Санкт-Петербургский государственный университет

Кафедра системного программирования

Боровков Данила Викторович

Левая рекурсия в монадических парсер-комбинаторах

Выпускная квалификационная работа бакалавра

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

Рецензент:
Подкопаев А. В.

Санкт-Петербург
2018

SAINT-PETERSBURG STATE UNIVERSITY

Department of Software Engineering

Borovkov Danila

Left recursion in monadic parser-combinators

Graduation Thesis

Scientific supervisor:
Ph.D Dmitry Boulytchev

Reviewer:
Anton Podkopaev

Saint-Petersburg
2018

Оглавление

Введение

В настоящее время трансляция программного кода играет важную роль в сфере программирования. Процесс трансляции [11] заключается в преобразовании кода программы, написанного на языке программирования высокого уровня, в машинный код или в текст на другом языке программирования. Одним из этапов трансляции традиционно является синтаксический анализ – проверка соответствия анализируемого текста некоторой грамматике и в построении дерева синтаксического анализа. Программное обеспечение, выполняющее синтаксический анализ, называется синтаксическим анализатором или парсером.

Исследована связь парсеров с таким математическим объектом, как монада [4] [10], позволяющим формально представить вычислительный процесс и его побочные эффекты. Было показано, что парсеры могут быть реализованы в виде монад, что предоставляет возможность построить математическую модель процесса синтаксического анализа. Помимо этого, сами монадические парсер-комбинаторы удобны для использования: например, они позволяют описывать синтаксический анализ контекстно-зависимых языков.

Одним из самых простых и удобных подходов к построению синтаксических анализаторов по входной грамматике являются парсер-комбинаторы [4]. Парсер-комбинатор – это функция высшего порядка, принимающая парсеры в качестве аргументов, конструирующая из них другой парсер и возвращающая его в качестве результата. Таким образом этот подход заключается в создании набора элементарных парсеров и нескольких парсер-комбинаторов, которые строят парсер из других парсеров, либо созданных с помощью парсер-комбинаторов, либо элементарных.

Однако такой подход не решает классические проблемы этой области. Во-первых, проблемой является использование стратегии longest match first по умолчанию, которая заключается в конструировании парсеров таким образом, чтобы в процессе синтаксического анализа при появлении альтернативных результатов в первую очередь рассматри-

вать самый длинный из них. Во-вторых, еще одной проблемой является левая рекурсия в грамматике анализируемого языка, которая приводит к зацикливанию алгоритмов синтаксического анализа. Поэтому эти алгоритмы предполагают, что, либо пользователь будет использовать грамматику без леворекурсивных правил, что бывает обременительно, либо он будет использовать алгоритм по переводу грамматики в форму без леворекурсивных правил, что требует дополнительных вычислений и дополнительных библиотек. В рамках данной работы было решено разработать библиотеку парсер-комбинаторов, не ограничивающую вид входной грамматики.

Для того, чтобы не разрабатывать библиотеку "с нуля", было решено использовать существующую библиотеку парсер-комбинаторов Ostap [3]: во-первых, у этой библиотеки имеются указанные выше недостатки; во-вторых, она использует монадические парсер-комбинаторы, которые являются объектом данного исследования. Кроме самих парсер-комбинаторов в рамках этой библиотеки было реализовано синтаксическое расширение для языка OCaml, позволяющее легко совершать построение парсеров с помощью понятных и кратких конструкций предметно-ориентированного языка похожего на язык описания грамматик Бэкуса-Наура. Преобразование этих конструкций в код на OCaml обеспечивается с помощью парсер-комбинаторов, то есть выразительность конструкций расширения непосредственно зависит от выразительности парсер-комбинаторов, и если парсер-комбинаторы имеют проблему левой рекурсии, то и пользователь расширения не сможет задать парсер с леворекурсивной входной грамматикой.

Перейдем к описанию цели и задач данной работы.

Постановка задачи

Целью данной работы является доработка библиотеки парсер-комбинаторов Ostap с добавлением возможности использования входных грамматик с леворекурсивными правилами и отказом от ограничения longest match first.

В связи с этим были поставлены следующие задачи:

- обзор существующих решений и анализ предметной области;
- разработка алгоритма монадических парсер-комбинаторов;
- интеграция разработанных парсер-комбинаторов в библиотеку Ostap;
- апробация доработанной библиотеки.

Глава 1. Обзор предметной области

Монады и парсеры

Монада – это паттерн, позволяющий описывать вычисления, происходящие над выбранным типом данных. Помимо результата вычисления монада позволяет описывать в своей структуре побочные эффекты и контекст вычисления и разделять само вычисление с обработкой побочных эффектов, что повышает композициональность программного кода. Для комбинирования вычислений монада обладает методами result и bind. Метод result позволяет создавать монаду с некоторым начальным контекстом по данным, которые передаются методу result как аргумент. Метод bind описывает изменение контекста в процессе последовательных вычислений. В строках 4 и 6 листинга 3 показаны сигнатуры этих методов для монады Parser. Однако не каждый объект с такими методами является монадой – для этого необходимо удовлетворять условиям, представленными в листинге 1.

Листинг 1: Законы монады

```
1 (result a) bind λ(b. n) = n[a/b]
2 m bind result = m
3 m bind λ(a. n bind λ(b. o)) = (m bind λ(a. n)) bind λ(b. o)
```

Первое условие говорит о том, что применение метода bind к монаде, полученной с помощью result, эквивалентно применению функции n к аргументу result. Второе условие показывает, что метод result является правым нейтральным по отношению к методу bind. Третье условие показывает, что bind ассоциативен.

У монады выделяют частный случай, называемый монадой плюс, который предоставляет дополнительные возможности комбинирования монад для описания вычислительного процесса. Для того, чтобы монада была монадой плюс, у нее должны быть методы plus и zero, сигнатуры который можно увидеть в листинге 4. Метод plus показывает каким образом можно комбинировать две монады так, чтобы в результате получалась одна монада. Метод zero просто возвращает монаду.

Эти два метода должны соответствовать условиям, которые показаны в листинге 2.

Листинг 2: Законы монады плюс

```
1 zero plus n = n
2 m plus zero = m
3 m plus (n plus o) = (m plus n) plus o
4 zero bind k = zero
5 m bind λ(a. zero) = zero
6 (m plus n) bind k = (m bind k) plus (n bind k)
```

Первые два условия показывают, что монада zero нейтральна по отношению к методу plus. Третье условие показывает, что plus ассоциативен. Четвертое и пятое условие показывает, что результатом bind является zero, если его левый аргумент является zero или если его правый аргумент возвращает zero. Шестое условие – условие дистрибутивности, которое связывает методы bind и plus.

Монады являются очень удобным средством для представления процесса синтаксического анализа [4], позволяющие не только описывать контекстно-свободные языки, но и рассматривать теоретическую модель процесса. В листинге 3 показано, как парсеры могут быть реализованы в виде монады.

Листинг 3: Парсеры как монады

```
1 type Parser a = String -> [(a,String)]
2 instance Monad Parser where
3 -- result :: a -> Parser a
4 result v = λ inp. [(v,inp)]
5 -- bind :: Parser a -> (a -> Parser b) -> Parser b
6 p `bind f = λ inp. concat [f v out | (v,out) <- p inp]
```

В данном примере представлен парсер, принимающий на вход строку и возвращающий результат, который является списком для того, чтобы у синтаксического анализа могло быть несколько вариантов. Список в строке 1 состоит из пар, где первый элемент – это суффикс строки, который удалось проанализировать, а второй элемент – это синтаксическое дерево. Для парсера такого типа в строках 4 и 6 реализованы

монадические методы result и bind. Эти методы должны удовлетворять законам монады, которые были показаны в листинге 1, что было доказано в работе [10].

Также парсер из листинга 3 может быть легко расширен до монады плюс, как показано в листинге 4.

Листинг 4: Парсеры как монады плюс

```
1 instance Monad0Plus Parser where
2 -- zero :: Parser a
3 zero = λ inp. []
4 -- (plus) :: Parser a -> Parser a -> Parser a
5 p 'plus' q = λ inp. (p inp ++ q inp)
```

Методы zero и plus также удовлетворяют монадическим законам [10], представленным в листинге 2.

Реализованные выше методы plus и bind – это парсер-комбинаторы alt и seq соответственно, имеющие следующий смысл: alt реализует выбор между двумя парсерами, а seq – последовательный процесс синтаксического анализа с передачей результата первого парсера второму. В дальнейшем, это передаваемое значение будет называться монадическим. Эти парсер-комбинаторы удобно рассматривать в связи с аналогичным форматом записи граматик в форме Бэкуса-Наура.

В 5 строке листинга 3 result x – это простейший парсер, возвращающий x в качестве результата, а zero в 3 строке листинга 4 – это парсер, не возвращающий ничего.

Монадическое представление парсеров обладает следующими преимуществами: метод bind позволяет обойтись без обработки пар из результатов, возникающих в других реализациях последовательного синтаксического анализа, а также такое представление позволяет использовать сокращенный синтаксис записи монад. Дополнительно, в работе [4] монада парсера была представлена в виде композиции монады недетерминизма и монады состояния, что способствует разделению реализации парсера на реализации соответствующих монад и переиспользованию их.

Подходы к реализации парсер-комбинаторов

Для реализации парсер-комбинаторов было рассмотрено несколько существующих алгоритмов, которые описаны ниже.

В статье [2] был описан алгоритм, который заключается в мемоизации состояния синтаксического анализа для каждого нетерминала грамматики анализируемого текста. В статье [7] у данного подхода были выявлены существенные недостатки, связанные с некорректной ассоциативностью в случае праворекурсивных правил. Также было предложено частично решение выявленных проблем, однако не для всех случаев.

В статье [8] был описан алгоритм мемоизации, которая основана на подсчете количества обращений с дальнейшим ограничением его функцией от длины строки. Однако эта информация может быть недоступна на момент синтаксического анализа, что, вместе с нелучшей производительностью, является серьезным недостатком этого алгоритма.

При реализации бибилотеки парсер-комбинаторов было решено использовать алгоритм библиотеки Meerkat [1], реализованный в стиле передачи продолжений, который заключается в следующем: у каждого нетерминала есть своя таблица мемоизации, к которой происходит обращение при его вызове. Единственность мемоизационной таблицы у каждого нетерминала обеспечивается с помощью функции makeCallWithName из библиотеки dsinfo [9]. В таблице хранится парсер, частично примененный к позиции в анализируемом тексте и переданный функции memoResult в качестве аргумента. Эта функция обеспечивает мемоизацию всех продолжений, с которыми вызывается данный нетерминал, и мемоизацию всех аргументов, которые были переданы продолжениям. При первом обращении происходит вызов частично примененного парсера. При дальнейших обращениях лишь обновляются таблицы и вызываются сохраненные продолжения, то есть на каждой позиции в строке каждый парсер вызывается не больше одного раза, что и является гарантией завершения процесса синтаксического анализа.

Библиотека Ostap

Несмотря на то, что парсер-комбинаторы привлекательная и выразительная техника для создания парсеров, она требует большого количества дополнительного программного кода, а пользователю удобнее работать с более классическим представлением грамматик, например, с формой Бэкуса-Наура. Возникает естественная идея о сочетании этих двух подходов, которая была реализована в виде библиотеки Ostap. Ostap – это библиотека монадических парсер-комбинаторов, разработанная на языке OCaml и предоставляющая предметно-ориентированный язык (далее DSL) для программирования парсеров, реализованый как синтаксическое расширения языка OCaml с помощью препроцессора Camlp5. Парсер-комбинаторы полностью абстрагированы от реализации дерева – пользователь может строить любое синтаксическое дерево, а также абстрагировано от входной последовательности лексем с помощью интерфейса stream из модуля Stream. Этот интерфейс, а также другие интерфейсы и модули, показаны на рисунке 1. Объект типа stream инкапсулирует лексический анализ: пользователь имеет возможность реализовать необходимые методы для лексического анализа в потомке stream и использовать лексемы с именами новых методов при описании парсеров. В листинге 5 можно увидеть на примере какие синтаксические конструкции предоставляет DSL Ostap для описания парсеров.

Листинг 5: Парсеры в синтаксисе Ostap

```
1  ostap (
2    primary:
3      c:IDENT {`N c} ;
4
5    exp[primary]:
6        e:exp[primary] -"+" p:primary {`E2 (e, p)}
7      | p:primary {`E1 p} ;
8
9    main: exp[primary] -EOF
10 )
```

Во-первых, рассмотрим конструкции IDENT и EOF в строках 3 и

9. Для корректной интерпретации пользователь должен реализовать в потомке stream методы getIDENT и getEOF, и при трансляции соответствующих конструкций будут вызваны эти методы. Строковые литералы также скрывают за собой вызов метода stream для разбора этой строки.

Во-вторых, в фигурных скобках пользователь может описать действия, строящие дерево разбора. При этом есть возможность использовать результаты каждого шага с помощью переменных, которые вводятся с помощью двоеточия перед парсером, результат которого необходимо положить в эту переменную.

В-третьих, рассмотрим конструкции DSL Ostap, использованные выше для описания парсеров. Вертикальная черта в строке 7 соответствует парсер-комбинатору plus, последовательная запись парсеров в строке 6 интрепретируется как bind. Описание остальных парсер-комбинаторов и синтаксиса DSL для их краткой записи можно посмотреть в [3].

В-четвертых, в этом примере показано, как реализовать параметризованный парсер: парсер exp параметризован парсером, разбирающим один элемент суммы. Тип параметра может быть любым, а не только парсером.

В-пятых, рассмотрим конструкцию ostap (expr). Она используется для обозначения того, что для описания выражения expr внутри скобок использован синтаксис DSL Ostap.

Помимо библиотеки парсер-комбинаторов и синтаксического расширения, в рамках библиотеки Ostap были реализованы модуль Matcher с реализацией интерфейса stream и модули Msg и Reason для описания ошибок синтаксического анализа. Это сделано для того, чтобы пользователю не приходилось самому реализовывать эти модуль, если ему нужна стандартная реализация.

На рисунке 1 показаны интерфейсы основных модулей библиотеки Ostap, необходимые для правильной работы парсеров из листинга 5. Первый из них – это Matcher.stream, который должен иметь методы getEOF и getIDENT для обработки лексем EOF и IDENT, использованных в листинге 5 и метод look для обработки строковых терминалов.

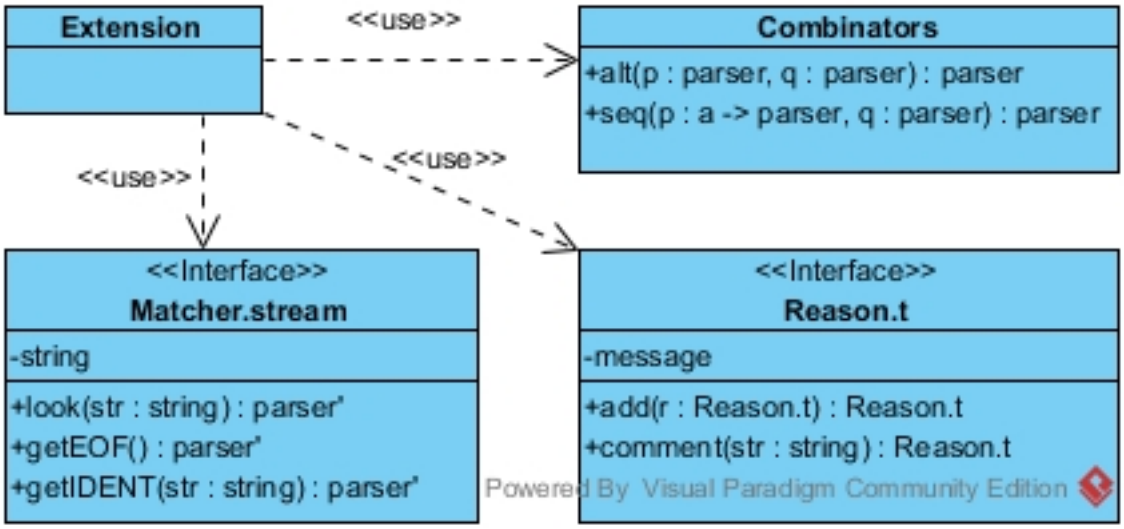

Рис. 1: Архитектура библиотеки Ostap

Также стрим содержит в себе входную строку. Еще один интерейс модуля, изображенный на рисунке 1 – это Reason.t. Назначение этого модуля заключается в хранении некоторой информации об ошибке. Для работы с этой информацией имеются методы add, который комбинируется две ошибки в одну, и метод comment, который добавляет информацию. Один из главных модулей – это модуль Combinators, который описывает элементарные парсеры и парсер-комбинаторы, основными из которых являются alt и seq. Все это используется в модуле расширения для генерации кода на OCaml по конструкциям DSL бибилотеки Ostap.

Глава 2. Монадические парсер-комбинаторы

В рамках данной работы были разработаны парсер-комбинаторы, использующие подход из статьи [1]. Подход необходимо было адаптировать для монадических парсер-комбинаторов. Кроме того, было решено по возможности сохранить интерфейс парсер-комбинаторов, чтобы использующий их код расширения претерпел минимальные изменения. Далее будут описаны парсер-комбинаторы, реализованные в рамках данной работы. В первом разделе будет описан использованный тип для парсера. Во втором разделе будут описаны основообразующие парсер-комбинаторы бибилотеки Ostap: seq и alt. В третьем разделе будет уделено внимание парсер-комбинатору many, реализация которого также потребовала существенных усилий.

Описание типа

В первую очередь был выведен следующий тип продолжений, показанный в листинге 6 вместе с типом самих парсеров.

Листинг 6: Тип парсера и продолжение

```
1 type ('a, 'b, 'stream) parser  = 'stream -> ('a, 'b, 'stream) k ->
2                                             ('b, 'stream) result
3
4 type ('a, 'b, 'stream) k = 'a -> 'stream -> ('b, 'stream) result
```

Тип парсера, представленный в строке 1, параметризован тремя типами: 'a – это тип монадического значения, которое передается этим парсером продолжению, 'b – это тип синтаксического дерева, которое строит этот парсер и 'stream – это тип входной последовательности, которую принимает этот парсер. Парсер принимает продолжение и строку как аргументы и возвращает результат с типом result.

В строке 3 описан тип продолжения, который параметризован так же, как и тип парсера. Продолжение принимает монадическое значение и остаток строки как аргументы и возвращает результат вычислений, которые представлены в этом продолжении. Такой тип продол-

жения был выбран исходя из реализации парсер-комбинатора seq, описанной в [3], которая заключается в том, что второй аргумент парсер-комбинатора seq передается первому в виде продолжения.

Реализация основных парсер-комбинаторов

Реализация простейших парсеров интереса не представляет, поэтому перейдем сразу к реализации парсер-комбинаторов seq и alt. В листинге 7 представлена реализация seq.

Листинг 7: Монадический парсер-комбинатор seq

```
1 let seq : ('a, 'b, 'stream) parser ->
2           ('a -> ('c, 'b, 'stream) parser) ->
3           ('c, 'b, 'stream) parser =
4   fun x y ->
5       fun s k ->
6           x s (fun a s' -> y a s' k)
```

Аргумент x в строке 2 – это парсер, который запустится первым, а y – это функция, которая примет монадическое значение первого парсера и вернет парсер, который запустится вторым.

Аргументы s и k в строке 3 – это строка и продолжение соответственно, на которых запускается парсер, созданный с помощью seq. Этот парсер работает следующим образом: запускается первый парсер x на строке s и продолжении, описанном в строке 4, которое запустится по завершении работы парсера x. Такое продолжение запускает второй аргумент seq – y, который получает монадическое значение первого парсера и остаток строки s', переданные продолжению парсером x, и продолжение k, которое запустится после окончания работы второго парсера.

Перейдем теперь к реализации парсер-комбинатора alt.

Листинг 8: Парсер-комбинатор alt

```
1 let alt : ('a, 'b, 'stream) parser ->
2           ('a, 'b, 'stream) parser ->
3           ('a, 'b, 'stream) parser =
4   fun x y ->
5     memo (fun s k -> (x s k) <@> (y s k))
```

Этот парсер-комбинатор принимает на вход два парсера x и y, одинакового типа. В 3 строке из них конструируется парсер, который принимает на вход строку s и продолжение k и запускает на них парсеры x и y. После этого результаты суммируются с помощью функции суммирования результатов <@> и сумма возвращается в качестве результата. Этот парсер мемоизируется с помощью функции memo, реализация которой описана в приложении А, и возвращается в качестве результата парсера alt x y.

Парсер-комбинатор many

Опишем реализацию еще одного парсер-комбинатора, который представляет интерес – комбинатора many, работа которого заключается в повторном применении парсера-аргумента.

Листинг 9: Парсер-комбинатор many

```
1  let rec many : ('a, 'b, 's) parser -> ('a list, 'b, 's) parser =
2    fun p ->
3      fun s k ->
4        let result : ('b, 'stream) result ref = ref (k [] s) in
5        let rec loop alist stream =
6          p stream (fun a stream' ->
7                      let alist' = List.rev (a :: (List.rev alist)) in
8                      let curResult = k (alist') stream' in
9                      result := curResult <@> !result;
10                     let _ = loop (alist') stream' in
11                     curResult)
12     in
13     let _ = loop [] s in
14     !result
```

Как и в предыдущих листингах s и k в строке 3 – это аргументы

парсера, сконструированного с помощью парсер-комбинатора many. В строке 4 создается ссылка на результат, с помощью которой будет производиться суммирование результатов каждого кратного применения парсера p для дальшейшей передачи суммы результатов как результата парсера many p. Начальное значение задается как результат нуля последовательных применений парсера p, то есть как незамедлительный вызов продолжения на пустом списке монадических значений, потому что p не вызывался ни разу, и на строке s, потому что мы не продвинулись по ней.

В строке 5 задается итеративный процесс повторных запусков парсера p. Текущая итерация получается информацию о предудущих запусках парсера p посредством двух аргументов: alist – список монадических значений, переданных предыдущими запусками парсера p, и stream – остатком строки после предыдущих запусков p. Единственное действие, которое совершает цикл на каждой итерации – это запуск парсера p на остатке строки stream и на продолжении.

В строке 6 продолжение принимает имеет два аргумента a и stream' – это монадическое значение, переданное парсером p и остаток строки после его работы соответственно. В строке 7 пополняется список монадических значений новым. В строке 8 происходит запуск продолжения k из строки 3 на новом списке и текущем остатке строки. Результат этого запуска суммируется с результатом всех предыдущих кратных запусков. Далее, в строке 9 происходит переход на следующую итерацию с новым остатком строки и обновленным списком монадических значений.

В строке 13 просиходит запуск цикла на начальных данных, коими являются пустой список для списка монадческих значений и вся строка. В строке 14 в качестве результата many p возвращается сумма результатов кратных запусков парсера p, которая строилась в ходе цикла.

Кроме этих трех парсер-комбинаторов были написаны и остальные парсер-комбинаторы, реализованные в библиотеке Ostap, однако их реализация не представляет интереса, и не будет описана в этой работе.

В основном необходимо было переписать реализацию в стиле передачи продолжений.

Глава 3. Интеграция в библиотеку Ostap

Парсер-комбинаторы, описанные в предыдущей главе, имеют тот же интерфейс, что и парсер-комбинаторы из библиотеки Ostap, кроме дополнительного аргумента у парсера – продолжения. Таким образом, получилось заменить только реализацию парсер-комбинаторов, а программный код расширения, который их использует, изменять практически не пришлось.

Однако, для корректной работы библиотеки изменения были необходимы. Они связаны с тем, что для правильной мемоизации необходимо использовать комбинатор неподвижной точки. С его помощью при рекурсивном вызове парсера мемоизирующие таблицы будут одни и те же и не будут создаваться заново при каждом вызове. Поэтому было решено при интерпретации каждой конструкции ostap (...), показанной в листинге 5, генерировать комбинатор неподвижной точки, принимающий на вход n аргументов, где n – количество парсеров, определяемых внутри конструкции, и возвращающий n-местный кортеж с парсерами, избавленными от рекурсивных вызовов. Для примера из листинга 5 будет сгенерирован комбинатор неподвижной точки с 3 аргументами и возвращающий тройку парсеров.

Для того, чтобы при запуске комбинатора неподвижной точки получились парсеры, избавленные от рекурсивных вызовов, в качестве аргументов ему должны быть переданы функции, которые реализованы так же как искомые парсеры, но в которых рекурсивные вызовы заменены на вызов аргументов. В листинге 10 показана такая функция для парсера exp из листинга 5.

Листинг 10: Пример аргумента комбинатора неподвижной точки

```
1 let exp' [primary] =
2   fun exp primary main ->
3       e:exp[primary] -"+" p:primary {`E2 (e, p)}
4     | p:primary {`E1 p}
```

Для наглядности было решено не переписывать тело парсера в чистый OCaml, а оставить как в листинге 5. Такую функцию легко сгене-

рировать зная имена определяемых парсеров и тело самого парсера. В данном примере у функции exp' помимо самого exp добавлены аргументы main и primary, потому что они так же описаны внутри конструкции ostap (...) в листинге 5 и могут быть вызваны в теле парсера exp, что может привести к рекурсивному вызову парсеров main и primary.

Как было сказано выше, для каждой конструкции ostap (...) генерируется свой комбинатор неподвижной точки с необходимым количеством аргументов. Реализация комбинатора была взята из [6] и обобщена на несколько аргументов. В листинге 11 показано, что будет сгенерировано для кода из листинга 5.

Листинг 11: Генерируемый код комбинатора неподвижной точки

```
let (primary, exp, main) =
  let generated_fixpoint f g h =
    let rec p = lazy (f (fun t  -> Lazy.force p t)
                                (fun t  -> Lazy.force q t)
                                (fun t  -> Lazy.force r t))
                and q = lazy (g (fun t  -> Lazy.force p t)
                                (fun t  -> Lazy.force q t)
                                (fun t  -> Lazy.force r t))
                and r = lazy (h (fun t  -> Lazy.force p t)
                                (fun t  -> Lazy.force q t)
                                (fun t  -> Lazy.force r t))
        in (Lazy.force p, Lazy.force q, Lazy.force r)
  in
  let primary' =
    fun exp primary main ->
      {* primary body in OCaml *}
  and exp' primary =
    fun exp primary main ->
      {* exp body in OCaml *}
  and main'  =
    fun exp primary main ->
      {* main body in OCaml *}
  in generated_fixpoint primary' exp' main'
```

На этом примере хорошо видно, что все эти конструкции легко строятся в общем случае, для любого количества определяемых парсеров. Однако такого комбинатора неподвижной точки недостаточно для корректной работы парсера.

Проблема возникает при использовании параметризованных парсеров, например exp из листинга 5. Дело в том, что необходимо убедиться в том, что при передаче парсеру exp одного и того же параметра не будет происходить создания новых мемоизационных таблиц. Для этого было решено производить мемоизацию по параметрам внутри комбинатора неподвижной точки: при генерации каждой из функций p, q и r из листинга 11 дополнительно генерируется код для создания мемозацинной таблицы и при передаче парсеру параметров сначала происходит поиск в таблице. В ячейке таблицы с индексом в виде кортежа параметров записан результат вызова функции f, g или h на этих параметрах, и, если вызов уже происходил, результат берется из таблицы.

С добавлением такой мемоизации получается, что генерируется комбинатор неподвижной точки, который учитывает не только количество определяемых парсеров в рамках данной конструкции ostap (..), но и количество параметров у каждой из них.

Всё, описанное выше, происходит при интерпретации конструкции ostap (..). Также пришлось изменить интерпретацию таких конструкций как CONST в третьей строке листинга 5. Как уже говорилось выше, строки символов в высшем регистре интерпретируются как вызов метода стрима, то есть для правильной работы такого парсера ему необходимо передавать как аргумент стрим, имеющий метод getCONST в случае с парсером CONST. Из-за того, что реализация использует стиль передачи продолжений, для описания типов методов стрима были использованы полиморфные типы. Однако, в таком случае, при вызове метода getCONST у стрима необходимо явно указывать тип и в нем полиморфный тип у соответствующего метода. В листинге 12 показано, что именно генерируется для решения описанной выше проблемы.

Листинг 12: Генерируемый код для конструкции CONST

```
1 (fun
2   (_ostap_stream :
3     <
4       getCONST :
5         'b . (string -> 'self -> ('b,'self) result) -> ('b,'self) result;
6       ..
7     >
8     as 'self) -> _ostap_stream#getCONST)
```

Интерпретация остальных конструкций синтаксиса предметно-ориентирова
языка библиотеки Ostap существенным образом не менялась.

Глава 4. Тестирование и апробация

Тестирование

Цель данного тестирования заключается в том, чтобы убедиться в работоспособности новой версии библиотеки Ostap на примерах как с леворекурсивными правилами вывода, так и без. Также было решено произвести сравнение с предыдущей версией библиотеки, однако при этом допускается ухудшение производительности, потому что в новой версии, в отличие от предыдущей, поддержаны грамматики с левой рекурсией, а так же допущено отступление от стратегии longest match first.

В первую очередь было решено провести тестирование корректности и работоспособности новой версии библиотеки на тестах библиотеки Ostap. Во-первых, было произведено тестирование лексических конструкций таких как IDENT и EOF, описанных ранее в этой работе, и парсер-комбинаторов alt и seq.

Во-вторых, было произведено тестирование конструкций для построения синтаксического дерева и использования переменных для обращения промежуточным результатам. Далее было произведено тестирование парсер-комбинатора many, параметризованных парсеров, а также определения нескольких парсеров в рамках одной конструкции ostap (...).

В-третьих, было произведено тестирование использования программного кода языка Ocaml, модулей и классов при описании парсеров. Затем были протестированы парсеры для контекстно-зависимых языков и использование условий при описании парсеров. После этого было произведено тестирование для парсеров с левой рекурсией и нарушением стратегии longest match first.

Помимо тестирования корректности было решено произвести сравнение новой версии библиотеки Ostap со старой. Для этого были разработаны несколько тестов, представленных в листинге 13 с помощью конструкций DSL библиотеки Ostap.

Листинг 13: Тесты

```
1  ParametrizedList
2  ostap (
3    list[elem] : elem ("," elem)*;
4    id4 : list[IDENT] EOF
5  )
6
7  LeftRecursion
8  ostap (
9    primary : CONST;
10   inner   : expr "-" primary | primary;
11   expr    : inner "+" expr | inner;
12   id7     : expr EOF
13 )
14
15 Expression
16 ostap (
17   expr [nlevels][operator][primary][level]:
18     {nlevels = level} => primary
19   | {nlevels > level} => expr[nlevels][operator][primary][level+1]
20                          ( operator[level] expr[nlevels][operator][primary][
     ↪ level] )?
21
22   primary: IDENT | CONST | "(" intExpr ")" | "-" primary;
23   operator[n]: {n == 0} => ("+" | "-") | {n == 1} => ("*" | "/");
24   intExpr: expr[2][operator][primary][0];
25   id5: intExpr EOF
26 )
```

В строках с 1 по 4 представлен первый тест, который заключается в грамматике списка. Он был выбран для сравнения версий бибилотеки Ostap на грамматике небольшого размера.

В строках с 6 по 11 представлен второй тест, который леворекурсивно описывает арифметические выражения с операциями сложения и вычитания. В связи с тем, что старая версия не работает на этом тесте, он просто показывает поведение новой версии на леворекурсивном примере.

В строках с 13 по 23 представлен третий тест, который описывает арифметические выражения с операциями с разными приоритетами. Этот тест нелеворекурсивен, то есть он опять используется для сравне-

ния версий, но он описывает парсеры с большим количеством параметров, что сильно увеличивает количество генерируемого кода в новой версии библиотеки Ostap.

Время работы тестов и количество использованной памяти было измерено с помощью инструмента corebench [5]. Corebench позволяет вычислять следующие параметры работы тестов: время работы, minor words, major words и promoted words. Для вычисления использованной программой памяти была использована формула (1).

$$memoryused = minorwords + majorwords - promotedwords \quad (1)$$

В таблице 1 представлено усредненное время работы и усредненное количество использованной памяти тестами, представленными в листинге 13, на входных данных различной длины при использовании как предыдущей версии библиотеки Ostap, так и новой. Усреднение происходит для того, чтобы нивелировать влияние других процессов, происходящих на компьютере, на тестирование. Длина входных данных указана в первой графе таблицы через двоеточие.

Исходя из данных в таблице 1 можно сделать вывод, что несмотря на то, что при использовании новой версии библиотеки Ostap генерируется гораздо больше кода, новые парсер-комбинаторы используют гораздо более быстрый алгоритм работы, что и дает гораздо лучшие показатели, чем предыдущая версия. Также в таблице представлены показатели запуска новой версии на леворекурсивном примере LeftResursion и, если сравнить эти данные с временем работы и с количеством занимаемой памяти запусков других тестов, то становится понятно, что примеры с левой рекурсией имеют более чем адекватное время работы.

Апробация

Для апробации было решено использовать новую версию библиотеки Ostap для проекта Reference Compiler. Это позволило использовать леворекурсивные правила, которые до этого приходилось предварительно переводить в нерекурсивную форму. Также получилось пока-

Таблица 1: Результаты тестирования

—	Предыдущая версия		Новая версия	
Тест: длина данных	Время	Память	Время	Память
ParametrizedList:100	2053.33 мкс	400.20 kw	27.11 мкс	4.79 kw
ParametrizedList:200	4085.10 мкс	799.97 kw	38.06 мкс	6.26 kw
ParametrizedList:400	8123.87 мкс	1599.48 kw	60.08 мкс	9.19 kw
ParametrizedList:800	16474.38 мкс	3198.50 kw	120.65 мкс	15.99 kw
Expression:1	134.08 мкс	24.98 kw	40.33 мкс	8.46 kw
Expression:2	259.29 мкс	48.98 kw	43.73 мкс	8.50 kw
Expression:5	648.97 мкс	121.15 kw	51.50 мкс	8.89 kw
Expression:10	1343.56 мкс	242.03 kw	63.54 мкс	9.39 kw
Expression:25	3519.90 мкс	609.18 kw	99.63 мкс	10.60 kw
Expression:50	7137.18 мкс	1236.09 kw	157.82 мкс	12.75 kw
Expression:100	14700.70 мкс	2546.11 kw	277.26 мкс	17.03 kw
Expression:200	29761.28 мкс	5391.20 kw	542.15 мкс	25.53 kw
Expression:400	66589.29 мкс	11981.09 kw	1038.10 мкс	42.62 kw
Expression:800	168943.53 мкс	28761.25 kw	2099.77 мкс	76.62 kw
LeftRecursion:100	—	—	56.78 мкс	6.04 kw
LeftRecursion:200	—	—	92.96 мкс	7.79 kw
LeftRecursion:400	—	—	162.32 мкс	11.26 kw
LeftRecursion:800	—	—	329.46 мкс	18.26 kw

зать, что с новой версией библиотеки Ostap не обязательно следовать стратегии longest match first, что позволяет избежать многих ошибок при разработке парсеров с помощью DLS библиотеки Ostap.

Заключение

В рамках данной работы были выполнены следующие задачи:

- был выполнен обзор следующих реализаций парсер-комбинаторов: алгоритма Warth, алгоритма Frost и алгоритма Meerkat; был выбран подход, разработанный в рамках библиотеки Meerkat;
- разработан алгоритм монадических парсер-комбинаторов, основанный на выбранном подходе;
- произведена интеграция реализованных парсер-комбинаторов в библиотеку Ostap;
- проведена апробация библиотеки и проведено сравнение эффективности с аналогами

Список литературы

[1] A. Izmaylova A. Afroozeh, 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] G. Hutton E. Meijer. Monadic Parser Combinators // technical report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham. — 1996.

[5] Jane Street LLC. Library corebench. — 2013. — URL: `https://ocaml.janestreet.com/ocaml-core/v0.9/doc/core_bench/index.html` (online; accessed: 30.05.2018).

[6] Kiselyov O. Many faces of the fixed-point combinator. — 2013. — URL: `http://okmij.org/ftp/Computation/fixed-point-combinators.html` (online; accessed: 10.05.2018).

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

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

[9] Sloane T. The dsinfo library. — 2016. — URL: `https://bitbucket.org/inkytonik/dsinfo` (online; accessed: 16.04.2018).

[10] Wadler Ph. Monads for functional programming // Proc. Marktoberdorf Summer school on program design calculi. — 1992.

[11] Мартыненко Б. К. Языки и трнасляции. — Издательсто Санкт-Петербургского унивверситета, 2013. — ISBN: 2-288-02870-2.

Приложение А. Алгоритм мемоизации

В листингах 14 и 15 представлена реализация алгоритма мемоизации, описанного в [1], выполненная мной на языке Objective Caml.

Листинг 14: Код алгоритма мемоизации. Часть 1

```
1  let memo : ('a, 'b, 'stream) parser -> ('a, 'b, 'stream) parser =
2    fun f ->
3      let table : ('stream, ('a, 'b, 'stream) parser') Hashtbl.t =
4                                                    Hashtbl.create 16 in
5      fun s k ->
6        try Option.get (Hashtbl.fold (fun s' p' acc ->
7                                        match acc with
8                                        | Some _                        -> acc
9                                        | None when (s # equal s') -> Some p'
10                                       | _                             -> None
11                      ) table None) k
12       with _ ->
13         let r = memoresult @@ (f s) in
14         Hashtbl.add table s r; r k
```

Как было показано в листинге 8 парсер-комбинатор alt использует функцию memo для мемоизации. Рассмотрим теперь в чем заключается ее суть. Эта функция принимает парсер в качестве аргумента и возвращает парсер, но уже мемоизированный, то есть на самом деле это тоже парсер-комбинатор. В теле этой функции в строке 3 листинга 14 создается таблица, которая мемоизирует частично примененные парсеры по стриму, к которому их применили. В строках с 5 по 10 происходит поиск по таблице с целью понять, вызывался ли данный парсер на данном стриме. Если вызывался, то в строке 9 найденному частично примененному парсеру просто передается продолжение. Если нет, то в строке 11 происходит частичное применение парсера к стриму, обработка функцией memoResult, которая возвращается частично примененный парсер с тем же типом. Затем в строке 12 происходит заполнение таблицы и передача продолжения. Таким образом в этой функции обеспечивается, что каждый парсер будет вызван на каждой позиции стрима только один раз, а значит бесконечного цикла не будет. Однако для правильной работы необходимо обеспечить, что для одного и того же парсера

не будут создаваться разные таблицы. Для этого был использован комбинатор неподвижной точки, о чем было рассказано в основных главах этой работы.

Листинг 15: Код алгоритма мемоизации. Часть 2

```
let memoresult : ('a, 'b, 'stream) parser' -> ('a, 'b, 'stream) parser' =
  fun p ->
    let ss : ('stream * 'a) list ref = ref      [] in
    let ks :                K.ks ref = ref K.empty in
    fun k ->
      if K.length !ks = 0
      then (
        ks := K.singleton k;
        p (fun a s ->
             match List.find_all
                     (fun (s', a') -> (a = a') && (s # equal s'))
                     !ss
             with
             | [] -> (
                       ss := (s, a) :: !ss;
                       K.fold (fun k acc -> acc <@> (k a s))
                              !ks
                              emptyResult
                     )
             |  _ -> emptyResult
          ))
      else (ks := K.add k !ks;
            List.fold_left (fun acc x ->
                              match acc, x with
                              | Parsed _,    _           -> acc
                              | Failed _,    Parsed _    -> x
                              | Failed opt1, Failed opt2 -> Failed (cmp opt1 opt2)
                            )
                            emptyResult
                            (List.map (fun (s, a) -> (k a s)) !ss))
```

Парсер на каждом стриме вызывается ровно один раз, но в функции memo из листинга 14 никак не обрабатываются продолжения, которые передаются парсеру и которые могут зациклить его. Для мемоизации продолжений была выделена функция memoResult, описанная в листинге 15 которая получает на вход частично примененный парсер (далее будет называться p) и возвращает тоже частично примененный

парсер (далее будет называться resultP). В строке 4 создается список ks для продолжений, на которых вызывался p, а строке 3 создается список ss для пар из стрима и монадического значения, которые p передавал продолжениям.

При получении продолжения в строке 6 происходит проверка первый ли это вызов resultP. Если вызов происходит впервые, то в список продолжений заносится текущее и происходит вызов p с продолжением, работа которого заключается в следующем: при получении монадического значения a и остатка стрима s происходит поиск в списке ss. Если эта пара была найдена, то есть на них уже происходил вызов, в строке 16 возвращается пустой результат. Если нет, то в строке 14 эта пара заносится в ss и в строке 15 происходит вызов всех продолжений из списка ks и суммирование всех результатов.

Если вызов resultP на продолжении k не является первым, то происходит следующее: в строке 18 продолжение заносится в список и в строке 25 происходит вызов этого продолжения на всех парах из монадичесого значения и остатка стрима, на которых происходил вызов прежних продолжений. В строках с 19 по 24 происходит работа с результатом, которая не вызывает интереса.

Таким образом, функция memoResult обеспечивает, что вызов частично примененного парсера p будет произведен только однажды и все переданные ему продолжения будут вызваны на всех парах из монадического значения и остатка стрима, которые p передавал продолжениям.