Википедия
Свободная энциклопедия

Заглавная страница
Рубрикация
Указатель А — Я
Избранные статьи
Случайная статья
Текущие события

Участие
Сообщить об ошибке
Сообщество
Форум
Свежие правки
Новые страницы
Справка
Пожертвовать

Инструменты
Ссылки сюда
Связанные правки
Служебные страницы
Постоянная ссылка
Сведения о странице
Цитировать страницу

Печать/экспорт
Создать книгу
Скачать как PDF
Версия для печати

В других проектах
Элемент Викиданных

На других языках
English
فارسی
Français
日本語
Português
中文
Править ссылки

Грамматика, разбирающая выражение

Материал из Википедии — свободной энциклопедии [править | править код]

Текущая версия страницы пока не проверялась опытными участниками и может значительно отличаться от версии, проверенной 17 сентября 2013; проверки требуют 13 правок.

Грамматика, разбирающая выражение (РВ-грамматика) — тип аналитической формальной грамматики, описывающей формальный язык в терминах набора правил для распознавания строк языка. Грамматика, разбирающая выражение, в сущности, представляет собой синтаксический анализатор рекурсивного спуска в чисто схематической форме, которая выражает только синтаксис и не зависит от конкретной реализации или применения синтаксического анализатора. Грамматики, разбирающие выражение, похожи на регулярные выражения и на контекстно-свободные грамматики (КС-грамматики) в нотации Бэкуса-Наура, но имеют отличную от них интерпретацию.

В отличие от КС-грамматик, РВ-грамматики не могут быть неоднозначными: если строка разбирается, то существует ровно одно дерево разбора. Это делает РВ-грамматики пригодными для компьютерных языков, но не для естественных.

Содержание [скрыть]

Определение [править | править код]

Формально, грамматика, разбирающая выражение, состоит из:

- конечного множества *N нетерминальных символов*;
- конечного множества *Σ терминальных символов*, не пересекающегося с *N*;
- конечного множества *P правил вывода*;
- выражения e_S, называемого *начальным выражением*.

Каждое правило вывода из P имеет вид A ← e, где A — нетерминальный символ, а e — выражение разбора. Выражение разбора — это иерархическое выражение, похожее на регулярное выражение, которое строится следующим образом:

1. Атомарное выражение разбора состоит из:
   - любого терминального символа,
   - любого нетерминального символа, или
   - пустой строки ε.
2. Для данных выражений разбора e, e_1 и e_2, следующие операторы порождают новые выражения разбора:
   - Последовательность: e_1 e_2
   - Упорядоченный выбор: e_1 / e_2
   - Нуль или более: e*
   - Один или более: e+
   - Необязательно: e?
   - И-предикат: &e
   - НЕ-предикат: !e

Фундаментальное отличие РВ-грамматики от КС-грамматики заключается в том, что оператор выбора РВ-грамматики является *упорядоченным*. Если первая альтернатива срабатывает, то все последующие — *игнорируются*. Таким образом, упорядоченный выбор некоммутативен, в отличие от книжных определений контекстно-свободных грамматик и регулярных выражений. Упорядоченный выбор аналогичен мягкому оператору отсечения в некоторых логических языках программирования.

Вследствие этого, при преобразовании КС-грамматики напрямую в РВ-грамматику всякая неоднозначность устраняется детерминированным образом в пользу одного из возможных деревьев разбора. Аккуратно выбирая порядок указания грамматических альтернатив, программист может получить значительный контроль над выбором нужного дерева разбора.

Как и булевы контекстно-свободные грамматики, РВ-грамматики имеют предикаты И- и НЕ-. Они помогают и далее устранять неоднозначность, если переупорядочивание альтернатив не может задать желаемое дерево разбора.

Интерпретация выражений разбора [править | править код]

Каждый нетерминал в РВ-грамматике, по существу, представляет собой разбирающую функцию в анализаторе рекурсивным спуском, а соответствующее выражение разбора представляет собой «код» этой функции. Каждая разбирающая функция принимает на вход строку и выдаёт один из следующих результатов:

- успех, в случае которого функция может опционально передвинуть вперёд или «поглотить» один или несколько символов входной строки
- провал, в случае которого вход не поглощается.

Нетерминал может завершиться успешно без поглощения ввода, и это состояние отлично от провала.

Атомарное выражение разбора, состоящее из единственного терминала, завершается успешно, если первый символ входной строки с ним совпадает, и поглощает его. Иначе результат неуспешен. Атомарное выражение из пустой строки всегда завершается успешно без поглощения. Атомарное выражение, состоящее из нетерминала *A*, представляет собой рекурсивный вызов функции-нетерминала *A*.

Оператор последовательности e_1 e_2 сначала вызывает e_1 и, если e_1 выполняется успешно, далее вызывает e_2 от части строки, оставшейся непоглощённой e_1 и возвращает результат. Если e_1 или e_2 проваливается, то проваливается и оператор последовательности e_1 e_2.

Оператор выбора e_1 / e_2 сначала вызывает e_1 и, если e_1 успешно, возвращает её результат. Иначе, если e_1 проваливается, оператор выбора восстанавливает входную строку в состояние, предшествующее вызову e_1, и вызывает e_2, возвращая её результат.

Операторы нуль-или-более, один-или-более и необязательности поглощают соответственно нуль или более, одно или более, или нуль либо одно последовательное появление своего подвыражения *e*. В отличие от КС-грамматик и регулярных выражений, эти операторы всегда являются жадными, и поглощают столько входных экземпляров, сколько могут. (Регулярные выражения сначала действуют жадно, но затем в случае провала возвращаются в исходное состояние и пытаются найти более короткую последовательность). Например, выражение a* всегда поглотит все доступные символы a, а выражение (a* a) всегда провалится, поскольку после выполнения первой части a* не останется символов a для второй.

Наконец, И-предикат и НЕ-предикат реализуют синтаксические предикаты. Выражение &*e* вызывает подвыражение *e*, и возвращает успех, если *e* успешно, и провал в противном случае, но никогда не поглощает ввода. Аналогично, выражение !*e* срабатывает успешно, если *e* проваливается и проваливается, если *e* успешно, так же не поглощая ввода. Поскольку выражение *e* может представлять собой сколь угодно сложную конструкцию, вычисляемую «наперёд» без поглощения входной строки, эти предикаты предоставляют мощные синтаксические средства предварительного анализа и устранения неоднозначности.

Примеры [править | править код]

Следующая РВ-грамматика распознаёт математические формулы с четырьмя действиями над неотрицательными целыми.

```
Value   ← [0-9]+ / '(' Expr ')'
Product ← Value (('*' / '/') Value)*
Sum     ← Product (('+' / '-') Product)*
Expr    ← Sum
```

В примере выше терминальными символами являются символы текста, представленные символами в одинарных кавычках, например `'('` и `')'`. Диапазон `[0-9]` представляет собой сокращённую запись десяти символов, обозначающих цифры от 0 до 9. (Это тот же синтаксис, что и для регулярных выражений). Нетерминальными символами являются символы, для которых есть правила вывода: *Value*, *Product*, *Sum*, and *Expr*.

В примерах ниже нет кавычек для улучшения читаемости. Строчные буквы являются терминальными символами, а прописные курсивные — нетерминалы. Настоящие анализаторы РВ-грамматик требуют кавычек.

Выражение разбора **(a/b)*** соответствует и поглощает последовательности произвольной длины из a и b. Правило ***S*** ← **a *S*? b** описывает простой контекстно-свободный язык $\{a^n b^n : n \geq 1\}$. Следующая РВ-грамматика описывает классический не контекстно-свободный язык $\{a^n b^n c^n : n \geq 1\}$:

```
S ← &(A 'c') 'a'+ B !('a' / 'b' / 'c')
A ← 'a' A? 'b'
B ← 'b' B? 'c'
```

Следующее рекурсивное правило соответствует стандартному оператору if/then/else языка C таким образом, что необязательный блок else всегда соответствует наиболее внутреннему if. (В контекстно-свободной грамматике это привело бы к классической неоднозначности болтающегося else).

```
S ← 'if' C 'then' S 'else' S / 'if' C 'then' S
```

Выражение разбора **foo &(bar)** соответствует и поглощает текст «foo», но только если за ним следует текст «bar». Выражение разбора **foo !(bar)** поглощает текст «foo» только если за ним *не* следует «bar». Выражение **!(a+ b) a** принимает один символ «a», но только если он не является первым в последовательности a произвольной длины, за которой следует b.

Следующее рекурсивное правило соответствует вложенному комментарию языка Паскаль. Символы комментариев помещены в одинарные кавычки для отличия их от операторов РВГ.

```
Begin ← '(*'
End   ← '*)'
C     ← Begin N* End
N     ← C / (!Begin !End Z)
Z     ← любой одиночный символ
```

Реализация анализаторов РВ-грамматик [править | править код]

Любая РВ-грамматика может напрямую быть преобразована в анализатор рекурсивным спуском. Из-за неограниченной способности к предварительному анализу результирующий парсер может работать, в худшем случае, экспоненциальное время.

Запоминая результат промежуточных шагов анализа и удостоверяясь в том, что каждая разбирающая функция вызывается не более одного раза для данной позиции входных данных, можно преобразовать любую РВ-грамматику в *packrat-парсер*, который всегда работает линейное время за счёт существенного увеличения затрат памяти.

Packrat-парсер — разновидность анализатора, работающего схожим с рекурсивным спуском методом, за исключением того, что при анализе он запоминает промежуточные результаты всех вызовов взаимно рекурсивных функций анализа. Из-за этого packrat-парсер способен анализировать множество контекстно-свободных грамматик и *любую* РВ-грамматику (включая некоторые, порождающие не контекстно-свободные языки) в линейное время[1].

Также возможно построить LL-анализатор и LR-анализатор для РВ-грамматик, но способность к неограниченному предварительному анализу в этом случае теряется.

Достоинства [править | править код]

РВ-грамматики хорошо заменяют регулярные выражения, потому что они строго мощнее. Например, регулярное выражение существенно неспособно найти соответствующие пары скобок, поскольку оно нерекурсивно, в отличие от РВ-грамматики.

Любая РВ-грамматика может быть анализирована за линейное время, используя packrat-анализатор, как описано выше.

Анализаторы для языков, представленных КС-грамматиками, такие как LR-анализаторы, требуют особого шага лексического анализа, который разбивает входные данные в соответствии с пробелами, пунктуацией и так далее. Это необходимо, так как эти анализаторы используют *предварительный анализ* для обработки некоторых КС-грамматик в линейное время. РВ-грамматики не требуют отдельного шага лексического анализа, а правила для него могут быть заложены вместе с другими правилами грамматики.

Многие КС-грамматики содержат существенные неоднозначности, даже когда они должны описывать однозначные языки. Проблема «висячего else» языков C, C++ и Java является одним из примеров этого явления. Эти проблемы часто разрешаются применением внешнего по отношению к грамматике правила. В РВ-грамматике эти неоднозначности никогда не возникают вследствие приоритизации.

Недостатки [править | править код]

Потребление памяти [править | править код]

Анализ РВ-грамматики обычно производится packrat-парсером, который запоминает лишние шаги анализа. Такой анализ требует хранения данных пропорционально длине входных данных, в отличие от глубины дерева разбора для LR-анализаторов. Это существенный прирост во многих областях: например, программный код, написанный человеком, как правило имеет практически константную глубину вложенности независимо от длины программы — выражения с глубиной свыше некоторой величины обычно подвергаются рефакторингу.

Для некоторых грамматик и некоторых входных данных, глубина дерева разбора может быть пропорциональна длине ввода, поэтому для оценки, не учитывающей этот показатель, packrat-анализатор может казаться не хуже LR-анализатора. Это похоже на ситуацию с алгоритмами графов: Беллман-Форд и Флойд-Уоршелл имеют одно время выполнения ($O(|V|^3)$) если учитывать только число вершин. Однако более точный анализ, учитывающий число рёбер, показывает время выполнения алгоритма Беллмана-Форда $O(|V| * |E|)$, что всего лишь квадратично к размеру входа, а не кубично.

Непрямая левая рекурсия [править | править код]

РВ-грамматики не могут содержать леворекурсивных правил, которые содержат вызов самих себя без продвижения по строке. Например, в вышеописанной арифметической грамматике хотелось бы передвинуть некоторые правила, чтобы приоритет произведения и суммы можно было выразить одной строкой:

```
Value   ← [0-9.]+ / '(' Expr ')'
Product ← Expr (('*' / '/') Expr)*
Sum     ← Expr (('+' / '-') Expr)*
Expr    ← Product / Sum / Value
```

Тут проблема в том, что для того, чтобы получить срабатывание для `Expr`, необходимо проверить, срабатывает ли `Product`, а чтобы проверить `Product`, нужно сначала проверить `Expr`. А это невозможно.

Однако, леворекурсивные правила всегда можно переписать, ликвидируя левую рекурсию. Например, леворекурсивное правило может повторять некоторое выражение неопределённо долго, как в правиле КС-грамматики:

```
string-of-a ← string-of-a 'a' | 'a'
```

Это можно переписать в РВ-грамматике, используя оператор +:

```
string-of-a ← 'a'+
```

С определёнными изменениями можно заставить обычный packrat-парсер поддерживать прямую левую рекурсию[1][2][3].

Однако, процесс переписывания косвенных леворекурсивных правил затруднён, особенно когда имеют место семантические действия. Хотя теоретически это и возможно, не существует анализатора РВ-грамматики, поддерживающего косвенную левую рекурсию, в то время как её поддерживают все GLR-анализаторы.

Незаметные ошибки в грамматике [править | править код]

Чтобы выразить грамматику в виде РВ-грамматики, её автор должен преобразовать все экземпляры недетерминированного выбора в упорядоченный. К несчастью, этот процесс связан с ошибками, и часто в результате получаются грамматики, неверно анализирующие некоторые входные данные.

Выразительность [править | править код]

Packrat-парсеры не могут анализировать некоторые однозначные грамматики, например следующую[4]:

```
S ← 'x' S 'x' | 'x'
```

Развитость [править | править код]

РВ-грамматики новы, и не получили широкого распространения. Регулярные выражения и КС-грамматики, напротив, существуют уже десятилетия, программный код, их анализирующий, совершенствовался и оптимизировался, а программисты имеют опыт их применения.

Ссылки [править | править код]

- *Medeiros, Sérgio*; *Ierusalimschy, Roberto* (2008). "A parsing machine for PEGs". *Proc. of the 2008 symposium on Dynamic languages*: article #2, ACM. DOI:10.1145/1408681.1408683. ISBN 978-1-60558-270-2.

Примечания [править | править код]

1. ↑ 1 2 *Ford, Bryan* Packrat Parsing: a Practical Linear-Time Algorithm with Backtracking. Massachusetts Institute of Technology (September 2002). Дата обращения 27 июля 2007. Архивировано 2 апреля 2012 года.
2. ↑ *Alessandro Warth, James R. Douglass, Todd Millstein.* Packrat Parsers Can Support Left Recursion (неопр.). — Viewpoints Research Institute, 2008. — January.
3. ↑ *Ruedi Steinmann.* Handling Left Recursion in Packrat Parsers (неопр.). — 2009. — March. Архивировано 6 июля 2011 года. Архивная копия от 6 июля 2011 на Wayback Machine
4. ↑ *Bryan Ford.* Functional Pearl: Packrat Parsing: Simple, Powerful, Lazy, Linear Time (англ.) : journal. — 2002.

Категория: Формальные грамматики

Эта страница в последний раз была отредактирована 9 июля 2019 в 03:50.