GLR-парсеры для грамматик в расширенной форме Бэкуса-Наура

Алефиров А.А., студент магистратуры кафедры компьютерных технологий НИУ ИТМО, alefirov93aa@gmail.com

Аннотация

Расширенная форма Бэкуса-Наура обладает преимуществами перед обычной формой Бэкуса-Наура.

В данной статье предлагается метод генерации синтаксических анализаторов, которые сохраняют расширенную форму Бэкуса-Наура исходных грамматик в своей структуре, имеют возможность производить вывод в терминах исходной грамматики, и работает со всем классом КС-грамматик.

Введение

Использование расширенной формы Бэкуса-Наура (EBNF) грамматик позволяет улучшить выразительность контекстно-свободных грамматик и широко применяется в спецификациях языков программирования[1]. Традиционно генераторы синтаксических анализаторов предварительно трансформируют исходные грамматики, упраздняя EBNF, добавляя новые нетерминалы и правила вывода. Однако уже многие годы создаются прямые алгоритмы генерации анализаторов непосредственно из грамматик в EBNF и их интерпретации[2][3]. Одно из преимуществ такого подхода - вывод порожденного синтаксическического анализатора соответствует терминологии исходной грамматики. Также синтаксические анализаторы, порождённые напрямую из грамматик в EBNF, могут показывать выигрыш в скорости распознавания перед анализаторами, порождёнными из трансформированных грамматик[4].

Упомянутые выше алгоритмы имеют ограничения по классу грамматик. В то же время нет решений вычисления семантических выражений (атрибутов), добавляемых в грамматики, для алгоритмов, сохраняющих EBNF. В данной статье описывается алгоритм генерации LR-парсеров, сохраняющий EBNF исходных грамматик, принимающий

весь класс контекстно-свободных грамматик. Также алгоритм позволяет вычисление семантических выражений, заданных для грамматик, по выводу сгенерированных анализаторов. Допуск всего класса КС грамматик обеспечивается с помощью использования техники обобщённого LR-анализа.

Термины и определения

Определение 1. ***Грамматика с регулярной правой частью*** $G = (V_N, V_T, S, Q, \delta, F, P)$, где V_N - конечное множество нетерминальных символов, V_T - конечное множество терминальных символов, $S \epsilon V_N$ - стартовый символ, Q - конечное множество состояний правых частей, $\delta : Q \times V \rightarrow Q$ - функция переходов конечных автоматов (где $V = V_N \cup V_T$), $F \subset Q$ - множество конечных состояний и P - множество правил вывода. Правило представляет собой пару *(A, q)*, где $A \epsilon V_N$ - левая часть правила, $q \epsilon Q$ - стартовое состояние правой части продукции. Грамматика, описанная в EBNF, трансформируется в эквивалентную ей грамматику с регулярной правой частью созданием конечных автоматов, допускающих те же языки, что и регулярные выражения правых частей исходной грамматики.

Определение 2. ***Стэк, представленный графом (GSS, Graph Structured Stack)*** GLR-анализа - граф, вершины которого характеризуются двумя свойствами - *state* и *level*. Первое обозначает, какому состоянию LR-автомата соответствует эта вершина, а второе - количество первых символов входной цепочки, уже обработанных анализатором. Дуги графа помечены символами входной цепочки, либо нетерминалами, к которым были свернуты части входной цепочки. Такой граф используется в GLR анализе вместо обычного стэка в классическом LR-анализе для обеспечения возможности недетерминированного анализа.

Алгоритм

Основная проблема, возникающая при LR-анализе грамматик с регулярными правыми частями - это определение левых концов правых частей правил для произведений свёрток, поскольку в таком случае основы для свёрток не имеют фиксированной длины. Данный алгоритм

является модификацией алгоритма RNGLR[5], приспособленной к распознаванию регулярных основ для редукций. Основная идея здесь заключается в том, что генерируемый парсер сохраняет представление о регулярных правых частях грамматик и пользуется им при поиске левых концов основ для свёрток.

В классическом RNGLR, когда анализатор достигает состояния, в котором можно применить редукцию для правила фиксированной длиной *l* правой части, он берёт вершину GSS *v*, соответствующую этому состоянию, и просто ищет в GSS все пути длины *l*, начинающиеся из *v*.

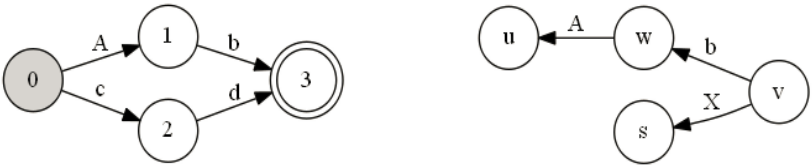

Рисунок 1: Пример конечного автомата - правой части правила *p* (слева) и фрагмента GSS (справа). Путь [*v, w, u*] соответствует основе для свёртки по правилу *p*

Когда правые части правил исходной грамматики представлены конечными автоматами, путь в GSS определяется как соответствующий основе для свёртки по правилу *p*, если обращенная последовательность грамматических символов этого пути допускается правой частью *p* (см. Рис. 1).

Сравнение выводов с алгоритмами синтаксического анализа, не сохраняющими EBNF

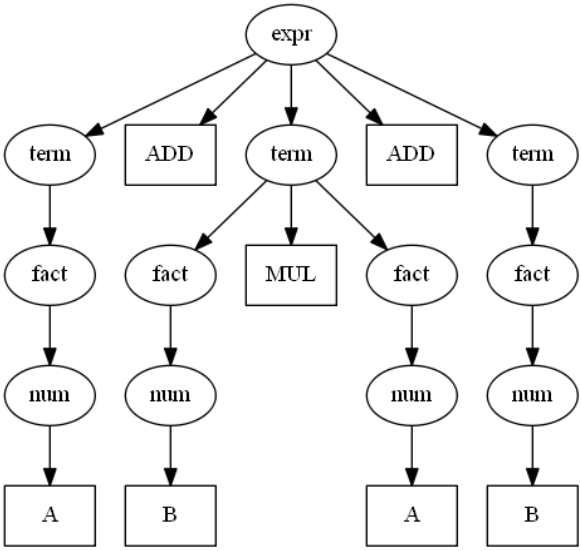

Рисунок 2: Вывод сгенерированного синтаксического анализатора, сохраняющего EBNF

Рассмотрим простую грамматику арифметических выражений (определена в нотации Yard[6]):

```
[<Start>]

expr : term ((ADD|MIN) term)*
term : fact ((MUL|DIV) fact)*
fact : num | LBR expr RBR
num : A | B
```

Данное описание грамматики использует средства регулярных выражений: звезду Клини и объединение.

Возьмем строку *A ADD B MUL A ADD B* языка, описываемого данной грамматикой, и сравним выводы синтаксических анализаторов, сохраняющего форму Бэкуса-Наура (см. Рис.2) и несохраняющего её (см. Рис. 3). Второй анализатор был сгенерирован из грамматики, трансформированной с помощью инструментов фреймворка YaccConstructor[7]. Видно, что вывод первого анализатора выгодно отличается от вывода второго отсутствием узлов, не относящихся к терминологии исходной грамматики, что может быть важно для пользователя генератора синтаксических анализаторов, а также своими размерами.

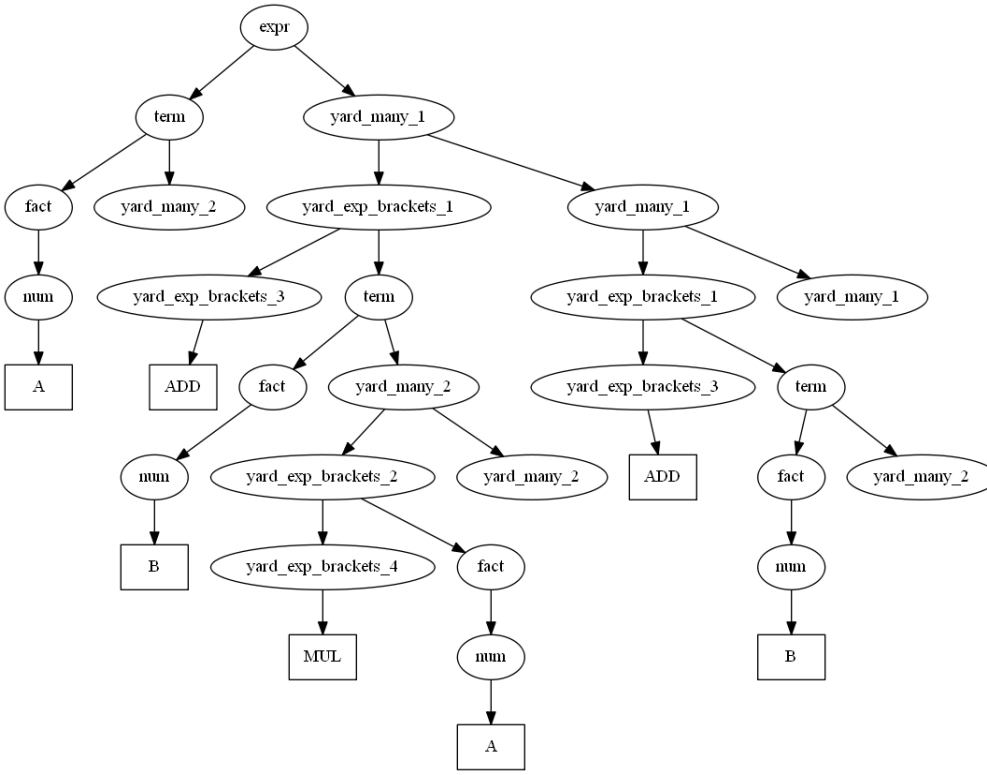

Рисунок 3: Вывод синтаксического анализатора, не сохраняющего EBNF

Традиционно пользователи генераторов синтаксических анализаторов пишут код с определением AST, экземпляр которого они хотят получить в виде вывода порождённого парсера, для чего они снабжают исходную грамматику атрибутами с вычислением узлов AST. Сохранение исходной формы грамматики в генерации парсеров позволяет автоматически генерировать такое определение и создавать парсеры, вывод которых будет в форме этого определения. Например, для грамматики арифметических выражений, написанной выше, генерируется такое определение в коде на языке F#:

```
type TExpr = Expr of TTerm * (Choice<Token, Token> *
TTerm) list
and TTerm = Term of TFact * (Choice<Token, Token> *
TFact) list
and TFact = Fact of Choice<TNum, Token * TExpr *
Token>
and TNum = Num of Choice<Token, Token>
```

В итоге пользователь получает вывод в понятной ему форме и он может работать с AST так, как если бы он сам создал определение AST. Например, создать интерпретатор синтаксического дерева арифметического выражения, созданного по сгенерированным типам, приведённым выше (код функции evaluateTerm опущен в силу своей аналогичности коду для evaluateExpr):

```
let rec evaluateExpr = function
    | Expr (leftTerm, rightPart) ->
        let f leftValue (sign, term) =
            let rightValue = evaluateTerm term
            match sign with
            | Choice1Of2 _ -> leftValue + rightValue
            | Choice2Of2 _ -> leftValue - rightValue
        rightPart |> List.fold f (evaluateTerm
leftTerm)
```

```
and evaluateFact = function
    | Fact fact ->
        match fact with
        | Choice1Of2 (Num num) ->
            getValue num
        | Choice2Of2 (_, expr, _) ->
            evaluateExpr expr
```

Приведённый код демонстрирует естественность формы получаемого вывода, с которой удобно работать пользователю. Узел вывода имеет последовательность потомков, что соответствует конкретному правилу грамматики, его левой части и цепочки грамматических символов, допускаемой его правой частью. Вывод звезды Клини от некоторого выражения представляется списком выводов этого выражения, вывод объединения двух выражений - это специальная конструкция выбора вывода одного из выражений.

В действительности сохранение знания об исходном устройстве грамматики позволяет вычисление любых атрибутов, и создание вывода в терминах, заданных пользователем - частный случай использования данной возможности.

Заключение

В данной статье представлен метод конструирования GLR-парсеров, сохраняющих исходную форму грамматик, описанных с помощью EBNF. Метод работает с классом контекстно-свободных грамматик, позволяет получать вывод порождённых анализаторов в терминах, заданных пользователем, и даёт возможность вычислять атрибуты, заданные в описании грамматики.

На данный момент в проекте YaccConstructor выполнена реализация представленного метода, но, к сожалению, она обладает недостатком меньшей скорости распознавания порождаемых анализаторов в сравнении с анализаторами, порождаемыми генераторами, не сохраняющими исходную расширенную форму Бэкуса-Наура. Дальнейшим развитием

работы, представленной в этой статье, представляется улучшение данного метода по производительности и оптимизация его реализации.

Литература

1.Knuth, D.E.: On the translation of languages from left to right. Information and Control 8, 607–639 (1965)

2.Purdom, P.W., Brown, C.A.: Parsing extended LR(k) grammars. Acta Inf., 15(1981), 115-127.

3.Morimoto, S.I., Sassa, M.: Yet another generation of LALR parsers for regular right part grammars. Acta Informatica 37, 671–697 (2001)

4.Borsotti, A., Breveglieri, L.,Reghizzi, S. C., Morzenti, A.: Complexity of Extended vs. Classic LR Parsers. Springer, LNCS 8614, 77-89 (2014)

5.Scott, E., Johnstone, A.: Right Nulled GLR Parsers, 2006

6.Чемоданов И.С.: Генераторы синтаксических анализаторов для решения задач автоматизированного реинжиниринга программ, 2007. 37 с. // URL: http://recursive-ascent.googlecode.com/files/IlyaChemodanov_Yard.pdf

7.YaccConstructor. URL: https://github.com/YaccConstructor/YaccConstructor