Разработка GLR-анализатора для среды .NET

Григорьев Семён

11 апреля 2009 г.

Содержание

1 Введение.

Задачи автоматизированного реинжиниринга программ выдвигают особые требования к генераторам синтаксических анализаторов.

Для устаревшего языка сложно (а зачастую и невозможно) задать однозначную контекстно-свободную грамматику. Необходимо существенно преобразовать его спецификацию, которая приводится в документации, чтобы получить такую грамматику, но при этом она перестает быть сопровождаемой [1]. Поэтому устаревший язык обычно задается с помощью неоднозначной контекстно-свободной грамматики.

При поддержке нескольких диалектов языка необходима возможность лёгкой трансформации грамматики. Однако,зачастую, изменение одного правила приводит к появлению десятков конфликтов в грамматике [1], которые необходимо разрешать вручную. Это требует большого количества времени.

Как вариант решения этих задач в статье [1] предлагается использовать GLR грамматики и соответствующие инструменты построения анализаторов. Действительно, GLR-алгоритм разрешает неоднозначности в грамматике на уровне концепции. По этому задание спецификации трансляции становится проще, требует меньше времени. Получившийся код компактнее и сопровождаемее.

Главным достоинством GLR-алгоритма является обработка неоднозначных грамматик. Анализатор, построенный с помощью данного алгоритма, в результате разбора строит не единственное дерево, а несколько деревьев — лес, который можно сократить, используя специальные фильтры, а можно ,при задании в одной спецификации нескольких диалектов, вернуть весь лес для дальнейшего выбора нужного дерева/диалекта.

Стоит отметить, что по производительности такой анализатор, являясь некоторой “надстройкой” над LR-анализатором, незначительно ему уступает. На сегодняшний день в соотношении производитель-ность/класс разбираемых языков GLR-алгоритм выглядит наиболее предпочтительно.

2 Цель.

Целью работы является разработка GLR-анализатора для среды .NET. Такое решение принято потому, что :

- На данный момент нет реализации подобного инструмента на этой платформе.
- В рамках .NET есть возможность легко комбинировать функциональный и объектно ориентированный подходы. Это может быть полезным при разработке.

3 Ожидаемые результаты.

Результатом работы будет являться представление спецификации трансляции на основе атрибутной GLR-грамматики в среде .NET и алгоритм построения GLR-анализаторов на основе подобного представления.

4 Реализация.

Платформой для создания инструмента выбран .NET . Основным языком реализации является F# [2]. Так же предлагается использовать C#.

4.1 Внутреннее представление.

За основу внутреннего представления грамматики взято представление инструмента YARD. Это представление и ,соответственно, входной язык инструмента, содержит такие конструкции, как

- Конструкции расширенной формы Бэкуса-Наура;
- Макроправила (параметризация одних правил другими);
- Сгруппированные альтернативы;
- Предикаты;

Планируется расширить представление конструкциями для расширенных регулярных выражений и перестановок.

Так как для реализации GLR-алгоритма требуется построение LR(k) таблицы и предварительный анализ и преобразоване грамматики [4],например удаление циклов (преобразование правил вида $A \to B$; $B \to A$), то необходимо преобразовать входную грамматику. Для преобразований воспользуемся алгоритмами предложенными в работе [3]. Но кроме этого ,возможно ,придётся проводить дальнейшие преобразования. В частности скорее всего потребуется явное раскрытие сгруппированных альтернатив.

```
nonterm yard_alt_1 {
-> a ;
-> b ;
}
в
yard_alt_1 -> a;
yard_alt_1 -> b;
```

5 Обзор.

Стоит задача разработки анализатора для произвольной контекстно свободных грамматик на платформе .NET и реализации его на функциональном языке программирования F#. Необходимость такой разработки была обоснована выше.

Автоматизированный реинжиниринг программ выдвигает особые требования к инструменту анализа и языку спецификаций трансляции [3]. Используемые в разрабатываемом инструменте грамматики YARD удовлетворяют основным таким требованиям. Поэтому наибольший интерес сейчас представляет анализ внутренних алгоритмов анализа в существующих инструментах.

Предпочтительным алгоритмам анализа является алгоритм разбора произвольной контекстно свободной грамматики. Поэтому нужно рассмотреть инструменты, основанные на этом алгоритме. Важен способ реализации алгоритма, так как существуют несколько альтернатив: алгоритм Томиты (GLR-алгоритм), алгоритм Эрли (Early), рекурсивно-восходящий алгоритм. Также следует обратить преобразования грамматики, необходимые для построения анализатора.

Так как язык программирования F# является функциональным, то особый интерес представляет рекурсивно-восходящий алгоритм, так как GLR-алгоритм и алгоритм Эрли суть императивно. В связи с этим интерес представляет инструмент Jade – генератор рекурсивно восходящих парсеров.

В настоящее время нет GLR анализаторов на платформе .NET. Но существуют другие инструменты основанные на GLR-алгоритме.

- ASF+SDF [10] (Algebraic Specification Formalism + Syntax Definition Formalism) — генератор с широкими возможностями, но достаточно сложным входным языком. Является SGLR-инструментом (Scannerless, Generalized-LR).

- Bison [11] — развитие инструмента YACC. Все грамматики, созданные для оригинального YACC, будут работать и в Bison. Является одним из самых популярных и совершенных "потомков" YACC. При включении соответствующей опции использует GLR-алгоритм (по умолчанию LALR).

- Elkhound [12] — позиционируется как быстрый и удобный GLR-инструмент, созданный в университете Беркли (США), тем не менее обладает достаточно "бедным" входным языком (например, он не поддерживает конструкций расширенной формы Бэкуса-Наура).

В работе [3] проведён подробный анализ этих инструментов. Подводя итог, можно сказать, что на текущий момент нет инструмента, полностью удовлетворяющего требованиям автоматизированного реинжиниринга программ.

Так же нас будет интересовать такой инструмент как Jade, так как он является генератором рекурсивно-восходящих парсеров.

Jade это генератор рекурсивно-восходящих LALR(1) парсеров с целевым языком C. Его подробное описание приводится в статье [6]. При его реализации появилась проблема объёма кода целевого парсера. Так как при построении детерминированного парсера необходимо генерировать процедуры для каждого состояния, то объём кода быстро растёт, с ростом количества правил в грамматике. Так для языка Java, по расчётам, приведённым в статье[6], объём кода составляет примерно 4 мегабайта. В Jade эта проблема решается путём создания глобальной структуры(массива состояний), где хранится информация, позволяющая переиспользовать процедуры. (Подробнее об этом можно прочесть в статье [6]).

5.1 Рекурсивно восходящий алгоритм.

Существует альтернатива табличным анализаторам – рекурсивно-восходящие анализаторы. Идея состоит в том, чтобы не использовать стек явно, а заменить его стеком вызова рекурсивных функций и эмулировать поведения автомата вызовом функций. Для этого можно построить функцию соответствующую каждому состоянию, как это делается в статье [8]. В этом случае можно получить детерминированный LR анализ.

Таким образом – рекурсивно восходящий алгоритм это аналог рекурсивного спуска, но для LR грамматики. Однако, как показано в статье [6], при такой реализации возникают проблемы с быстрым ростом объёма кода целевых функций.

В статье [9] показан подход к решению этой задачи несколько с другой стороны. Показано, что кожно построить функции, оперирующие уже не одним состоянием, а множеством состояний, и свести ,таким образом, количество необходимых функций к двум взаимно рекурсивным:

- parse q i $=\{(A \to a., i) | A \to a. \in q\} \bigcup$
  $\{(A \to a.b, k) | i = xj, (A \to a.b, k) \in climb\ \text{q x j}\} \bigcup$
  $\{(A \to a.b, k) | B \to e, (A \to a.b, k) \in climb\ \text{q B j}\ \}$

- climb q X i $= \{(A \to a.Xb, k) | (A \to aX.b, k) \in parse(\text{goto q X}) i, a \neq e, A \to a.Xb \in q\} \bigcup$
  $\{(A \to a.b, l) | (C \to X.c, j) \in parse(\text{goto q X}) i, (A \to a.b, l) \in climb\ \text{q C j}\}$

Определим также функцию:

- goto q X = $\{A \to aX.b | A \to a.Xb \in q^*\}$,

  где

- $q^* = q \bigcup \{B \to .c | A \to a.Bb \in q^*\} \bigcup \{x \to .x | A \to a.xb \in q^*\}$

Пользуясь таким определением можно построить функции из статьи[8], однако сейчас интереснее их реализация, согласно определению.

Важно, что при решении нашей задачи нет необходимости генерации функции для каждого состояния. Так как цель – получение недетерминированного разбора, то можно реализовать всего две функции: parse и climb, определённые выше. Это позволит решить проблему объёма кода, возникшую в Jade. Объём кода резко сократится и, более того, будет получен недетерминированный вариант разбора.

Используя рекурсивно-восходящий алгоритм можно получить эквивалент алгоритму Томиты. При использовать описанных ваше функции ,автоматически получается ветвление в момент возникновения неоднозначности (разделение стека в алгоритме Томиты). Это получается благодаря тому, что они принимают на вход состояние и возвращают множество всех возможных состояний.

Однако возникает проблема – сложность алгоритма, основанного на функциях parse и climb экспоненциальная. Для борьбы с этим в статье [7] предлагается дополнить функции механизмом запоминания результатов предыдущих вызовов, чтобы при очередном вызове , в случае, если эта функция уже вызывалась с такими параметрами, то результат возвращался без повторного вычисления. Доказывается [7], что в при такой реализации оценки по времени будут порядка $O(n^3)$.

Запоминание результатов предыдущих вычислений и будет аналогом объединения состояний и получения структурированного в виде графа стека (graph-structured stack) в алгоритме Томиты.

Технически этого можно добиться реализовав функцию memoize. На языке F# её можно реализовать так:

```
let memoize (f: 'a ->'b) =
     let t = new System.Collections.Generic.Dictionary<'a,'b>()
     in
     fun a ->
            if t.ContainsKey(a)
            then t.[a]
            else
               let res = f a
               in
               t.Add(a,res);
               res
```

В дальнейшем в качестве функции f будет выступать функция parse и climb.

Дополнительного ускорения можно добиться заранее вычислив функцию goto. Действительно, построение замыкания (closure), необходимое для вычисления функции goto дорогая операция, а вызов goto происходит при каждом вызове функции climb. Поэтому можно вычислить goto на этапе построения анализатора, и в целевой программе (в функции climb) свести вызов (goto q X) к поиску по ключу.

5.2 Расширенные контекстно свободные грамматики.

На практике удобно пользоваться расширенной контекстно свободной грамматикой. То-есть такой грамматикой, у которой в правых частях правил могут использоваться конструкции регулярных выражений. Необходимость использования EBNF обоснована в работе [3].

В качестве примера рассмотрим грамматику арифмитических выражений без приоритетов :

1) $E \rightarrow E$'+'E
2) $E \rightarrow E$'$*$'E
3) $E \rightarrow$'('E')'
4) $E \rightarrow$'a'

В данном случае удобно объединить правила 1) и 2) в одно:

$E \rightarrow E$('+' | '$*$')E

Далее:

$E \rightarrow (E$('+' | '$*$')$E)|($ '('E')' $)|$'a' (1).

Для того, чтобы построить табличный LR анализатор необходимо преобразовать расширенную контекстно свободную грамматику. Нужно избавиться от конструкции BNF в правых частях. Однако можно попробовать обойтись без преобразований грамматики.

Существуют способы поддержки расширенной контекстно свободной грамматики на уровне анализатора, без преобразований входной грамматики [7]. Для этого нужно переопределить функцию goto. Можно заменить позицию точки в правиле на номер состояния конечного автомата, построенного по соответствующему регулярному выражению. Заметим, что определённая выше функция goto может быть описана в терминах конечного автомата. В этом случае регулярное выражение и ,следовательно ,построенный по нему автомат очень прост.

Для построения конечного автомата по регулярному выражению воспользуемся алгоритмом Томпсона[5]. Результатом работы данного алгоритма является недетерминированный конечный автомат (НКА). Так как правая часть правила (1) - регулярное выражение, то применим к ней алгоритм Томпсона. Получим следующий НКА: **здесь должен быть граф**

Каждое состояние и каждый переход КА должны преобразоваться в состояние и переход LR автомата соответственно. Заметим, что в в полученном НКА много ε-переходов. Например, каждая альтернатива вносит два дополнительных состояния и 4 ε-перехода. Чтобы уменьшить количество переходов в результирующем LR-автомате можно преобразовать НКА в детерминированный конечный автомат(ДКА). Для этого применим стандартный алгоритм преобразования НКА в ДКА[5]. Полученный в результате преобразований ДКА будет выглядеть следующим образом:**здесь должен быть граф**

Перейдём к построению LR ситуаций. Как было сказано выше, основная идея состоит в том, что точку из классического определения ситуации нужно заменить состоянием, получившегося на предыдущем шаге, ДКА. Поясним. Предположим, что правило грамматики не содержит конструкций BNF. Например:

$E \rightarrow E$'+'E

Множество ситуаций для данного правила:

1) $E \rightarrow .E$'+'E
2) $E \rightarrow E$'.+'E
3) $E \rightarrow E$'+'.E
4) $E \rightarrow E$'+'E.

Теперь построим ДКА в соответствии с нашим алгоритмом. Получим:

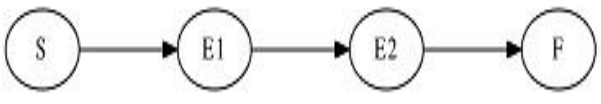

Рис. 1:

У построенного ДКА четыре состояния: S_1 , S_2 , S_3 , S_4. Сопоставим состоянию S_1 ситуацию $E \rightarrow .E$'+'E, S2 ситуацию $E \rightarrow E$'.+'E и так далее. В итоге получим соответствие:

$S_1 - E \to .E'+'E$
$S_2 - E \to E'.+'E$
$S_3 - E \to E'+'.E$
$S_4 - E \to E'+'E.$

Таким образом, киждой LR-ситуации мы смогли сопоставить состояние ДКА, построенного по правой части правила. Теперь вернёмся к предыдущему примеру и обобщим ситуацию.

Предположим, что в правой части правила - произвольное регулярное выражение. Мы построили по нему ДКА. Теперь построим для каждого состояния ДКА LR-ситуацию. Множество состояний ДКА:

5.3 Построение деревьев вывода.

Для практических целей одна только информация о том, принадлежит входная цепочка данному языку или нет, не очень полезна. Гораздо более ценный результат - дерево вывода цепочки в данной грамматике. В нашем случае речь будет идти о множестве деревьев (лесе). Для краткости будем называть его лесом вывода.

Дополним эту грамматику и пронумеруем правила. Введём стартовый нетерминал S и соответственно правило $S \to E$ и получим следующую грамматику:

1) $S \to E$
2) $E \to E + E$
3) $E \to E * E$
4) $E \to (E)$
5) $E \to a$

Данная грамматика порождает арифметические выражения с двумя бинарными операциями и скобками.

Очевидно, что такая грамматика неоднозначна. Существуют цепочки, которые выводимы несколькими способами. Например: возьмём цепочку a+a+a. Она имеет два левосторонних вывода:
$S \xrightarrow{1} E \xrightarrow{2} E + E \xrightarrow{5} a + E \xrightarrow{2} a + E + E \xrightarrow{5} a + a + E \xrightarrow{5} a + a + a$
и
$S \xrightarrow{1} E \xrightarrow{2} E + E \xrightarrow{2} E + E + E \xrightarrow{5} a + E + E \xrightarrow{5} a + a + E \xrightarrow{5} a + a + a$
Над стрелкой - номер применяемого на данном шаге вывода правила.

Так как инструмент предназначен для работы с неоднозначными грамматиками, то в общем случае он должен строить лес вывода. То есть в случае если грамматика неоднозначна и существует несколько способов вывода некоторой входной цепочки в данной грамматике, то должны быть построены все возможные деревья вывода.

Список литературы

[1] *Mark G.J. van den Brand, Alex Sellink, Chris Verhoef* Current Parsing Techniques in Software Renovation Considered Harmful.// IWPC '98: Proceedings of the 6th International Workshop on Program Comprehension. — IEEE Computer Society, Washington,1998.

[2] *h*ttp://www.research.microsoft.com/fsharp (дистрибутивы и документация по языку F#)

[3] *Чемоданов И.С.* Генератор синтаксических анализаторов для решения задач автоматизированного реинжиниринга программ.

[4] *Dick Grune, Ceriel Jacobs* PARSING TECHNIQUES A Practical Guide

[5] *Ахо А., Сети Р., Ульман Дж.* Компиляторы: принципы, технологии, инструменты. М:. Издательский дом «Вильямс»2003. 768 с.

[6] *Ronald Veldena* Jade, a recursive ascent LALR(1) parser generator. September 8,1998

[7] *Rene Leermakers* Non-deterministic Recursive Ascent Parsing. Philips Research Laboratories, P.O. Box 80.000, 5600 JA Eindhoven, The Netherlands.

[8] *Larry Morell, David Middleton* RECURSIVE-ASCENT PARSING. Arkansas Tech University Russellville, Arkansas.

[9] *Lex Augusteijn* Recursive Ascent Parsing (Re: Parsing techniques). lex@prl.philips.nl (Lex Augusteijn) Mon, 10 May 1993 07:03:39 GMT

[10] http://www.cwi.nl/projects/MetaEnv (сайт разработчиков ASF+SDF)

[11] http://www.gnu.org/software/bison (сайт разработчиков Bison)

[12] http://www.cs.berkeley.edu/smcpeak/elkhound (сайт разработчиков Elkhound)