Григорьев Семён Вячеславович.
Научный руководитель: Лукичёв Александр Сергеевич.
Разработка GLR-анализатора для платформы .NET.

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 предлагается использовать GLR грамматики и соответствующие инструменты построения анализаторов [1]. 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 по неоднозначной грамматике 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.

Работу GLR-алгоритма можно рассматривать как параллельное исполнение набора LR-анализаторов. При этом данный набор дополняется процедурой управления стеками, оптимизирующей представление стеков путем их «склеивания» и «расклеивания», что позволяет хранить и строить параллельные выводы в рамках одного LR-анализатора, лишь в моменты их различия добавляя параллельный анализатор.

Оказалось, что весьма наглядно такой алгоритм может быть представлен в виде двух взаимно-рекурсивных функций (рекурсивно-

восходящий алгоритм, recursive ascent). При этом расклеивание стека получается естественным образом как ветвление в одной из функций, а обратное склеивание может быть реализовано как кэширование результата функции.

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

Удобным способом формального определения грамматики, элементов и атрибутов языка программирования является расширенная нормальная форма Бэкуса-Наура. Поэтому инструмент должен работать с расширенными контекстно-свободными грамматиками.

При работе с инструментом пользователь ожидает получить результат описанный в терминах заданной им грамматики. Это выдвигает дополнительные требования к алгоритму. В случае, если входная грамматика была каким-либо образом преобразована, например с целью раскрыть конструкции EBNF, то появляется необходимость в построении "обратного" преобразования. Это преобразование должно "перевести" результат обратно в термины входной грамматики. Такие преобразования требуют дополнительных ресурсов и усложняют инструмент. Поэтому наиболее предпочтительными является алгоритмы, позволяющие обойтись без дополнительных преобразований грамматики.

2 Обзор

Предпочтительным алгоритмом анализа является алгоритм для работы с произвольными контекстно-свободными грамматиками. Поэтому были рассмотрены инструменты, обладающие соответствующими возможностями. Важным моментом является внутренняя реализация, так как существуют несколько альтернативных подходов к работе с произвольными контекстно-свободными грамматиками: алгоритм Томиты (GLR-алгоритм), алгоритм Эрли (Early), рекурсивно-восходящий алгоритм.

В настоящее время существуют следующие инструменты основанные на GLR-алгоритме.

- ASF+SDF [13] (Algebraic Specification Formalism + Syntax Definition Formalism) - генератор с широкими возможностями, но достаточно сложным входным языком. Является SGLR-инструментом (Scannerless, Generalized-LR).

- Bison [14] - 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 [15] - 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 [6] проведён подробный анализ этих инструментов.

Необходимо отметить, что все рассмотренные инструменты либо обладают сравнительно бедным входным языком, что усложняет разработку, либо проводят сильные преобразования грамматики во время работы, что может серьёзно усложнить отладку.

Так же был рассмотрен инструмент Jade. Jade это генератор рекурсивно-восходящих LALR(1) парсеров с целевым языком C. Его подробное описание приводится в статье [9]. При реализации данного инструмента 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 объём кода составляет примерно 4 мегабайта [9]. В Jade эта проблема решается путём создания глобальной структуры(массива состояний), где хранится информация, позволяющая переиспользовать процедуры [9].

2.1 Вычисление атрибутов

При работе с произвольными грамматиками выдвигаются дополнительные требования к алгоритму вычисления атрибутов. В качестве атрибута пользователь может указать действие, обладающее побочным эффектом. Так как в момент разбора мы не можем сказать, завершиться ли текущая ветвь удачно, то нельзя вычислять атрибуты непосредственно по ходу разбора входной строки, так как в этом случае могут быть совершены лишние действия.

Были рассмотрены два подхода к решению этой проблемы:

- Отложенные вычисления (continuation passing style, CPS). Непосредственно во время разбора атрибуты не вычисляются. Вычисления откладываются. Строится функция, которая вычисляется только один раз, после удачного завершения разбора.

- Построение леса вывода и последующее вычисление атрибутов над ним. Первым шагом строится лес вывода. В него попадают только деревья, соответствующие успешным вариантам разбора. Следующим шагом над полученным лесом производятся вычисления, соответствующие заданным атрибутам. Назовём это действие интерпретацией дерева вывода.

Оба этих подхода гарантируют, что будут выполнены действия, соответствующие только успешным вариантам разбора.

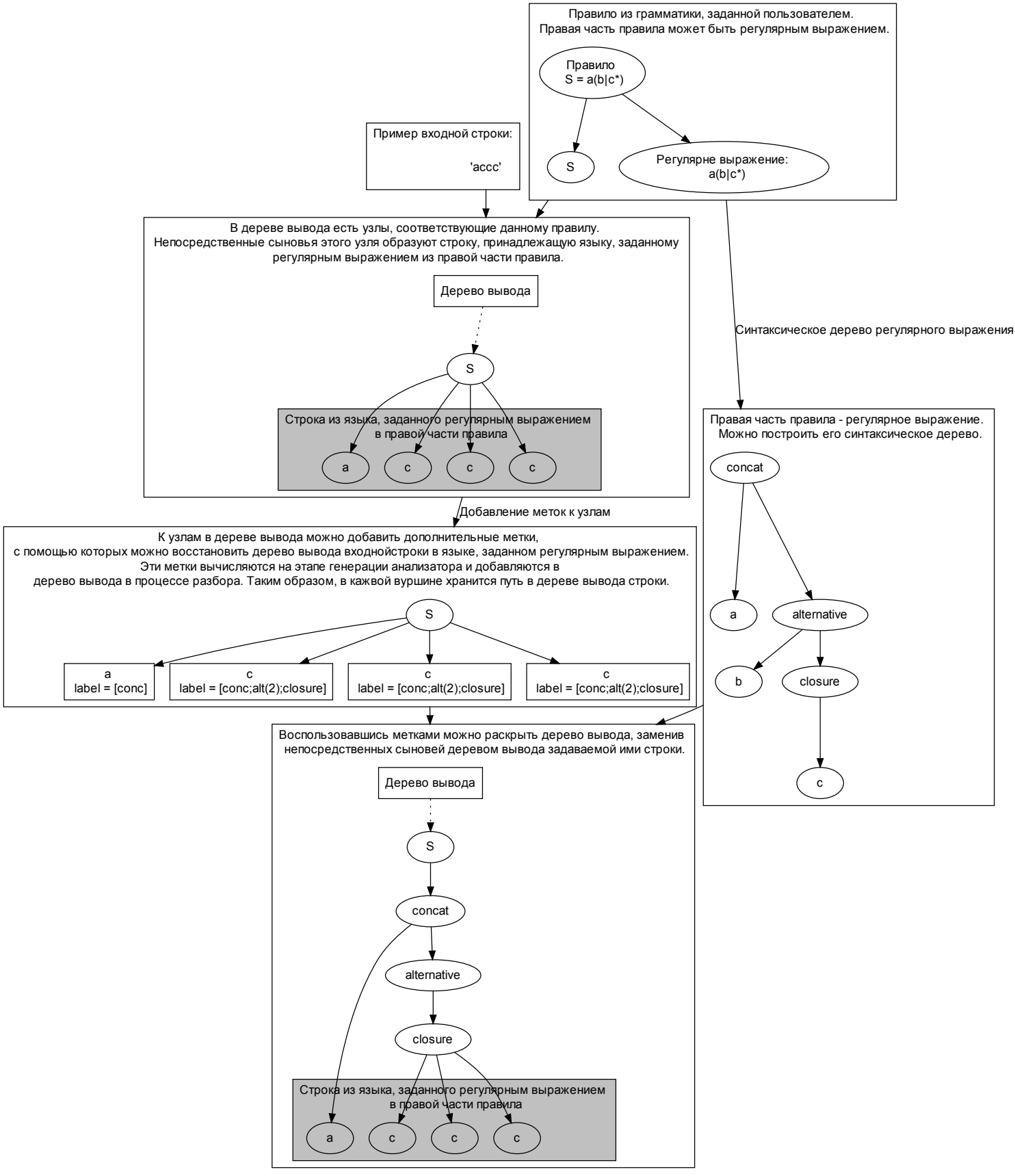
Второй подход является более удобным для конечного пользователя, так как позволяет явно получить дерево вывода, что упрощает отладку.

Вторым важным требованием к алгоритму вычисления атрибутов является минимизация, а в идеале - отсутствие, преобразований входной грамматики/дерева вывода.

Так как инструмент должен работать с EBNF грамматиками без их преобразования, то при вычислении над деревом вывода возникли сложности, связанные с тем, что для конкретного узла сыновья одного уровня образуют строку языка, заданного регулярным выражением в правой части правила, соответствующего этому узлу. Для вычислений нужна более подробная информация о выводе данной строки. Для решения этой проблемы возможны несколько вариантов:

- С преобразованием входной грамматики. Добавление правил в грамматику, соответствующих атрибутам в исходной грамматике.

- Без преобразования входной грамматики. Добавление меток к узлам дерева вывода, с помощью которых можно восстановить дерево вывода этой строки.


Правило из грамматики, заданной пользователем.
Правая часть правила может быть регулярным выражением.
Правило
S = a(b|c*)
S
Регулярне выражение:
a(b|c*)
Пример входной строки:
'accc'
В дереве вывода есть узлы, соответствующие данному правилу.
Непосредственные сыновья этого узла образуют строку, принадлежащую языку, заданному регулярным выражением из правой части правила.
Дерево вывода
S
Строка из языка, заданного регулярным выражением в правой части правила
a
c
c
c
Синтаксическое дерево регулярного выражения
Правая часть правила - регулярное выражение.
Можно построить его синтаксическое дерево.
concat
a
alternative
b
closure
c
Добавление меток к узлам
К узлам в дереве вывода можно добавить дополнительные метки,
с помощью которых можно восстановить дерево вывода входнойстроки в языке, заданном регулярным выражением.
Эти метки вычисляются на этапе генерации анализатора и добавляются в
дерево вывода в процессе разбора. Таким образом, в кажвой вуршине хранится путь в дереве вывода строки.
S
a
label = [conc]
c
label = [conc;alt(2);closure]
c
label = [conc;alt(2);closure]
c
label = [conc;alt(2);closure]
Воспользовавшись метками можно раскрыть дерево вывода, заменив непосредственных сыновей деревом вывода задаваемой ими строки.
Дерево вывода
S
concat
alternative
closure
Строка из языка, заданного регулярным выражением в правой части правила
a
c
c
c

Был выбран второй вариант, так как одним из требований к инструменту было минимальное преобразование входной грамматики.

Результат обзора - в качестве алгоритма анализа выбран рекурсивно-восходящий алгоритм, для вычисления атрибутов выбран второй подход(интерпретация дерева вывода).

3 Постановка задачи

В результате обзора были сформулированы следующие требования к инструменту:

- Возможность работы с неоднозначными грамматиками.
- Возможность воспринимать основные типы грамматик, встречающиеся в формальных описаниях языков. Как правило грамматики в EBNF.
- Поддержка атрибутных грамматик.

Цель данной работы: разработать прототип инструмента, обладающего перечисленными выше возможностями.

4 Описание решения

Необходимость поддержки неоднозначных грамматик предопределила выбор GLR-алгоритма в качестве анализатора. Для реализации выбран рекурсивно-восходящий алгоритм, модернизированный для работы с расширенными контекстно-свободными грамматиками без их преобразования.

В качестве фронтенда, обладающего мощным и удобным языком спецификации трансляции был выбран YARD.

Для вычисления атрибутов выбран второй подход - интерпретация дерева вывода. Основная идея заключается в том, что генератор строит набор функций - каждая функция соответствует одному правилу грамматики. После построения дерева вывода, оно обходится снизу вверх и в каждом узле вычисляется соответствующая ему функция. Параметры функции - сыновья данного узла. Функции вызываются по рефлексии, что показывает возможность в будущем сделать инструмент более гибким благодаря возможности динамической компиляции пользовательского action-кода.

Инструмент реализуется на платформе .NET [3], на функциональном языке F# [2].

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] *h*ttp://www.microsoft.com/NET/

[4] *I*SO/IEC 14977 : 1996(E)

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

[6] *Чемоданов И.С., Дубчук Н.П.* Обзор современных средств автоматизации создания синтаксических анализаторов // Системное программирование. - СПб.: Изд-во С.-Петерб. ун-та, 2006. - с. 286-316.

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

[8] *Ахо А., Сети Р., Ульман Дж.* Компиляторы: принципы, технологии, инструменты. М:. Издательский дом <Вильямс>2003. 768 с.

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

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

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

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

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

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

[15] http://www.scottmcpeak/elkhound/ (сайт разработчиков Elkhound)