Санкт-Петербургский Государственный Университет

Математико-механический факультет

Кафедра системного программирования

Создание генератора GLR трансляторов для .NET

Курсовая работа студента 445 группы

Авдюхина Дмитрия Алексеевича

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

/подпись/

Санкт-Петербург

2012

Содержание

1 Введение

В реинжиниринге программного обеспечения [1] в первую очередь возникает задача синтаксического анализа. При этом частой является ситуация, когда у разработчиков нет подходящего инструмента, который мог бы этот анализ производить. Существующие парсеры изначально направлены на создание исполняемого кода, и редко позволяют получить представление входного потока, пригодную для решения задач реинжиниринга. В частности, почти никогда не сохраняется дерево разбора.

Всвязи с этим широкое распространение получили генераторы синтаксических анализаторов – инструменты, принимающие на вход грамматику, описанную в определенной форме, и порождающие на ее основе парсер, соответствующий ей. Генераторы отличаются друг от друга классом принимаемых грамматик и алгоритмом разбора, используемым порождаемым анализатором.

Наиболее часто в промышленности используются инструменты, основанные на LALR(1) [5] и LL(1) алгоритмах. Они обладают следующими преимуществами:

- Высокая производительность - время работы зависит линейно от длины входного потока.
- Класс LALR(1) грамматик достаточно широк. Кроме того, часто для грамматики существует эквивалентная ей LALR(1) грамматика.
- Существуют стандартные решения того, каким образом решаются возникающие конфликты, что делает поведение порожденного парсера предсказуемым.

Но они обладают существенным недостатком – они не предназначены для работы с неоднозначными грамматиками. В силу этого они не могут построить все деревья вывода, что бывает необходимо в задачах реинжиниринга.

Кроме того, на практике возникающие неоднозначности часто являются локальными и разрешаются после рассмотрения нескольких токенов. Однако LALR(1) анализаторы могут выбрать неверный способ разбора и не

разобрать корректную входную строку. Для борьбы с этим приходится переписывать грамматику, в результате чего она заметно отличается от оригинальной. Она становится менее понятной и более трудной в сопровождении.

Более мощным классом анализаторов являются GLR парсеры (Generalized LR) [8], основной особенностью которых является то, что они рассматривают все возможные пути разбора. Таким образом, они избавлены от перечисленных недостатков. При этом они не сильно уступают LALR(1) анализаторам по производительности:

- На полностью однозначных грамматиках время их работы также линейно.
- На неоднозначных время зависит кубически от размера входной строки. Но при этом строятся и эффективно хранятся внутри сложной структуры данных все возможные деревья разбора, число которых в общем случае может расти экспоненциально.

При использовании GLR анализаторов существенно упрощается процесс разработки и сопровождения грамматик, так как нет необходимости вручную бороться с неоднозначностями. Важным преимуществом GLR парсера является то, что он сохраняет все ветви разбора, и довольно быстро неверные ветви отмирают, а корректные остаются. В то же время LALR(1) парсер рассматривает лишь один путь разбора, который может оказаться неверным.

Возможность GLR парсеров порождать все возможные деревья в дальнейшем планируется использовать для распознавания диалектов языка. В процессе разбора будут существовать несколько ветвей. При нахождении определенных конструкций языка уточняется диалект, и из деревьев остаются только те, которые ему соответствуют.

2 Постановка задачи

В рамках данной работы были поставлены следующие задачи:

- Изучить существующие GLR алгоритмы. Поскольку на данный момент методов существует большое число, предполагается изучить лишь некоторые из них.
- Выбрать один из GLR алгоритмов и создать основанный на нем генератор для платформы .NET [4], который по входной грамматике строит
  - Синтаксический анализатор, результатом работы которого является структура, содержащая все деревья разбора входной строки.
  - Транслятор, реализующий вычисление атрибутов над полученной структурой.
- Аргументировать преимущества выбранного алгоритма и мотивацию к созданию собственного инструмента.
- Провести тестирование полученного инструмента, демонстрирующее его работоспособность.
- Поскольку генератор накладывает ограничения на использование конструкций, содержащихся в описании грамматики, необходимо также доработать преобразования, которые порождают эквивалентную грамматику без этих конструкций.

3 Принцип работы и обзор существующих GLR алгоритмов

Общая схема GLR разбора заключается в построении следующего графа:

- Вершины в нем характеризуются двумя параметрами: уровень и стандартное состояние парсера, используемое при LALR(1) анализе. Уровень вершины равен номеру токена, обработка которого осуществлялась в тот момент, когда вершина была создана.
- Изначально в графе есть только одна вершина на улевом уровне, из которой не исходит ребер.
- Входной поток токенов последовательно обрабатывается, в результате чего
- Ребра создаются в тот момент, когда происходит либо shift, либо reduce в какой-то вершине. С ребром связывается полученное в результате этого действия дерево разбора. При этом, reduce требует также обхода путей, которые входят в данную вершину для формирования набора сыновей получающегося дерева. Если ребро ведет в вершину с состоянием s и на текущем уровне не существует вершины с состоянием s, то создается соответствующая вершина.

В итоге в вершину, соответствующую конечному состоянию, будут входить ребра, содержащие информацию, по которой можно олучить все деревья разбора.

Рассмотрим некоторые GLR алгоритмы:

- Оригинальный алгоритм, разработанный Томитой [10], обладает тем недостатком, что он не всегда корректно работает с грамматиками, содержащими скрытую левую рекурсию:

  $$A ::= \beta A \alpha, \text{ где } \beta \neq \varepsilon, \text{ но } \beta \overset{*}{\Rightarrow} \varepsilon$$

  На практике такие ситуации встречаются достаточно часто, поэтому было необходимо поддержать работу с такими правилами.

- Алгоритм Фарши [7], одна из первых модификаций, поборовших этот недостаток, после каждого reduce–действия осуществляет поиск всех путей, которые содержат добавленное ребро. В результате многие вершины перепосещаются много раз, что отрицательно сказывается на производительности.

- Алгоритм Рекера, основанный на предыдущем алгоритме и отличающийся от него способом хранения леса вывода. Был реализован в инструменте ASF+SDF [15].

- Алгоритм Недерхофа и Сарбо [9] отличается способом обработки ε-правил. Для них создается много новых правил, которые, кроме того, не всегда являются корректными, что приводит к частым откатам и падению эыыективности.

Таким образом, полученные алгоритмы, корректно работающие с грамматиками, содержащими скрытую левую рекурсию, сильно проигрывают по производительности оригинальному либо из-за большого количества вводимых правил, либо из-за усложнения процесса разбора. Существуют другие модификации, но они также обладают теми же недостатками.

В качестве реализуемого алгоритма был выбран RNGLR (Right-Nulled GLR) [6], имеющий преимущества перед прочими GLR алгоритмами. Рассматривается несколько измененный алгоритм Томиты, который теперь не всегда корректно работает с правой скрытой рекурсией. Затем используется следующая идея: для правил вида

$$A ::= \alpha\beta, \text{ где } \beta \neq \varepsilon, \text{ но } \beta \stackrel{*}{\Rightarrow} \varepsilon$$

можно выполнить reduce даже в том случае, если в графе присутствует только строка α, то есть правая часть правила (β), из которой выводится пустая строка, необязательна. Полученный алгоритм уже работает корректно со всеми контекстно–свободными грамматиками. При этом оригинальный алгоритм разбора изменяется незначительно, число применяемых правил вырастает не сильно, что положительным образом сказывается на производительности.

На данный момент реализовано множество генераторов парсеров и трансляторов, основанных на GLR алгоритмах. Однако многие из них осно-

ваны не на RNGLR, в результате чего проигрывают либо по производительности, либо по классу покрываемых грамматик. Существуют инструменты, которые создают RNGLR парсеры для .NET, например, Hime Parser [13]. Однако, поскольку в дальнейшем планируется использовать полученный генератор для разбора диалектов языка, что потребует серьезного изменения алгоритма, было решено, что портирование и переиспользование функциональности существующего инструмента станет более сложной задачей, чем написание и дальнейшее изменение нового.

4 Реализация

4.1 YaccConstructor

На кафедре Системного Программирования СПбГУ ведется работа над инструментом YaccConstructor [11], позволяющим создавать синтаксические анализаторы под .NET. Основным языком разработки проекта является мультипарадигмальный язык F# [14].

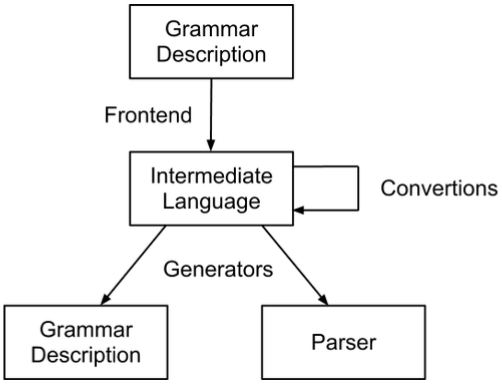

Рис. 1: Модульная структура YaccConstructor

Инструмент имеет модульную структуру [2], включающую в себя следующие части:

- Frontend – парсер входной грамматики, принимающий на вход ее описание на определенном языке.
- Внутреннее представление, в которое переводится входная грамматика. Над этим представлением производятся преобразования, которые порождают новую грамматику, эквивалентную исходной. Многие из них созданы с целью удаления из грамматики различных конструкций, которые не может обработать генератор.
- Backend (генератор) – по внутреннему представлению либо создает синтаксический анализатор, либо печатает полученную после всех преобразований грамматику на определенном (возможно, отличном от начального) языке.

Модульность YaccConstructor позволяет легко добавлять новую функциональность и создавать новые инструменты. Описываемый в данной работе генератор GLR анализаторов реализуется как один из модулей YaccConstructor-a.

4.2 Ограничения на грамматику

YARD [3], основной язык в YaccConstructor, предоставляет широкие возможности описания грамматик и синтаксически управляемых трансляций. Среди них нас будут интересовать следующие:

- S- и L-атрибуты. Правила могут иметь вид:

$$rule[arg] \ : \ l = left \ r = right \ \{arg \ + \ l \ + \ r\};$$

  Результат трансляции подвыражения $left$ будет сохранен в переменную l (связывание $left$ с l), аналогично в r будет содержаться результат трансляции $right$. Затем будет посчитана их сумма с arg, который передается правилу как аргумент, и возвращена в качестве результата трансляции правила rule.

- Запись правил в расширенной форме Бэкуса-Наура (EBNF).

- Параметризованные правила (макроправила) позволяют передавать правила как аргументы других правил. В данном примере:

$$rule << first \ second >> : \ first \ second;$$
$$s \ : \ rule << a \ b >>;$$

  правило для s будет раскрыто как $s \ : \ a \ b$;.

В реализцемом алгоритме S- и L-атрибуты раскрываются следующим образом: Формируется функция, формальными параметрами которой являются наследуемые атрибуты. Для каждого связывание происходит цикл соответствующей переменной по результату трансляции (поскольку их может быть несколько). Результатом трансляции является список из всех возможных вариантов трансляции. Добавление к нему элементов происходит на самом глубоком уровне вложенности.

Например, правило

$$rule[arg] \ : \ l = left \ r = right \ \{arg \ + \ l \ + \ r\};$$

породит функцию следующей структуры:

```
let translate_rule = fun arg ->
  [for l in translate_left do
    for r in translate_right
      yield arg + l + r]
```

Выбранный алгоритм накладывает ограничения на входную грамматику, и требует, чтобы в ней не было параметризованных правил и EBNF-конструкций. Ранее в YaccConstructor-е были реализованы преобразования, порождающие грамматику, эквивалентную оригинальной и не содержащую эти конструкции. Однако, на тот момент аргументами макроправил и подвыражениями EBNF-конструкций могли быть только литералы, терминалы и нетерминалы. Было решено, что такое ограничение является слишком сильным, и предпринята попытка расширить правила так, чтобы они могли принимать произвольные конструкции. Всвязи с этим заметно усложнился алгоритм преобразования. При этом раскрытие EBNF удалось реализовать полностью, а на метаправила было наложено следующее ограничение: в их аргументах нельзя использовать атрибуты, определенные за пределами макроправила.

4.3 Структуры данных

Основной причиной, по которой GLR транслятор работает эффективно даже в том случае, когда количество деревьев разбора растет экспоненциально, является то, что они имеют много одинаковых частей, которые могут храниться в памяти в единственном экземпляре и разделяться между различными деревьями. Такая структура, Shared Packed Parse Forest (SPPF) [6], сливает узлы, соответствующим одинаковым нетерминалам и принимающим одинаковую входную цепочку. При этом, если существует множество различных выводов одного нетерминала, то все возможные альтернативы образуют множество семейств детей, которое разделяет одну и ту же родительскую вершину. Например, на рисунке показано, каким об-

разом могут быть сжаты деревья:

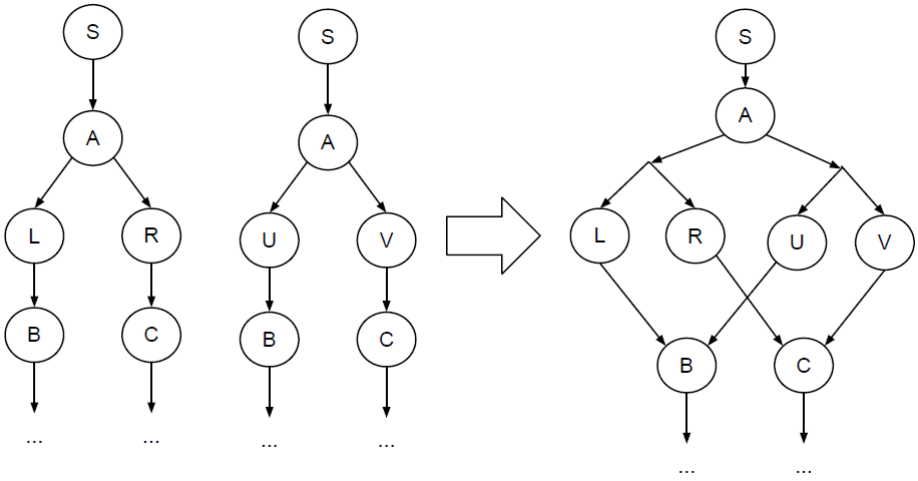

Рис. 2: Компактное представление деревьев вывода

В реализации SPPF представляется двумя косвенно рекурсивными типами:

- MultiAST – либо токен, либо нетерминал. В первом случае хранится значение, содержащееся в токене, Во втором - список семейств деревьев (AST) и ссылка на результат вычислений, который будет получен в процессе трансляции.
- AST – либо ε–дерево (то есть не содержащее не одного терминала), либо пара – номер применяемого правила и массив дочерних узлов.

При этом все ε–деревья, которые считаются заранее отдельно. В результате фильтрации из списка детей остается только один. Однако, поскольку в конечной структуре явно ε–деревья не хранятся (в этом нет необходимости, поскольку в процессе трансляции известно, какой нетерминал в данный момент разбирается), их фильтрацию нужно производить отдельно.

Для работы парсера необходимы action и goto таблицы. Отличие от обычных LALR(1) таблиц заключается в том, что хранятся все возможные shift и reduce действия, и, в силу специфики RNGLR алгоритма, дополнительные reduce для правил с обнуляемой правой частью. Это приводит к

тому, что увеличиваются размеры таблиц и усложняется их структура, и их загрузка в память требует больших ресурсов, что делало практически невозможным применение алгоритма. Чтобы этого избежать, все пары чисел представлялись как одно целое, и вся структура представлялась в виде одного большого массива (подмассивы представляются числом – длиной и самими элементами).

4.4 Тестирование

Тестирование инструмента происходило на большой, достаточно полной и неоднозначной грамматике transact – sql. Инструмент породил по ней транслятор, который сначала был запущен на небольших файлах. В результате для всех файлов были получены деревья разбора, а для содержащих неоднозначности также было порождено несколько деревьев.

Затем транслятор был запущен на большом входном файле, состоящем из более чем 700 000 токенов (более 100 000 строк кода). Для него было построено дерево разбора. Затем планировалось отфильтровать полученный лес и оттранслировать лишь одно из его деревьев. Однако процесс фильтрации завершился с ошибкой, потому что обычный рекурсиный обход получившегося дерева приводил к переполнению стека. Выборка одного дерева была переписана без рекурсии, что было достигнуто за счет хранения стека данных на куче.

Аналогичная проблема возникла и у транслятора, и решена она была тем же образом. Такие инструменты, как fsyacc [12], с этой проблемой не сталкиваются, поскольку они не строят явно дерево разбора, а выполняют трансляцию сразу же во время reduce-а.

Модифицированный инструмент на указанном файле пострил полный граф разбора за 1.5 минуты и оттранслировал одно из его деревьев за 0.5 минуты. Таким образом, была достигнута суммарная производительность более 6500 токенов в секунду и, тем самым, была подтверждена работоспособность инструмента.

5 Заключение

В процессе работы были получены следующие результаты:

- Реализован работающий генератор GLR трансяторов, поддерживающий как S–, так и L–атрибутные контекстно–свободные грамматики.
- Было проведено тестирование, показавшее работоспособность инструмента и его применимость для работы с большими файлами. Однако выяснилось, что при построении и трансляции единственного дерева порожденный транслятор сильно проигрывает по производительности LALR парсеру, созданному при помощи fsyacc, в 10–15 раз. Как недостаток fsyacc можно указать то, что для его применения потребовалось приведение грамматики к форме, не содержащей конфликтов, влекущих некорректный разбор, чего в случае с GLR парсером не пришлось бы делать. Однако практическое применение полученного инструмента сильно ограничивается его производительностью.
- Доработаны преобразования грамматики, приводящие ее к форме, принимаемой генератором.

В качестве дальнейшей работы над инструментом рассматриваются следующие варианты:

- Оптимизация времени работы порождаемого транслятора.
- Сравнение полученного иструмента с уже существующими генераторами GLR парсеров. Также планируется построить большую грамматику, не содержащую неоднозначностей, и сравнить производительность генерируемого парсера с LALR парсерами той же грамматиками.
- Поддержать такие элементы языка YARD, как резольверы и явные литералы в правилах. На данный момент первые не реализованы никаким образом, а вторые специальным преобразованием YaccConstructor-а преобразуются в токены.
- Поддержать стандартные средства, позволяющих сократить описания грамматик: приоритет и ассоциативность операций.

- Поддержка нескольких стартовых правил, что позволит на основе одних и тех же таблиц производить разборы грамматик, имеющих много общих элементов. В частности, это бывает полезно, когда необходимо разобрать некоторое подвыражение. При этом для LALR парсера это расширение не всегда позволительно, потому что в результате него сильно может вырасти число конфликтов.

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

6 Литература

Список литературы

[1] Автоматизированный реинжиниринг программ / Под ред. проф. А.Н. Терехова и А.А. Терехова. - СПб.: Издательство С.-Петербургского университета, 2000. 332 с.

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

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

[4] http://www.microsoft.com/NET/ (сайт платформы .NET)

[5] *Aho A.V., Johnson S.C.* LR Parsing.

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

[7] *Nozohoor-Farshi R.* GLR parsing for ϵ-grammars, 1991.

[8] *Scott McPeak, George C. Necula.* Elkhound: A fast, practical GLR parser generator

http://www.scottmcpeak.com/elkhound/ (сайт Elkhound)

[9] Nederhof, M.J. adn Sarbo, J.J. 1996. Increasing the applicability of LR parsing. In Recent Advances in Parsing Technology, H.Bunt and M. Tomita, eds. Kluwer Academic, Amsterdam, the Netherlands, 35–57.

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

[11] http://code.google.com/p/recursive-ascent/ (сайт разработки инструмента YaccConstructor)

[12] http://fsharppowerpack.codeplex.com/wikipage?title=FsYacc (пример использования FsYacc)

http://cs.wellesley.edu/~cs301/htmlman-3.06/manual026.html
(описание синтаксиса FsYacc)

[13] http://himeparser.codeplex.com/ (сайт разработки проекта Hime Parser)

[14] http://www.research.microsoft.com/fsharp (сайт F#)

[15] http://www.meta-environment.org (сайт разработчиков ASF+SDF)