Санкт-Петербургский государственный университет

Кафедра системного программирования

Боровков Данила Викторович

Монадические парсер-комбинаторы с поддержкой левой рекурсии

Курсовая работа

Научный руководитель:
к. ф-м. н., доцент Булычев Д. Ю.

Санкт-Петербург
2017

Оглавление

1. Введение

1.1. Парсер-комбинаторы

Синтаксический анализ — это процесс сопоставления входной последовательности лексем дерева разбора в некоторой грамматике. Один из подходов к синтаксическому анализу заключается в применении парсер-комбинаторов — функций высшего порядка, строящих парсеры из других парсеров. Это очень удобный способ построения парсеров, сочетающий в себе простоту и композициональность.

Стандартные парсер-комбинатор — это alt и seq. На интутивном уровне alt представляет из себя пасер, который принимает два парсера как аргументы и разбирает строку либо с помощью первого, либо с помощью второго. Seq тоже получает два парсера как аргументы, но применяет их поледовательно, то есть сначала совершаем синтаксический разбор с помощью первого парсера, а потом с помощью второго, начиная с того места, на котором закончил первый. Использование этих парсеров можно увидеть на примере.

Программа 1: pali.ml

```
let rec pali x = alt (seq (terminal 'a') (seq pali (terminal 'a')))
                     (alt (seq (terminal 'b') (seq pali (terminal 'b')))
                          (alt (seq (terminal 'c') (seq pali (terminal 'c')))
                               empty)) x
 in seq pali eof
```

Здесь представлена грамматика, принимающая любой палиндром состоящий четного числа символов a, b или c, то есть S —> aSa | bSb | cSc | eps. В этом примере также представлены парсер-комбинаторы terminal c, eof и empty, которые являются в некотором смысле конечными парсерами, которые не задействуют другие парсеры для синтаксического анализа, а действуют исключительно сами. terminal c разбирает один символ, то есть, если текущий символ входной строки равен c, то terminal c успешно разбирает его и успешно завершается с продвижением по строке. Иначе он просто завершается без продвижения по строке. Eof и empty работают в похожей манере, но успешно заверша-

ются в случае конца строки и в любом случае соответственно. Хотелось бы отметить, что определение успешного завершения зависит от реализации. В данном примере я не привязываюсь ни к какой реализации, поэтому использую общие слова.

1.2. Левая рекурсия

Леворекурсивная грамматика — это такая грамматика, в которой из некоторого нетерминала S существует невырожденный вывод вида S —> Sx, где x — последовательность нетерминалов и териналов.

Левая рекурсия тоже бывает разная: если вывод из определения леворекурсивной граматики получился длиной 1, то левая рекурсия называется прямой. Грамматики с такой левой рекурсией поддержать относительно просто, и существует большое количество решений по поддержке только прямой рекурсии. С непрямой левой рекурсией, где вывод нетерминала из него самого получается длиной больше 1, бороться сложнее, потому что её сложнее найти и обозначить.
Левая рекурсия — это классическая проблема парсер-комбинаторов и синтаксического анализа в целом. Существует множество публикаций на эту тему, некоторые из которых будут рассмотрены далее.

2. Постановка задачи

Целью данной работы является реализация комбинаторов в рамках библиотеки Ostap с поддержкой левой рекурсии.
Были запланированы следующие задачи:

- изучить подход к синтаксическому анализу с использованием парсер-комбинаторов;
- изучить существующие решения поддержки леворекурсивных грамматик;
- реализовать поддерживающие левую рекурсию комбинаторы в рамках бибилотеки Ostap

3. Обзор предметной области

3.1. Ostap

Ostap[3] — это библиотека парсер-комбинаторов, разработанная Д.Ю. Булычевым в 2009 году. Реализованное там синтаксическое расширение для Objective Caml предоставляет пользователю широкие возможности по удобному представлению грамматик и дерева вывода.

На примере представлена реализация комбинаторов alt и seq в Ostap. Разберемся, что будет не так в случае классической леворекурсивной грамматики, которая также представлена на примере ниже.

Программа 2: grammar.ml

```
let alt x y s =
    match x s with
    | Failed x ->
        (match y s with
        | Failed y -> Failed (join x y)
        | Parsed (ok, err) -> Parsed (ok, join x err)
        )
    | x -> x

let seq x y s =
    match x s with
    | Parsed ((b, s'), err) ->
        (match y b s' with
        | Failed  x     -> Failed (join err x)
        | Parsed (s, e) -> Parsed (s, join err e)
        )
    | x -> cast x

let rec S x = alt (seq S (terminal 'a')) (terminal 'a') x
    in seq S eof
```

Первым делом мы попадаем в alt и просто передаем строку его первому аргументу. Им является seq, он также просто вызывает свой первый аргумент с этой же строкой, которым опять является alt с такими же аргументами, как и первый. Мы не обработали ни одного символа и опять вызываем такой же alt с такой же строкой, как и в первый раз. Таким образом, мы зациклились. Значит проблема поддержки леворекурсивных грамматик в Ostap не решена, даже в случае прямой левой

рекурсии.

Также в нынешней реализации парсер-комбинатора alt второй аргумент вызывается только в случае неудачи первого. Таким образом, если есть несколько деревьев разбора строки, мы находим только одно.

3.2. Meerkat

Meerkat[1] — это библиотека парсер-комбинаторов, в основе которой лежит работа Джонсона[4] 1995 года по мемоизации распознавателей. Спустя 20 лет после публикации были наконец реализованы идеи Джонсона в рамках библиотеки Meerkat. Библиотека написана на языке Scala, поэтому там пришлось использовать fixed-point combinator для рекурсивных определений таким образом:

Программа 3: fix.ml

```
1 val A = fix(A => rule("A",
2     seq(A, terminal("a")), terminal("a")))
```

где rule — аналог alt в Meerkat. Первый аргумент rule — это имя нетерминала, которому соответствует rule. Комбинаторы написаны в Continuation Passing Style — это стиль программирования, использующий механизм продолжений для передачи управления. В Meerkat используются продолжения, имеющие тип Int -> Unit, то есть принимают позицию во входной строке и совершают какое-то действие, не возвращая ничего. Парсер при это имеет тип Int -> (Int -> Unit) -> Unit, то есть принимают позицию, с которой надо парсить, и продолжение, которое нужно исполнить по завершению, и совершает какое-то действие, не возвращая ничего. Для борьбы с левой рекурсией была использована мемоизация. Опишем в общих словах, в чем заключается мемоизация. Для каждого нетерминала создается таблица. Индексами являются позициями в строке. В ней хранится результаты, полученные в предыдущие посещения этого нетерминала. В случае разбора нетерминалом позиции i, мы смотрим элемент таблицы под индексом i, и, если там что-то есть, то мы просто берем этот элемент как результат. Это в общих словах как работает мемоизация. Данный способ поддер-

живает все контекстно свободные грамматики. В статье было показано, что для грамматики S —> SSS | SS | b этот метод работает за кубическое от длины строки время. Для грамматики Java, этот способ работает за время близкое к линейному.

3.3. Growing the seed

В своей статье[2] Warth описывает мемоизацию, которую он предлагает использовать в парсерах Packrat для поддержки грамматик с леворекурсивными правилами. Его способ заключается в том, что для каждого нетерминала создается так называемое seed, к которому происходит обращение при каждом посещении нетерминала. При первом обращении seed инициализируется, и происходит завершение с отрицательным результатом. Затем при каждом следующем посещении оно обновляется и возвращается. Но у данного подхода были выявлены существенные недостатки в статье Тратта[6] — в случае правой рекурсии в леворекурсивных правилах некорректно строится дерево разбора (правая рекурсия - это то же самое, что и левая, но в выводе нетерминал должен появиться на правом конце вывода, а не на левом), а именно, неправильно строится дерево в плане ассоциативности. Траттом был предложен способ по исправлению этой проблемы, но только в случае прямой правой рекурсии в левой рекурсии, но с непрямой не удалось научиться бороться. Это существенный минус, поэтому было решено не использовать данный способ.

3.4. Мемоизация количества обращений

В мемозации, описанной в статье Фроста[5], используется базовый способ борьбы с левой рекурсией — ограничение количества обращений к нетерминалу в ходе левой рекурсии. Рассмотрим этот метод на примере прямой левой рекурсии S —> Sx, где x — это некоторая последовательность терминалов и нетерминалов, из которых не выводится пустая строка. Если мы обратились k раз подряд к этому правилу, то входная строка должна иметь вид Sxx..xx. Таким образом можно

ограничить количество рекурсивных вызовов этого правила отношением ”длина входной строки”/”минимальная длина вывода из x”. В случае непрямой левой рекурсии алгоритм гораздо сложнее, но опять-таки все зависит от длины входной строки. Но использовать длину строки не всегда возможно. Например, может быть просто входной поток символов, который нужно обрабатывать, конца у него нет, и длины строки у него нет. Таким образом этот подход нам не подходит.

4. Реализация

Для реализации комбинаторов в рамках библиотеки Ostap был выбран способ Джонсона реализованный в библиотеке Meerkat. Опишем поконкретнее как именно выглядит мемоизация. В моей реализации на OCaml используются продолжения [Char] -> Unit и парсеры имеют тип [Char] -> ([Char] -> Unit) -> Unit.

Парсер одного терминала выглядит таким образом:

Программа 4: cpsterminal.ml

```
let cpsterminal c =
function
    | s::ss when s = c -> success ss
    | ss -> failure ss
```

где success ss — это функция, принимающая продолжение и вызывающая его на строке ss. А failure ss — принимает продолжение и не делает ничего. Таким образом, при успешном разборе будет вызвано продолжение.

Seq выглядит таким образом:

Программа 5: cpsseq.ml

```
let cpsseq a b =
    function s ->
        (function k ->
            a s (function t -> b t k))
```

То есть cpsseq a b принимает строку и продолжение и вызывает первый парсер на этой строке с новый продолжением. А второй парсер записывается как раз в это продолжение, чтобы сработать при успешном завершении парсера a.

Теперь посмотрим на самую сложную часть — реализацию комбинатора alt. Для каждого нетерминала создается три таблицы:

- Hash — хэш-таблица для результатов применения с индексами — позициями применения;

- Ks — список продолжений, которые вызывались на этом нетерминале;
- Rs — список позиций, на которых вызывались продолжения из таблицы Ks

При вызове cpsalt a b co строкой s и продолжением k происходит обращение к Hash по индексу s. Если там что-то есть, то вызываем это с продолжением k. Если пусто, то записываем туда новый элемент. Он создается таким образом. Это функция memoresult, которая принимает парсер примененный к строке и продолжение. При ее частичном вызове (то есть при передаче частично примененного парсера), создаются Ks и Rs, которые изначально пустые. При передаче частично примененной memoresult продолжения k происходит вот что: если Ks пустая, то мы добавляем k в Ks и вызываем частично примененный парсер с продолжением, которое при его запуске на позиции s заносит s в Rs и вызывает все продолжения из Ks на s. Если Ks не пустая, то частично примеренный парсер мы не запускаем, то есть каждый парсер на каждой позиции будет вызван не более одного раза. Мы опять добавляем продолжение в Ks и вызываем его на всех позициях добавленных в Rs.

Для хэш-таблицы был использован модуль Hashtbl, который предоставляет изменяемую таблицу. Для таблиц было решено использовать ссылки на список, чтобы их можно было модифицировать. Для разыменования ссылки используется оператор !.

Для единичного создания таблиц у каждого нетерминала используется оператор неподвижной точки. В Objective Caml, в отличие от Scala, есть конструкция let rec для построения рекурсивных опеределений, но fix-point также используется для разграничения, чтобы было понятно, что повторное применение одного и того же alt — это повторное применение, а значит нужно использовать уже созданную таблицу. Для этого необходимо использовать конструкцию lazy—force

внутри fix-point, потому что сам по себе fix-point ничего не сделает.

Программа 6: fixcomb.ml

```
let fix f =
    let rec p = lazy ((f (function t -> force p t)))
    in force p
```

Оператор lazy показывает, что его аргумент надо вычислять только один раз, и, встраивая его в fix-point таким образом, как показано ниже, мы будем создавать таблицы только один раз для каждого нетерминала. Решение было протестировано на многих грамматиках, таких как S —> Sa | a, S —> S+S | a, грамматика палиндромов, показанная выше, грамматика математических выражений : A —> M | A+M; M —> P | M*P; P —> a | b | c.

5. Заключение

В ходе работы над этой курсовой были изучены статьи с решениями по поддержке леворекурсивных грамматик: статьи Фроста, Тратта, Варта, Джонсона и по Миркату. Было решено использовать подход, описанный в статье по Миркату, остальные не подошли по разным причинам. Подход был реализован на языке Objective Caml. Реализация была протестирована на многих грамматиках.

Список литературы

[1] A. Izmaylova A. Afoozeh, van der Storm T. Practical, General Parser Combinators // PEPM '16 Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation. — 2016. — P. 1–12.

[2] A. Warth J. R. Douglass, Millstein T. Packrat Parsers Can Support Left Recursion // Partial Evaluation and Semantics-based Program Manipulation, PEPM '08. — 2008. — P. 103–110.

[3] Boulytchev D. Ostap: Parser Combinator Library and Syntax Extension for Objective Caml. — 2009.

[4] Johnson M. Memoization in Top-Down Parsing // Computational Linguistics. — 1995. — Vol. 21. — P. 405–417.

[5] R. A. Frost R. Hafiz, Callaghan P. Parser Combinators for Ambiguous Left-Recursive Grammars // Practical Aspects of Declarative Languages, PADL'08. — 2008.

[6] Tratt L. Direct Left-Recursive Parsing Expression Grammars // Technical Report EIS-10-01, Middlesex University. — 2010.