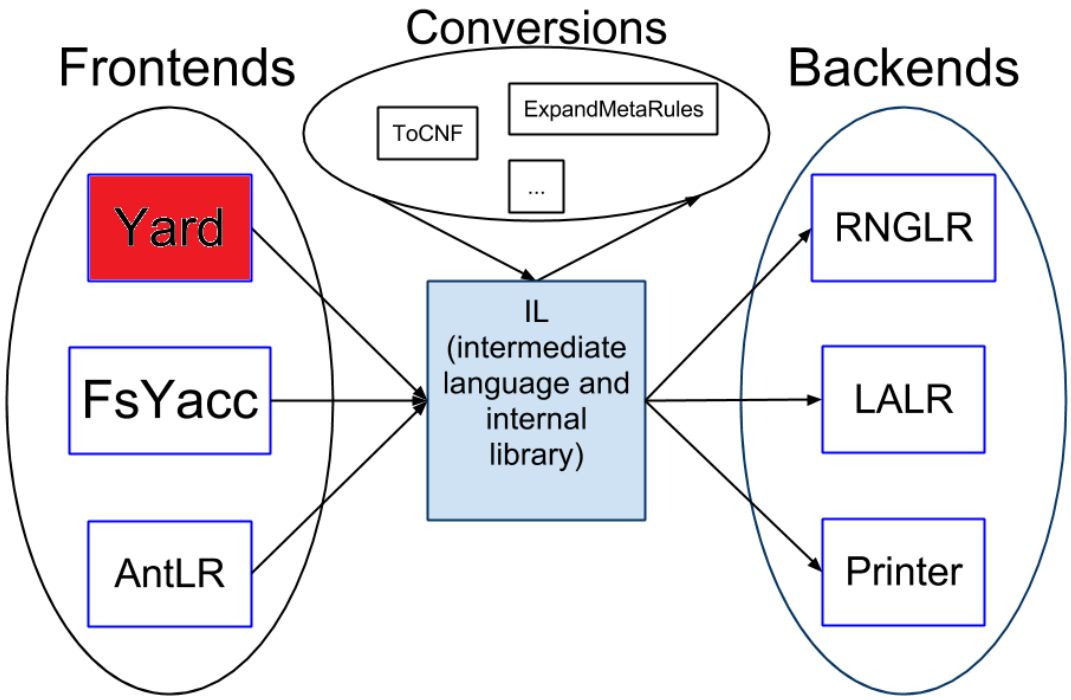

Рис. 1: YaccConstructor

Язык прост в изучении, а так же в понимании уже написанного. Опираясь на обзор современных средств автоматизации создания синтаксических анализаторов [6], можно оценить язык YARD, по рассмотренным там критериям. Синтаксическое правило состоит из левой и правой частей. В левой части название правила, в правой части тело - последовательность терминальных и нетерминальных символов, в которые правило должно разбираться при разборе. Для определения синтаксических правил используется расширенная форма Бэкуса-Наура[1], в которой введены операции * и +. Запись (rule)* означает, что правило нужно применить от 0 до N раз, где N — целое положительное число. Запись (rule)+ означает, что правило нужно применить N раз, где N — целое положительное число. Язык Yacc такие конструкции

не поддерживает, что немного усложняет написание грамматики и понимание уже написанного. Так же в языке YARD есть конструкция вида rule?, встречая такую конструкцию, анализатор определяет, возможно ли разобрать входной поток по этому правилу, если это возможно, то разбирает, а если нет, то пропускает и переходит к другим правилам. Такие конструкции позволяют уменьшить объем разрабатываемой грамматики.

Самое главное простота синтаксиса данного языка. Язык удобен, прост в понимании, код, написанный на языке YARD легко читается, понимание прочитанного так же не вызывает трудностей в отличие от множества других языков спецификаций. Семантика языка YARD так же отличается своей доступностью, простой и универсальностью, что позволяет переносить правила из документации в инструмент с минимальными изменениями. Например, так выглядит правило для конструкции сортировки в выборке (ORDER BY) языка SQL в документации и на языках YARD и FsYacc:

Документация:

```
[ ORDER BY
    {
        order_by_expression
        [ COLLATE collation_name ]
        [ ASC | DESC ]
    } [ ,...n ]
]
```

YARD:

```
orderBy:
    (KW_ORDER KW_BY
      comma_list<< (sql_expr (KW_COLLATE ident)?  (KW_ASC | KW_DESC)?) >>
    )?
```

FsYacc:

```
orderBy:
    |
    | ORDER BY orderByList

orderByList:
    | orderByColumn
```

```
    | orderByColumn COMMA orderByList

orderByColumn:
    | ID
    | ID ASC
    | ID DESC
```

Видно, что правило, написанное на языке YARD, подверглось наименьшим изменениям, по сравнению с правилами, написанными на других языках. Здесь приведен пример правила для конструкции ORDER BY языка T-SQL, в приложениях Вы сможете найти ссылки на полные грамматики T-SQL, написанные на Yacc[5] и FsYacc[4] и используемые в этом документе.

3. Особенности реализации

В ходе работы стало ясно, что в исходной документации присутствуют неточности, с которыми необходимо бороться вручную. Например, в исходной документации правило для оператора `EXECUTE PROCEDURE` позволяет писать только ключевое слово `OUTPUT`, но в исходном коде допускается написание не только ключевого слова `OUTPUT`, но и ключевого слова `OUT`. Таким образом правило для этого оператора на языке YARD немного отличается от исходного. В качестве еще одного примера можно привести правило для конструкции `SELECT`, которое подверглось многочисленным изменениям в процессе разработки.

Исходя из постановки задачи, правила из документации переносились по мере необходимости. Это необходимо для увеличение скорости разработки, так как написать часть грамматики быстрее, чем писать всю грамматику полностью. Так же необходимо точно знать, какие конструкции есть в обрабатываемой системе, а каких нет. Это важно для создания правильных оценок времени разработки транслятора и самого проекта в целом. Такой перенос правил по мере необходимости достигается за счет мощного синтаксиса языка спецификации трансляций, а так же использованного в инструменте YaccConstructor алгоритма синтаксического анализа GLR, позволяющего не разрешать неоднозначности в грамматике вручную, что особенно важно на начальном этапе разработки.

3.1. Особенности практического применения алгоритма синтаксического анализа GLR

Как упоминалось выше, одной из важных задач при автоматизированном реинжиниринге программного обеспечения является создание синтаксического анализатора входного языка и большинство инструментов, предназначенных для этого, используют алгоритмы LALR(1) и LL(1). Но такие алгоритмы не предназначены для работы с неоднозначными грамматиками. При использовании этих алгоритмов конфликты, появляющиеся в неоднозначных грамматиках, необходимо обрабатывать сразу, что приводит к дополнительным затратам времени и сил и плохо сказывается на сроках подготовки первой версии.

Более мощным классом анализаторов являются, основанные на GLR-алгоритме, основной особенностью которых является то, что они рассматривают все возможные пути разбора, благодаря чему могут бороться с неоднозначностями и сохранять все ветви разбора. При использовании GLR-анализаторов существенно упрощается процесс разработки и сопровождения грамматик, так как нет необходимости вручную

бороться с неоднозначностями в грамматике. В инструменте YaccConstructor реализован такой генератор GLR-трансляторов.

Работа с GLR-транслятором имеет некоторые особенности. Например, он позволяет быстро создать первый рабочий прототип парсера, однако у такого прототипа часто возникают проблемы с производительностью из-за большого числа возможных конфликтов. Поэтому парсер необходимо дорабатывать, а именно удалять из грамматики неоднозначности, которые замедляют работу транслятора.

Проблема, описанная выше, возникла и в рамках данной работы. Графики, приведенные ниже, наглядно отображают зависимость количества обработанных в секунду токенов от количества считанных токенов, в свою очередь эта зависимость влияет на производительность парсера.

Все замеры производительности парсера производились на компьютере с операционной системой Windows 7, с 64-битовым процессором Intel Core i3 -330M, тактовая частота которого равна 2,13 ГГц, и объемом оперативной памяти 2 Гб на входном файле размеров 11 Мб. На графике (рис. 2) черным цветом изображена зависимость количества токенов, обработанных за секунду, от количества считанных токенов в самом первом рабочем варианте парсера, а красным эта же зависимость после чистки грамматики.

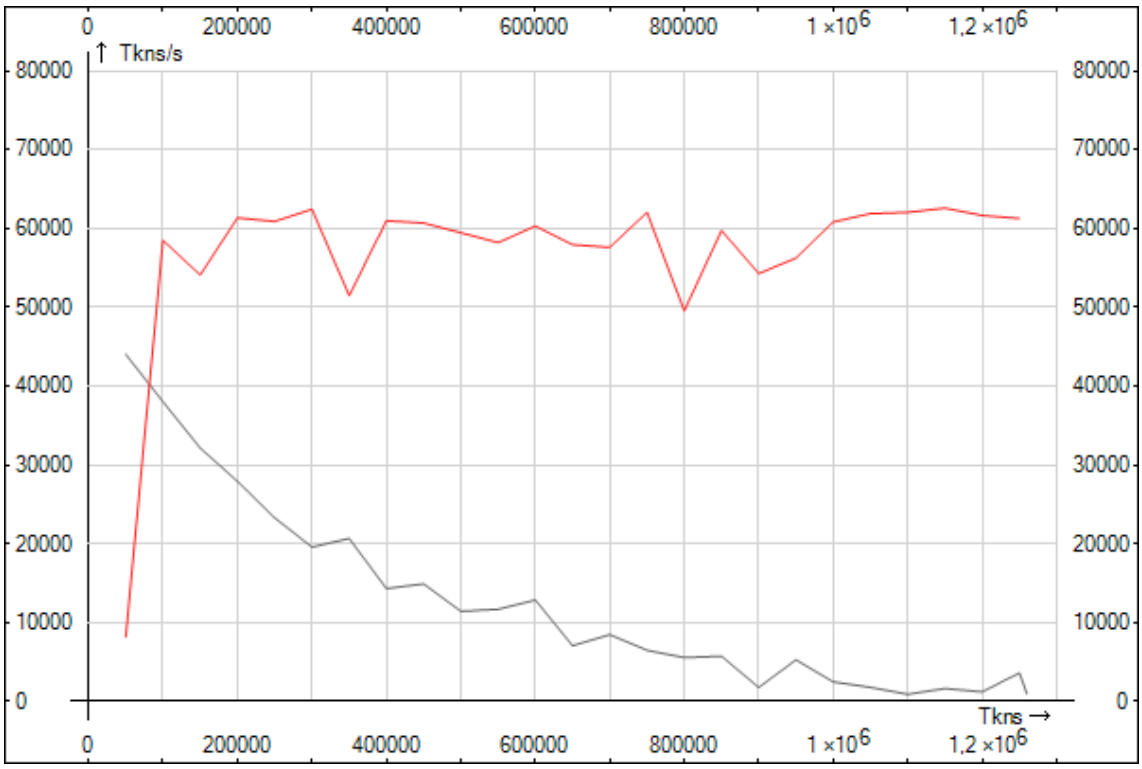

Рис. 2: Производительность парсера

На графике видно, что со временем производительность в первом рабочем прототипе парсера стремительно падает. На момент создания этого прототипа парсера насчитывалось более 555 тысяч конфликтов, а время необходимое для разбора файла составляло 290 секунд. Изменив лишь стартовое правило, количество конфликтов удалась сократить, а время необходимое для разбора того же файла уменьшилось на 30 секунд.

После доработки грамматики удалось повысить производительность парсера, сведя количество конфликтов к минимуму, результаты так же представлены на рисунке 2. Основные проблемы, порождающие неоднозначности в грамматике, были связаны с попыткой излишней детализации грамматики и дублированием кода. Попытки сделать грамматику как можно более точной привели к появлению лишних деталей и правил, необходимо было выделить более общие правила. Выделение таких общих конструкций позволило уменьшить количество дублируемого кода и снизить количество конфликтов, возникающих при разборе. Это типичные ошибки, возникающие при разработке парсеров.

При такой производительности транслятора время необходимое для разбора входного файла сократилось до 27 секунд.

Как правило, лексический анализатор работает быстрее, чем синтаксический. Поэтому если запустить лексический и синтаксический анализаторы в разных потоках выполнения, то это позволит уменьшить время работы за счет равномерного распределения работы между лексером и парсером. Результаты такого запуска лексического и синтаксического анализаторов в разных потоках можно увидеть на графике(рис. 3), где производительность парсера отображается черным цветом, а лексера красным. Места, где производительность лексера резко снижается, обозначают то, что в это время лексер ждал пока парсер обработает свою порцию токенов.

При таком запуске производительность удалось повысить, а время необходимое для разбора того же файла составило 20 секунд, что в 1,35 раз больше производительности работы парсера и лексера в одном потоке и более чем в 14 раз больше производительности первого рабочего прототипа. Это увеличение роста производительности отображено на графике(рис. 4), где черным цветом показывает производительность парсера после чистки грамматики, а красный при параллельном запуске лексера и парсера.

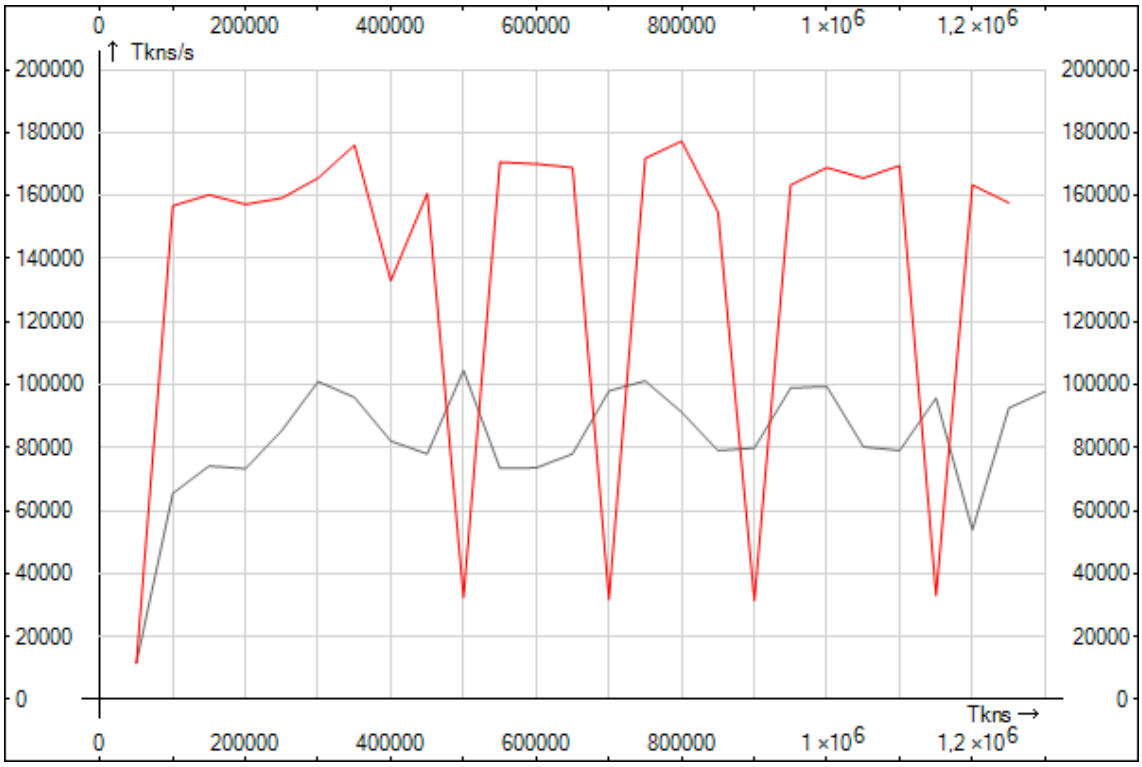

Рис. 3: Производительность лексера и парсера

4. Результаты

В ходе выполнения данной работы были получены следующие результаты.

- Получена грамматики подмножества языка Transact-SQL, в которой описаны основные управляющие конструкции, такие как: IF-THEN-ELSE, WHILE, и некоторые конструкции языка управления данными, например SELECT, UPDATE. Парсер тестировался на 2 млн строк кода, из которого уникальных процедур 11, 1200 строк кода. На примере этой работы получен практический опыт разработки парсеров, а так же написание тестов для них.

- Разработка парсера велась по доступной документации, в которой встречались ошибки и неточности, описанные выше. Полученный парсер разбирает конструкции, встречающиеся в обрабатываемом коде.

- Из-за минимальных отличий синтаксиса языка YARD и языка спецификаций, используемого в официальной документации, результирующая грамматика не имеет критичных отличий от исходной.

Так же получены рекомендации по доработке языка YARD и инструмента YaccConstructor в целом. В ходя работы стало ясно, что необходимы дополнитель-

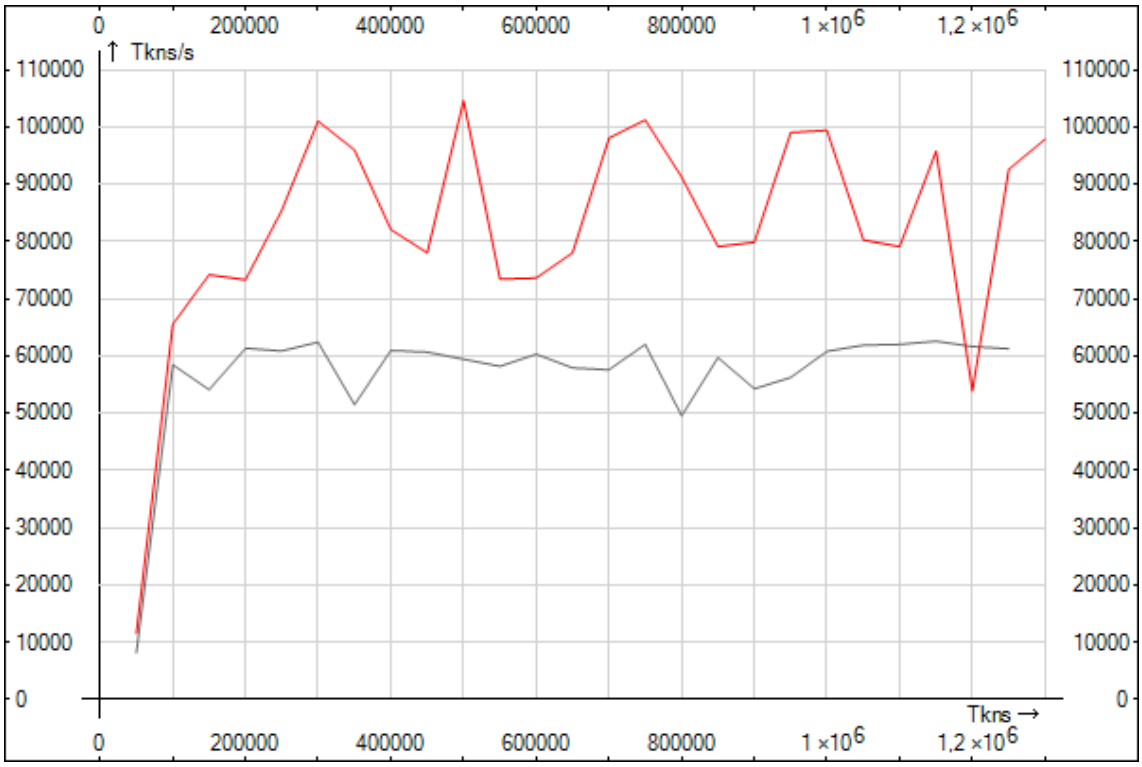

Рис. 4: Производительность парсера

ные средства отладки, такие как визуализация стека и вывод последних обработанных токенов. Для языка необходима полноценная IDE с навигацией по грамматике и поддержкой её рефакофринганга, должна быть реализована функция контроля качества кода на уровне парсера, аналогичная light-синтаксиу для языка F#. Кроме того, необходим контроль форматирования, проверка отсутствия конфликтов литералов и ключевых слов, проверка отсутствия неиспользуемых правил и использования неописанных нетерминалов. Это позволило бы упростить как процесс разработки грамматики, так и процесс ее чтения и сопровождения.

В качестве системы контроля версий проекта по разработке средств для реинжиниринга программного обеспечения используется Git, поэтому в рамках работы был дополнительно получен опыт работы с этой системой версионирования и инструментами для работы с ней.

Так же было принято участие в конференции “Технологии Microsoft в теории и практике программирования 2013”, проводимой Санкт-Петербургским государственным Политехническим университетом, при поддержке Российского представительства компаний Microsoft и EMC, посвященной технологиям Microsoft. Выступление проходило на секции, посвященной применению перспективных методов и технологий разработки программного обеспечения. На этой секции доклад занял второе ме-

сто. Тезисы к данной работе опубликованы в сборнике материалов межвузовского конкурса-конференции студентов, аспирантов и молодых ученых.

Список литературы

[1] А. Ахо, Р. Сети, Дж. Ульман. Компиляторы: принципы, технологии, инструменты.

[2] MSDN. — URL: http://msdn.microsoft.com/ru-ru/library/bb510741.aspx.

[3] YaccConstructor Сайт проекта. — URL: http://code.google.com/p/recursive-ascent/wiki/YaccConstructor.

[4] Грамматика T-SQL на языке FsYacc. — URL: http://sqlparser.codeplex.com/SourceControl/changeset/view/16372#147911.

[5] Грамматика T-SQL на языке Yacc. — URL: http://yaxx.googlecode.com/svn/trunk/sql/sql2.y.

[6] Чемоданов И. С., Дубчук. Н. П. Обзор современных средств автоматизации создания синтаксических анализаторов. — URL: http://www.sysprog.info/2006/12.pdf.