

СПИСОК ТЕРМИНОВ

- а) **Формальный язык** (язык) — множество конечных слов (строк, цепочек) над конечным алфавитом.
- б) **Формальная грамматика** (англ. *formal grammar*) — способ описания формального языка, представляющий собой четверку $\Gamma = \langle \Sigma, N, S \in N, P \subset ((\Sigma \cup N)^* N (\Sigma \cup N)^*) \times (\Sigma \cup N)^* \rangle$, где:
 - Σ — алфавит, элементы которого называют **терминалами** (англ. *terminals*).
 - N — множество, элементы которого называют **нетерминалами** (англ. *nonterminals*).
 - S — начальный символ грамматики (англ. *start symbol*).
 - P — набор правил вывода (англ. *production rules* или *productions*) $\alpha \rightarrow \beta$.
- в) **Контекстно-свободная грамматика** (КСГ, англ. *context-free grammar*) — грамматика, у которой в левых частях всех правил стоят только одиночные нетерминалы.
- г) **Контекстно-свободный язык** (КС язык, англ. *context-free language*) — язык, задаваемый контекстно-свободной грамматикой.
- д) **Синтаксический анализ** (разбор, парсинг, англ. *parsing*) — процесс сопоставления линейной последовательности лексем (слов, токенов) формального языка с его формальной грамматикой.
- е) **Синтаксический анализатор** (парсер, англ. *parser*) — программа, производящая синтаксический анализ.
- ж) **Распознаватель** (англ. *recogniser*) — программа, определяющая, принадлежит ли строка формальному языку.
- и) **Семантика языка** — это значение, которое определяется синтаксической структурой распознанной строки.
- к) **Входной поток** (ввод) — линейная последовательность токенов, которая передаётся парсеру для разбора.
- л) **LL(k) грамматика** — грамматика, при разборе которой на основании k токенов входного потока можно однозначно определить правило вывода, которое необходимо применить.

- м) **Поиск с возвратом** (возврат, англ. *backtracking*) — техника, при которой парсер возвращает поток ввода в исходное состояние после попытки разбора каждого правила при разборе нетерминала.
- н) **Longest match first (LMF)** — принцип проектирования парсера, при котором правило, при разборе которого способно поглотиться наибольшее число символов входного потока, должно быть опробовано первым при разборе нетерминала языка.
- п) **Предпросмотр** (англ. *lookahead*) — число символов входного потока, которое парсер учитывает при выборе правила в рамках разбора нетерминала.
- р) **Предметно-ориентированный язык** (англ. *domain-specific language, DSL*) — формальный язык, специализированный для конкретной области применения.
- с) **Продолжение** (англ. *continuation*) — абстрактное представление состояния программы в определённый момент, которое может быть сохранено и использовано для перехода в это состояние; в качестве продолжений можно использовать функции.
- т) **Стиль передачи продолжения** (англ. *continuation passing style, CPS*) — это стиль программирования, в котором функции вместо возвращения значений передают контроль продолжениям, которые определяют, что случится далее.

СОДЕРЖАНИЕ

СПИСОК ТЕРМИНОВ	4
ВВЕДЕНИЕ	8
1. ПОДХОДЫ К СИНТАКСИЧЕСКОМУ АНАЛИЗУ.....	9
1.1. Виды синтаксических анализаторов	9
1.2. Преимущества комбинаторных парсеров	9
1.3. Существующие практические реализации комбинаторных парсеров и их проблемы.....	10
1.4. Левая рекурсия	12
1.5. Типы комбинаторных парсеров.....	13
1.6. Существующие решения	14
1.7. Цель и задачи работы.....	16
Выводы по главе 1	16
2. РАЗБОР В СТИЛЕ ПЕРЕДАЧИ ПРОДОЛЖЕНИЯ	17
2.1. Стил ь передачи продолжения (CPS)	17
2.1.1. Монада Cont.....	18
2.1.2. Монада продолжения полиморфная по результату.....	18
2.1.3. Монада продолжения с состоянием.....	19
2.1.4. Недетерминированная монада продолжения.....	20
2.2. CPS-парсеры	20
2.2.1. Алгоритм разбора с левой рекурсией	21
2.2.2. Алгоритм работы мемоизированных CPS-парсеров.....	23
2.2.3. Реализация алгоритма CPS-парсеров	23
2.2.4. Пример использования CPS-парсера.....	26
2.3. Генерация ключей мемоизации	27
2.3.1. Парсеры первого порядка.....	27
2.3.2. Парсеры высшего порядка.....	28
Выводы по главе 2	30
3. ПРИМЕНЕНИЕ CPS-ПАРСЕРОВ	32
3.1. Разбор арифметических выражений	32
3.2. Разбор контекстно-свободных грамматик.....	35
3.3. Разбор не контекстно-свободных грамматик.....	35
3.4. Сравнение свойств разработанной библиотеки с существующими инструментами	36

ЗАКЛЮЧЕНИЕ	39
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	40

ВВЕДЕНИЕ

Актуальность работы

Различных синтаксических форм представления данных (языков программирования, DSL) становится всё больше, при этом многие такие формы имеют схожие структуры. В связи с этим возникает необходимость писать комбинируемые парсеры, чтобы иметь возможность разбивать их на части и переиспользовать для разбора разных языков. Это позволило бы ускорить время разработки парсеров для схожих формальных языков. Кроме того, такой подход дал бы возможность разбивать крупные парсеры для сложных формальных языков на небольшие части, удобные для долгосрочной поддержки и развития. Наиболее полно задачу комбинации парсеров решают комбинаторные парсеры.

Тем не менее существующие реализации комбинаторных парсеров не поддерживают произвольную композицию и параметризацию парсеров. Это происходит в том числе из-за чувствительности большинства существующих решений к левой рекурсии. В рамках работы решаются проблемы существующих комбинаторных парсеров. В результате работы разработан комбинаторный парсер, нечувствительный к левой рекурсии, который поддерживает произвольную композицию и параметризацию парсеров.

Структура работы

- а) В Главе 1 представлено описание предметной области, обзор существующих решений. В конце главы сформулированы цели и задачи работы.
- б) В Главе 2 представлено описание выбранного подхода, детали реализации решения.
- в) В Главе 3 представлена верификация полученных результатов, даны примеры практического применения разработанного подхода.

ГЛАВА 1. ПОДХОДЫ К СИНТАКСИЧЕСКОМУ АНАЛИЗУ

В этой главе даётся введение в область синтаксического анализа формальных языков, разбираются принятые подходы и приёмы, использующиеся при построении парсеров.

1.1. Виды синтаксических анализаторов

Для выполнения синтаксического анализа используют парсеры. По сути, парсер — это функция из некоторого состояния в список пар из изменённого состояния и результата, то есть функция $S \rightarrow (R \times S)^*$, где S — множество состояний, R — множество результатов. На практике состоянием обычно является поток некоторых символов или токенов, а результатом — дерево разбора. Есть несколько подходов к написанию парсеров. Самый наивный — ручное написание парсера, например, с помощью нисходящего рекурсивного спуска. Такой подход позволяет строить производительные и тонко настраиваемые парсеры, однако, как правило, не подходит для разбора произвольных КСГ, а также значительно увеличивает время и сложность разработки программы синтаксического анализа. Одним из более передовых подходов является использование генераторов парсеров (таких как Bison [2] или ANTLR [1]), которые создают парсер на основе описания грамматики путём использования некоторого предметно-ориентированного языка. Другим передовым подходом является использование комбинаторных парсеров, которые представляют из себя функции, путём композиции которых пользователь неявно задаёт грамматику и семантику языка, анализ которого будет производиться.

1.2. Преимущества комбинаторных парсеров

Традиционно для крупных языков программирования используют генераторы парсеров, так как они за счёт выполнения статического анализа грамматики имеют меньшее время работы. Такое время работы также достигается за счёт того, что генераторы парсеров работают как правило только с $LL(k)$ грамматиками. Следует заметить, что такие грамматики однозначны.

Напротив, комбинаторные парсеры зачастую способны работать с любыми контекстно-свободными грамматиками, однако при этом имеют полиномиальное или даже экспоненциальное относительно длины ввода время работы на некоторых грамматиках и входных данных.

Крайне важной чертой, отличающей комбинаторные парсеры от генераторов парсеров, является интегрированность первых в язык программирования, на котором разрабатывается инструмент, в рамках которого осуществляется синтаксический анализ. Это позволяет получить такие преимущества, как, например, проверка типов на этапе компиляции парсера. Кроме того, такая интегрированность позволяет писать комбинаторные парсеры более абстрактно, отходя от деталей грамматики и семантики конкретного языка программирования. Комбинаторные парсеры можно объединять произвольным образом, а части описания этих парсеров переиспользовать.

1.3. Существующие практические реализации комбинаторных парсеров и их проблемы

Простейшая реализация комбинаторного парсера использует поиск с возвратом, чтобы иметь возможность разбирать произвольные КСГ. Однако использование возврата может приводить к экспоненциальному времени разбора некоторых слов для ряда грамматик. Например, рассмотрим грамматику на рисунке 1.

$$\begin{array}{l} A \rightarrow xAa \\ \quad xAb \\ \quad \in \end{array}$$

Рисунок 1 – Грамматика с экспоненциальной сложностью

На вводах типа "xxxxbbbb" (которые можно записать выражением $x^n b^n$) сложность разбора в соответствии с грамматикой на рисунке 1 растёт экспоненциально с ростом n . Действительно, при раскрытии нетерминала A парсер, использующий поиск с возвратом, сначала пытается применить первое правило, то есть $A \rightarrow xAa$. При этом на вводах типа $x^n b^n$ при раскрытии A всегда необходимо применять второе правило. Получается, что парсер как бы перебирает в лексикографическом порядке все строки вида $x^n(a|b)^n$ для некоторого n , притом строка $x^n b^n$ будет проверена последней, а значит, парсер сделает число шагов равное числу строк вида $x^n(a|b)^n$ минус 1, то есть для заданного n парсер сделает 2^{n-1} шагов.

Во избежание экспоненциального времени работы некоторые практические реализации комбинаторных парсеров могут делать возврат, только если

пользователь явно укажет, что он необходим при разборе правила некоторого нетерминала (так делает Megaparsec [11] с помощью комбинатора `try`). Это позволяет достигнуть линейного времени работы, однако использование этого подхода приводит к тому, что парсеры приходится писать с учётом LMF, что не позволяет разбирать неоднозначные грамматики, а также приводит к невозможности описать любую КСГ. Это плохо само по себе, однако ещё хуже то, что следствием этого является невозможность комбинировать такие парсеры, ведь при произвольном объединении грамматик путём композиции функций-парсеров LMF может перестать соблюдаться. Например, при объединении грамматик, состоящих из нетерминалов A и B с рисунка 2, путём задания парсера для нетерминала C строку "somebody" нельзя будет полностью распознать, потому что её префикс "some" будет поглощён правилом $A \rightarrow some$, хотя грамматика из нетерминала B способна поглотить весь поток ввода.

$$\begin{array}{lcl}
 A & \rightarrow & \text{anybody} \\
 & & \text{some} \\
 B & \rightarrow & \text{somebody} \\
 & & \text{any} \\
 C & \rightarrow & A \\
 & & B
 \end{array}$$

Рисунок 2 – Неправильное объединение парсеров

Другим подходом, позволяющим комбинаторным парсерам с возвратом работать за время меньшее, чем экспоненциальное, является использование мемоизации. Причина экспоненциального времени работы парсеров с возвратом (как в том числе можно видеть из примера с рисунка 1) заключается в том, что одна и та же часть ввода разбирается парсером несколько раз. Такого поведения можно избежать путём запоминания промежуточных результатов разбора парсера. При использовании комбинаторного парсера любой нетерминал грамматики является функцией, которая принимает поток ввода и возвращает список пар из изменённого состояния и результата. Например, парсер для нетерминала $A \rightarrow aA|\epsilon$ на строке "aaa" вернёт [("aaa", "") ("aa", "a"), ("a", "aa"), ("", "aaa")]. Для мемоизации парсера A необходимо завести таблицу, в которую при первом вызо-

ве парсера A на "aaa" добавится запись о том, что для ввода "aaa" парсер должен вернуть [("aaa", "") ("aa", "a"), ("a", "aa"), ("", "aaa")]. При последующих вызовах парсера A на "aaa" будет возвращаться мемоизированный результат. При условии возможности находить хеш частей входного потока и сравнивать их за константное время эта операция выполняется за константное время. Следует заметить, что подход с мемоизацией обладает теми же свойствами, что и хорошо известный алгоритм Эрли [4, 13] за исключением того факта, что мемоизированные парсеры не могут быть записаны леворекурсивно. Как следствие, мемоизированные парсеры с возвратом способны разбирать любую КСГ за полиномиальное время.

Тем не менее большинство мемоизированных комбинаторных парсеров с возвратом ограничены в том, как их можно комбинировать между собой. Особенно это касается комбинаторных парсеров высшего порядка, которые на основе одних парсеров создают другие. Одной из проблем является невозможность большинства комбинаторных парсеров завершаться при разборе леворекурсивных грамматик.

1.4. Левая рекурсия

Левая рекурсия возникает, когда для разбора нетерминала грамматики необходимо разобрать этот же нетерминал в той же позиции.

$$E \rightarrow E + E$$

t

Рисунок 3 – Леворекурсивная грамматика

Грамматику и семантику некоторых языков гораздо удобнее описывать с использованием левой рекурсии. Кроме того, левая рекурсия может возникнуть при использовании комбинаторных парсеров высшего порядка. Рассмотрим пример из листинга 1.

Листинг 1 – Возникновение левой рекурсии

```
seq (x, y) = x ";" y
stmt = seq (stmt, stmt) | VAR ":"=" EXPR
```

Сам по себе парсер `seq` нелеворекурсивен, однако при вызове его с `stmt` в качестве первого аргумента возникает левая рекурсия. Этот нарочито про-

стой пример показывает, как левая рекурсия может неожиданным образом возникнуть при использовании парсеров высшего порядка.

Таким образом, одно из главных преимуществ комбинаторных парсеров, заключающееся в их композиционности, остаётся раскрытым не до конца в случае, когда леворекурсивные грамматики не могут быть разобраны парсером.

1.5. Типы комбинаторных парсеров

Для написания комбинаторных парсеров обычно используют одну из двух абстракций: **монаду** (англ. *monad*) или **аппликатив** (англ. *applicative*). Парсеры, которые используют эти абстракции называют соответственно **монадическими** и **аппликативными**. Можно заметить, что тип любого парсера (то есть $S \rightarrow (S \times R)^*$) принадлежит классу монад [8]. Монадические парсеры являются подмножеством аппликативных, то есть с помощью монадического парсера всегда можно закодировать аппликативный. Монадические парсеры обладают большей выразительной мощностью, чем аппликативные, что позволяет с их помощью описывать не только контекстно-свободные грамматики. Однако эта мощь приводит к тому, что структуру монадических парсеров невозможно анализировать статически, что нельзя сказать об аппликативных парсерах. В листинге 4 приведён пример монадического парсера, который парсит слова из не контекстно-свободного языка, задаваемого множеством слов $\{a^n b^n c^n \mid n \in \mathbb{N}\}$. Заметим, что структура парсера `anbnсn` формируется во время работы программы, а не на этапе компиляции, и зависит от числа символов 'a', которое присутствует в начале разбираемой строки.

Листинг 2 – Класс монад в Haskell

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
```

Листинг 3 – Класс аппликативов в Haskell

```
class Applicative f where
  (<$>) :: (a -> b) -> f a -> f b
  pure  :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

Листинг 4 – Монадический парсер для не КС языка

```
anbn cn :: Parser String String
anbn cn = do
  a <- some (char 'a')
  b <- replicate (length a) (char 'b')
  c <- replicate (length a) (char 'c')
  return (a ++ b ++ c)
```

1.6. Существующие решения

На текущий момент есть несколько алгоритмов синтаксического анализа, позволяющих разбирать любые леворекурсивные контекстно-свободные грамматики с использованием комбинаторных парсеров. Рассмотрим их подробнее.

Использование длины остатка ввода [6]

Данный подход использует длину оставшегося ввода и специальный контекст для подсчёта числа леворекурсивных вызовов, которое было сделано в некоторой ветви разбора. Если число леворекурсивных вызовов превышает оставшуюся длину ввода, ветвь разбора завершается с пустым результатом. Для обеспечения полиномиального времени работы используется мемоизация. Главным недостатком данного подхода является асимптотика $O(n^4)$ для леворекурсивных грамматик. Для всех остальных грамматик асимптотика — $O(n^3)$. Кроме того для работы данного подхода необходимо знать длину ввода.

Cancellation-разбор [12]

Данный подход использует «cancellation set», который хранит посещённые в текущей ветке разбора нетерминалы. С помощью «cancellation set» можно отслеживать леворекурсивные вызовы нетерминалов и успешно их проводить. Недостатком разбора является экспоненциальное время работы, а также необходимость изменения грамматики, путём добавления специальных нетерминалов, которые ответственны за работу с «cancellation set».

Леворекурсивный разбор PEG [15]

PEG — это особый вид грамматик, которые могут быть разобраны за линейное время с использованием алгоритма Packrat [5]. Оригинальный алгоритм не способен работать с левой рекурсией, тем не менее в него можно добавить поддержку леворекурсивных парсеров с помощью процесса под названием «grow the seed» [15], в рамках которого леворекурсивная цепочка правил применяются к потоку ввода много раз до тех пор, пока это применение

закончивается успешно. Данная модификация сохраняет линейное время работы алгоритма.

Такое время работы достигается в алгоритме Packrat с помощью использования мемоизации, а также за счёт особенностей PEG, в частности того факта, что PEG локально и глобально однозначны в отличие от КСГ, которые могут быть локально неоднозначными даже при условии глобальной однозначности.

Недостаток этого подхода заключается в том, что в PEG привычная операция альтерации между правилами нетерминалов не является симметричной. Например, грамматика $A \leftarrow aa/a$ не эквивалентна $A \leftarrow a/aa$ (в PEG вместо «|» принято писать «/», а вместо «→» писать «←»): на вводе "ab" парсер для $A \leftarrow ab/b$ должен вернуть "ab", а парсер для $A \leftarrow b/ab$ вернуть "a", потому что правило $A \leftarrow aa$ не будет применено к вводу, так как первое завершилось успешно. Говоря иначе, парсеры для PEG не делают возврат, потому что само описание PEG делает это невозможным. В связи с этим парсеры для PEG должны быть написаны с учётом LMF, что не позволяет их комбинировать.

Разбор в стиле передачи продолжений или CPS [10]

Джонсон предложил подход к написанию распознавателей с использованием мемоизированных продолжений. Оказывается, что распознаватели, полученные путём применения этого подхода нечувствительны к левой рекурсии, а также имеют время работы $O(n^3)$. К сожалению, в оригинальной работе приводится способ строить только распознаватели, но не парсеры. Следствием этого является использование множеств для хранения результатов разбора в оригинальной работе, что приводит к необходимости уметь сравнивать такие результаты, а также искать их хеш за константу. Это не проблема для распознавателей, где результатом является длина разобранного префикса ввода, однако для парсеров такое требование неприемлемо, ведь при работе с ними пользователь сам определяет тип результата, для которого не всегда возможно написать константную по времени реализацию сравнения и поиска хеша. Кроме того, при наивном расширении алгоритма на парсеры время его работы становится неограниченно полиномиальным, где степень полинома зависит от размера грамматики. Одно из решений этих двух проблем предложено в библиотеке комбинаторных парсеров Meerkat [9]. Предложенное решение опирается на построение SPPF в рамках разбора. Такой подход не позволя-

ет задавать пользователю произвольную семантику для разбора. Тем не менее для однозначных грамматик существует способ расширить подход, использованный в оригинальной работе Джонсона, на парсеры, в том числе на монадические комбинаторные парсеры, сохранив возможность задания произвольной семантики разбора.

1.7. Цель и задачи работы

После введения в предметную область можно поставить цель и определить задачи ВКР.

Цель работы

Добиться возможности произвольной композиции и параметризации парсеров.

Задачи работы

- а) Изучение литературы по теме.
- б) Создание библиотеки комбинаторных парсеров, обладающих следующими свойствами:
 - 1) Распознавание любой однозначной контекстно-свободной грамматики.
 - 2) Нечувствительность к LMF.
 - 3) Нечувствительность к левой рекурсии.
 - 4) Полиномиальное время работы.
 - 5) Возможность задания семантики разбора.
- в) Оценка работы написанной библиотеки.

Выводы по главе 1

В этой главе были описаны основные подходы к созданию инструментов синтаксического анализа. Были разобраны тонкости реализации комбинаторных парсеров, рассмотрены существующие решения. Кроме того, были поставлены цель и задачи ВКР.

ГЛАВА 2. РАЗБОР В СТИЛЕ ПЕРЕДАЧИ ПРОДОЛЖЕНИЯ

Для достижения цели, поставленной в работе, было принято решение взять за основу CPS распознаватели и усовершенствовать их. В этой главе сначала будет рассмотрено, что такое стиль передачи продолжения в целом, далее будет описана монада продолжения, после этого её мемоизированная версия. Наконец, будет показано, как создать парсер с помощью мемоизированной монады продолжения.

2.1. Стиль передачи продолжения (CPS)

Для написания реализованной в рамках ВКР библиотеки был выбран язык программирования Haskell. Haskell является чистым нестрогим функциональным языком. Разберёмся, как устроен стиль передачи продолжения в подобном языке. Рассмотрим пример кода из листинга 5 для расчёта квадрата гипотенузы треугольника [7] по теореме Пифагора.

Листинг 5 – Теорема Пифагора

```
add :: Int -> Int -> Int
add x y = x + y

square :: Int -> Int
square x = x * x

pythagoras :: Int -> Int -> Int
pythagoras x y = add (square x) (square y)
```

Чтобы записать тот же код в CPS необходимо, чтобы функции `add`, `square` и `pythagoras` вместо `Int` возвращали функцию, которая принимает продолжение и возвращает результат, эту функцию будем называть *отложенным вычислением*. Тип такой функции $(Int \rightarrow r) \rightarrow r$, то есть само продолжение имеет тип $Int \rightarrow r$. Код, переписанный в стиле передачи продолжения, представлен на листинге 6. Можно сказать, что теперь все функции принимают дополнительный аргумент с типом $Int \rightarrow r$, который является продолжением.

Теперь, чтобы получить результат вычислений, например, для треугольника с катетами 3 и 4, можно вызвать `pythagoras_cps 3 4 id`.

Листинг 6 – Теорема Пифагора в CPS

```

add_cps :: Int -> Int -> ((Int -> r) -> r)
add_cps x y = \k -> k (add x y)

square_cps :: Int -> ((Int -> r) -> r)
square_cps x = \k -> k (square x)

pythagoras_cps :: Int -> Int -> ((Int -> r) -> r)
pythagoras_cps x y = \k ->
  square_cps x $ \x_squared ->
  square_cps y $ \y_squared ->
  add_cps x_squared y_squared $ k

```

2.1.1. Монада Cont

Пусть необходимо последовательно соединить два отложенных вычисления $(a \rightarrow r) \rightarrow r$ и $(b \rightarrow r) \rightarrow r$, притом второе вычисление хочется составлять на основе первого. Функция для этого представлена на листинге 7.

Листинг 7 – Соединение отложенных вычислений

```

chainCPS :: ((a -> r) -> r) ->
  (a -> ((b -> r) -> r)) ->
  ((b -> r) -> r)
chainCPS s f = \k -> s $ \x -> f x $ k

```

Заметим, что если заменить $(a \rightarrow r) \rightarrow r$ на $m\ a$, $a\ (b \rightarrow r) \rightarrow r$ на $m\ b$, то функция `chainCPS` по сути будет являться монадическим `bind`.

Разовьём эту идею, добавив новый тип с листинга 8.

Листинг 8 – Тип Cont

```

newtype Cont r a = Cont {
  runCont :: (a -> r) -> r
}

```

По сути тип `Cont` — это обёртка над функцией $(a \rightarrow r) \rightarrow r$. Этот тип является монадой, что показано на листинге 9.

2.1.2. Монада продолжения полиморфная по результату

Недостаток монады `Cont` с листинга 8 заключается в том, что тип результата (то есть `r`) должен оставаться неизменным внутри монады. Это означает, что при использовании функции `bind` нельзя изменить тип результата монады,

Листинг 9 – Включение типа `Cont` в класс `Monad`

```
instance Monad (Cont r) where
  return x = Cont ($ x)
  s >>= f = Cont $ \c -> runCont s $
    \x -> runCont (f x) c
```

что не позволяет, например, при описании парсеров соединять через `bind` парсеры, возвращающие разные по типу результаты. Чтобы это исправить нужно, чтобы тип результата определялся не типом `Cont`, а типом продолжения `a -> r`. Для этого воспользуемся Haskell. Обновленная версия типа `Cont`, которая полиморфна по результату, а также включение `Cont` в класс монад для неё представлен на листинге 10. Можно заметить, что определение включения `Cont` в класс монад не изменилось.

Листинг 10 – Монада продолжения полиморфная по результату

```
newtype Cont a = Cont
  { runCont :: forall r. (a -> r) -> r
  }

instance Monad (Cont r) where
  return x = Cont ($ x)
  s >>= f = Cont $ \c -> runCont s $
    \x -> runCont (f x) c
```

2.1.3. Монада продолжения с состоянием

Для написания мемоизированного парсера в CPS необходимо, чтобы отложенные вычисления использовали таблицу мемоизации. Такая таблица мемоизации является по сути изменяемым состоянием. Для его реализации можно воспользоваться монадой состояния. Тогда тип `Cont` необходимо изменить. Изменённый тип представлен на листинге 11.

Листинг 11 – Тип `Cont` с состоянием

```
newtype Cont a = Cont {
  runCont :: forall r. (a -> State MemoTable r)
    -> State MemoTable r
}
```


Теперь продолжения имеют тип `a -> State MemoTable r` и могут использовать таблицу мемоизации для вычисления результата. Класс `Monad` для нового типа продолжений определяется так же, как на листинге 10.

2.1.4. Недетерминированная монада продолжения

В результате работы парсера может получиться несколько результатов или вовсе ни одного, что свидетельствует о том, что разбираемая строка не принадлежит языку. Поэтому отложенные вычисления в монаде `Cont` должны возвращать не единственный результат (как это следовало из типа `Cont` до сих пор), а коллекцию результатов. В качестве коллекции используем обычный список. Снова изменим тип `Cont`, заменив тип результата `r` на `[r]`. Изменённый тип представлен на листинге 12. Важно заметить, что изменённый тип принадлежит классу `Alternative`.

Листинг 12 – Недетерминированная монада продолжения с состоянием

```
newtype Cont a = Cont {
  runCont :: forall r. (a -> State MemoTable [r])
    -> State MemoTable [r]
}

instance Alternative Cont where
  empty :: Cont a
  empty = Cont (\_ -> return empty)
  (<|>) :: Cont a -> Cont a -> Cont a
  (<|>) l r =
    Cont
      ( \k -> do
          r1 <- runCont l k
          r2 <- runCont r k
          return $ r1 <|> r2
      )
```

2.2. CPS-парсеры

Теперь с использованием разработанной в разделе 2.1.4 `Cont` монады несложно написать парсер в стиле передачи продолжения. Как уже было замечено ранее парсер — это функция типа `s -> [(r, s)]`, где `s` — это тип потока ввода, а `r` — тип результата разбора. Если возвращаемый список пустой, то разбираемая строка не принадлежит языку, если возвращаемый список содержит более одного элемента, то строка может быть разобрана несколькими способами. На самом деле парсер не обязан возвращать список. Например, вместо

списка парсер может возвращать `Maybe` (листинг 13). В этом случае его тип — `s -> Maybe (r, s)`. Парсер с таким типом может возвращать `Nothing`, если разбор закончился неудачно, или `Just (r, s)`, если разбор завершился успехом. Заметим, что такой парсер не может вернуть несколько результатов разбора. Подробнее с этим можно ознакомиться в рамках работы [8].

Листинг 13 – Тип `Maybe` в Haskell

```
data Maybe a = Just a | Nothing
```

Такая независимость парсера от контейнера для результата наводит на мысль написать обобщённый тип парсера, параметризованный относительно типа такого контейнера: `type ParserT m s r = s -> m (r, s)`. Полученный тип в точности соответствует монадическому трансформеру `StateT` из Haskell (листинг 14). При условии, что тип `m` принадлежит классу монад, тип `StateT` также принадлежит к классу монад.

Листинг 14 – `StateT` трансформер монад

```
newtype StateT s m a = StateT (s -> m (a, s))
```

Получается, что любой монадический парсер с типом ввода `s` и типом результата `r` можно записать, как `StateT s m r`, где `m` — это некоторый тип, принадлежащий классу монад. Как можно было заметить на примере со списком и `Maybe`, тип `m` определяет свойства парсера. Тип `Cont` из раздела 2.1.4 является монадой, а значит, значения с типом `StateT s Cont r` являются монадическими парсерами. Такие монадические парсеры будем называть парсеры в CPS. Введём для них тип с именем `Parser`, как на листинге 15. Тип результата `r` опущен, чтобы кайнд полученного типа был `* -> *` и `Parser` принадлежал к классу монад (в Haskell частично параметризованные типы не поддерживаются).

Листинг 15 – Парсер в CPS

```
type Parser s = StateT s Cont
```

2.2.1. Алгоритм разбора с левой рекурсией

Проблема с комбинаторными парсерами, написанными леворекурсивно, заключается в том, что такие парсеры вызывают сами себя, при этом не меняя

состояние разбора. В связи с этим код производит леворекурсивные вызовы, пока не переполнится стек вызовов и программа не завершится с ошибкой.

Чтобы избежать подобного исхода, можно сделать следующее в случае, когда разбор нетерминала приводит к леворекурсивному вызову:

- а) Не производить леворекурсивный вызов, при этом сохранив цепочку парсеров, которая должна была быть вызвана после него.
- б) Когда у нетерминала, разбор которого привёл к леворекурсивному вызову, появляется новый результат, вызвать сохранённые цепочки парсеров на этом результате.

Заметим, что если при вызове сохранённой цепочки парсеров у нетерминала, разбор которого привёл к леворекурсивному вызову, появляется новый результат, то надо снова вызвать сохранённые цепочки парсеров на этом результате.

$$A \rightarrow \begin{matrix} Aa \\ b \end{matrix}$$

Рисунок 4 – Леворекурсивная грамматика

Чтобы лучше понять описанный алгоритм, рассмотрим пример. Взглянем на грамматику на рисунке 4. Парсер для этой грамматики принимает строки вида ba^* . Если придерживаться описанного алгоритма, то парсер для нетерминала A на строке "baa" сначала попытается вызвать самого себя. При этом алгоритм должен определить, что произошёл леворекурсивный вызов и не вызывать парсер, а вместо этого сохранить то, что должно распознаться после вызова A , то есть парсер для терминала a . Потом парсер для нетерминала A попытается вызвать парсер для терминала b , этот парсер вернёт результат ("b", "aa"). Получается, что у нетерминала A появился новый результат, значит, на этом результате нужно согласно алгоритму вызвать сохранённый парсер, то есть парсер для терминала a , который вернёт результат ("ba", "a"). У нетерминала A снова появился новый результат, значит, на нём снова нужно вызвать парсер для a , который вернёт результат ("baa", ""). Таким образом парсер, следуя описанному ранее алгоритму, смог разобрать всю входную строку, несмотря на левую рекурсию.

2.2.2. Алгоритм работы мемоизированных CPS-парсеров

Работа, проделанная в предыдущих разделах, была необходима для того, чтобы мемоизировать парсеры так, чтобы они поддерживали левую рекурсию.

По сути алгоритм, который был описан в разделе 2.2.1 и есть алгоритм работы мемоизированных CPS-парсеров. При этом цепочка парсеров, которая должна быть вызвана после леворекурсивного вызова, есть продолжение парсера, который к этому вызову привёл. В связи с этим алгоритм можно описать следующим образом, теперь уже без упора на левую рекурсию.

- а) Для каждого парсера нетерминала храним список его результатов и список его продолжений.
- б) При появлении у парсера нового продолжения вызываем его на всех результатах.
- в) При появлении у парсера нового результата вызываем на нём все продолжения.

Важно заметить, что подобный алгоритм не только позволяет записывать парсеры леворекурсивно, но и гарантирует полиномиальное время работы парсера. Чтобы убедиться в этом, обратимся к источникам. В статье, где впервые был предложен подобный алгоритм для CPS распознавателей [10], утверждается, что он работает за полиномиальное время. Алгоритм для парсеров производит те же самые шаги, что и алгоритм для распознавателей, следовательно, должен также работать за полиномиальное время. Кроме того, в статье, где была разработана другая библиотека CPS-парсеров под названием Meerkat [9], доказывалось, что CPS-парсеры работают за полином, притом для не сильно неоднозначных грамматик это время ограничено $O(n^3)$. Наконец, нужно сказать, что представленный алгоритм является частным случаем мемоизированных парсеров. В статье [13] доказывалось, что свойства мемоизированных парсеров аналогичны свойствам парсеров Эрли.

2.2.3. Реализация алгоритма CPS-парсеров

На листингах 16 и 17 представлен основной код, необходимый для работы мемоизированных CPS-парсеров. Напомним, что тип `Cont` принадлежит классу монад, как это было показано ранее.

Два типа `MemoEntry` и `MemoTable` с листинга 16 используются для мемоизации. Тип `MemoTable` — это тип таблицы мемоизации, из этой таблицы по ключу, который уникально идентифицирует парсер, а также вводу можно

получить значение типа `MemoEntry`. `MemoEntry` хранит результаты применения парсера с ключом к некоторому вводу, а также продолжения этого парсера на этом вводе, то есть по сути парсеры, которые могут быть вызваны после парсера, `MemoEntry` которого рассматривается. `MemoTable` используется в качестве состояния для `Cont`. Для удобства также введён тип `ContState`, который является типом результата мемоизированного продолжения. Введённый на листинге 15 тип `Parser` переименован в `BaseParser`, далее станет понятно для чего.

Можно заметить, что в коде используется тип `Dynamic` (на листинге 16), а также функции `fromDynUnsafe`, `toDyn` и `toDynContinuation` (на листинге 17). Они необходимы, потому что таблица мемоизации должна хранить значения разных типов, и используются для приведения типов, что на Haskell можно сделать с использованием `Dynamic` [3].

Листинг 16 – Типы для CPS-парсера

```
data MemoEntry k s a r = MemoEntry
{ results :: [(a, s)],
  continuations :: [(a, s) -> ContState k s [r]]
}

type MemoTable k s =
  Map.HashMap
    k
    (Map.HashMap s (MemoEntry k s Dynamic Dynamic))

type ContState k s = State (MemoTable k s)

newtype Cont k s a = Cont
{ runCont ::
  forall r.
    (Typeable r) =>
    (a -> ContState k s [r]) ->
    ContState k s [r]
}

type BaseParser k s = StateT s (Cont k s)
```

На листинге 17 определена функция `baseMemo`, которая делает из обычного парсера мемоизированный. Она принимает уникальный ключ и парсер для мемоизации. Когда мемоизированный парсер вызывается впервые, в таблице мемоизации по ключу для этого парсера создаётся пустой словарь, это происходит в строке

Листинг 17 – Мемоизация для CPS-парсера

```

baseMemo ::
  (Typeable a, Hashable k, Hashable s, Eq k, Eq s) =>
  k -> BaseParser k s a -> BaseParser k s a
baseMemo key parser = StateT $ \state ->
  Cont $ \continuation ->
  do
    modify (Map.insertWith (\_ old -> old) key Map.empty)
    entry <- gets $
      \table -> Map.lookup state $ table Map.! key
    case entry of
      Nothing -> do
        modify $
          addNewEntry state $
            MemoEntry [] [toDynContinuation continuation]
        runCont
          (runStateT parser state)
          ( \result -> do
            modify (addResult state result)
            conts <- gets $ \table ->
              continuations $ (table Map.! key) Map.! state
            join <$> mapM
              ( \cont ->
                fmap fromDynUnsafe <$> cont (first toDyn result)
              )
            conts
          )
      Just foundEntry -> do
        modify (addContinuation state continuation)
        join <$> mapM
          (continuation . first fromDynUnsafe)
          (results foundEntry)

```

`modify (Map.insertWith (_ old -> old) key Map.empty).` Далее проверяется, есть ли в этом словаре запись для входа, на котором был вызван парсер.

Если такой записи нет, то она создаётся путём вызова функции `addNewEntry`, в запись сразу же помещается продолжение, с которым был вызван мемоизированный парсер. Далее вызывается сам парсер с особым продолжением. Это продолжение по завершении парсера с некоторым результатом сохраняет этот результат в `MemoEntry`, а также вызывает все сохранённые продолжения парсера на полученном результате и возвращает их результаты, соединённые в единый список.

Если запись в таблице мемоизации по ключу парсера и вводу есть, то в неё добавляется переданное в парсер продолжение, это происходит путём вызова функции `addContinuation`. Далее это продолжение вызывается на всех сохранённых результатах парсера.

Можно заметить, что сам мемоизированный парсер для заданного ввода вызывается единожды, далее используются его сохранённые результаты, этим и объясняется полиномиальное время работы алгоритма.

2.2.4. Пример использования CPS-парсера

Покажем, как использовать написанный парсер. Сделаем это на примере грамматики с рисунка 4. Для этого сначала введём примитивный парсер, который разбирает переданную в него строку. Его можно найти на листинге 18. Функция `stripPrefix` импортирована из модуля `Data.List` стандартной библиотеки Haskell. Для наглядности в качестве потока ввода в парсере используется `String`, это далеко не самое удачное решение с точки зрения производительности, библиотека позволяет использовать и другие, более эффективные потоки ввода.

Листинг 18 – Парсер для строки

```
string :: String -> BaseParser k String String
string s =
  StateT
    ( \state ->
      case stripPrefix s state of
        Just s' -> return (s, s')
        Nothing -> empty
    )
```

Теперь можно написать парсер для грамматики с рисунка 4. Он представлен на листинге 19. В качестве ключа для парсера используется целое число 1. Ввиду того, что парсер единственный, ключ может быть любым. Главное, чтобы тип ключа поддерживал проверку на равенство, а также поиска хеша за $O(1)$. Для написания парсера `baa` использован тот факт, что любая монада является также функтором и аппликативом. Это значит, что для парсеров определены операторы `<$>` и `<*>`.

Листинг 19 – Парсер для строк *ba**

```
baa :: BaseParser Int String String
baa =
  baseMemo 1 $
    (++) <$> baa <*> chunk "a"
    <|> chunk "b"
```

2.3. Генерация ключей мемоизации

2.3.1. Парсеры первого порядка

Расставлять ключи мемоизации вручную неудобно. Более того, при таком подходе сложно обеспечить уникальность ключей. Поэтому необходимо предоставить средства для автоматической генерации ключей мемоизации. Для этого добавим новый тип `Key` с листинга 20. `Key` позволяет сравнивать значения разных типов.

Листинг 20 – Тип ключей мемоизации

```
data Key = forall a. (Typeable a, Hashable a, Eq a) => Key a

instance Eq Key where
  (==) :: Key -> Key -> Bool
  (==) (Key a) (Key b) =
    case cast b of
      Just x -> a == x
      Nothing -> False

instance Hashable Key where
  hashWithSalt :: Int -> Key -> Int
  hashWithSalt s (Key a) = hashWithSalt s a

makeStableKey :: (Typeable a) => a -> Key
makeStableKey a = Key (unsafePerformIO $ makeStableName a)
```

На том же листинге 20 введена функция `makeStableKey` `a`, которая генерирует ключ на основе `stableName` [14] своего аргумента. Для аргументов с одинаковыми адресами в памяти функция `makeStableKey` возвращает равные ключи, при этом функция также может вернуть равные ключи для равных аргументов, даже если у этих аргументов разные адреса в памяти. Так происходит из-за оптимизаций компилятора, это не мешает использованию `makeStableKey` по назначению. Предполагается, что функция `makeStableKey` будет вызываться с функциями-парсерами в качестве аргумента, чтобы сгенерировать ключ для этого парсера, как это сделано на листинге 21.

Листинг 21 – Пример использования makeStableKey

```

baa :: BaseParser Key String String
baa =
  baseMemo (makeStableKey baa) $
    (++) <$> baa <*> chunk "a"
    <|> chunk "b"

```

Теперь можно ввести функцию мемо, которая будет автоматически генерировать ключи. Закодируем парсер baa с её помощью, как на листинге 22.

Листинг 22 – Мемо с автоматическим ключом

```

мемо ::
  (Typeable s, Typeable a, Hashable s, Eq s) =>
  Parser s a -> Parser s a
memo p = baseMemo (makeStableKey p) (parser p)

baa :: BaseParser Key String String
baa =
  мемо $
    (++) <$> baa <*> chunk "a"
    <|> chunk "b"

```

2.3.2. Парсеры высшего порядка

Функция мемо отлично работает для парсеров первого порядка, однако её невозможно использовать с парсерами высшего порядка. Для примера представим, что необходимо мемоизировать парсер count p n с листинга 23, который принимает два аргумента: парсер и число раз, которое этот парсер необходимо последовательно применить к вводу. Чтобы мемоизировать этот парсер, в качестве ключа можно было бы использовать что-то похожее на Key (makeStableKey count, makeStableKey p, n), то есть тройку из адреса самого парсера count, адреса его аргумента p и числа n. Так как все три элемента тройки поддерживают операцию равенства и хеширования, то и тройка тоже её поддерживает, притом за $O(1)$. Мемоизированная версия count называется mCount на листинге 23.

Тем не менее в таком подходе есть проблема. Предположим, что написан парсер joinedCount n = mCount (mCount (string "a") 3) n. Такой пример кажется несколько надуманным, однако суть его в том, что в mCount подставляется парсер высшего порядка. Теперь для

Листинг 23 – Парсер высшего порядка count

```

count ::
  BaseParser Key String String ->
  Int ->
  BaseParser Key String String
count p n = join <$> replicateM n p

mCount ::
  BaseParser Key String String ->
  Int ->
  BaseParser Key String String
mCount p n = baseMemo
  (Key (makeStableKey mCount, makeStableKey p, n)) $
  join <$> replicateM n p

```

первого аргумента внешнего mCount неправильным будет вызывать makeStableKey (mCount (string "a") 3), потому что этот вызов каждый раз будет возвращать новый ключ, для одинаковых парсеров, ведь mCount (string "a") 3 создаётся заново при каждом вызове (если не вмешаются оптимизации компилятора, на которые в общем случае рассчитывать нельзя).

Чтобы избежать такой ситуации необходимо в парсере mCount для мемоизации вместо makeStableKey p использовать ключ парсера p. Для этого в каждом парсере необходимо хранить его ключ. Введём новый тип парсера на листинге 24.

Листинг 24 – Тип парсера с ключом

```

data Parser s a = Parser
  { key :: Maybe Key,
    parser :: BaseParser Key s a
  }

```

Очевидно, что тип Parser принадлежит классу монад. При операциях fmap, pure, <*>, >>=, <|> для созданного парсера ключ равен Nothing. Так сделано по двум причинам. Первая причина — невозможность сравнивать функции по значению, это приводит к невозможности сгенерировать ключи мемоизации для комбинаторов, которые принимают произвольные функции, таких как fmap, <*> и >>=. Примерно по той же причине ключ не генерируется для комбинатора pure, в pure передаётся некоторый результат парсера, на этот результат не хочется налагать ограничения на возможность проверки на

равенство и поиска хеша. Вторая причина, по которой некоторые ключи равны `Nothing` — производительность. Например, для комбинатора `<|>` ключ нужно генерировать на основе двух его аргументов-парсеров, таким образом, ключ получается вложенным, время сравнения таких ключей потенциально может зависеть от размера грамматики.

Для других парсеров в том числе `empty` и некоторых примитивных парсеров, таких как `string`, ключ генерируется на основе аргументов. То есть, например, все парсеры `string` для одинаковых строк имеют одинаковые ключи. Введём для удобства две новые функции, ответственные за мемоизацию: `memo` и `memoWithKey`. Теперь можно переопределить `mCount`. Функции `memo`, `memoWithKey` и обновлённый `mCount` представлены на листинге 25. Для парсеров высшего порядка всегда необходимо указывать ключи явно.

Листинг 25 – Исправленный `mCount`

```
memo ::
  (Typeable s, Typeable a, Hashable s, Eq s) =>
  Parser s a -> Parser s a
memo p = Parser k (baseMemo k (parser p))
  where
    k = makeStableKey p

memoWithKey ::
  (Typeable s, Typeable t, Hashable s, Eq s) =>
  Key -> Parser s t -> Parser s t
memoWithKey k p = Parser k (baseMemo k (parser p))

getOrMakeKey :: (Typeable s, Typeable a) => Parser s a -> Key
getOrMakeKey (Parser (Just k) _) = k
getOrMakeKey p@(Parser Nothing _) = makeStableKey p

mCount ::
  Parser String String ->
  Int ->
  Parser String String
mCount p n = memoWithKey
  (Key (makeStableKey mCount, getOrMakeKey p, n)) $
  join <$> replicateM n p
```

Выводы по главе 2

В этой главе были проанализированы существующие алгоритмы, с помощью которых потенциально возможно достичь цели ВКР. Был выбран и до-

работан наиболее удачный алгоритм. Далее доработанный алгоритм был практически реализован с использованием языка программирования Haskell.

ГЛАВА 3. ПРИМЕНЕНИЕ CPS-ПАРСЕРОВ

В этой главе рассматриваются результаты, полученные в рамках второй главы. В том числе время работы полученного парсера, возможности его практического применения, а также ограничения.

3.1. Разбор арифметических выражений

Для оценки времени работы закодируем с использованием разработанной библиотеки парсер для языка арифметических выражений, заданного грамматикой с листинга 5. В качестве семантики языка используем алгебраические типы данных с листинга 26. Сам парсер представлен на листинге 27. Заметим, что код парсера в точности соответствует грамматике, несмотря на то, что она леворекурсивна. Это достигается за счёт использования функции `memo` при написании парсеров `f`, `term` и `expr`.

$$\begin{aligned}
 F &\rightarrow \text{number} \\
 &\quad (E) \\
 T &\rightarrow T * F \\
 &\quad T / F \\
 E &\rightarrow E + T \\
 &\quad E - T
 \end{aligned}$$

Рисунок 5 – Грамматика арифметических выражений

Листинг 26 – Семантические значения языка арифметических выражений

```

data Expr = ExprOp Expr String Term | ExprVal Term

data Term = TermOp Term String F | TermVal F

data F = FExpr Expr | FVal Int

```

Для сравнения парсер для языка арифметических выражений также был написан с использованием `Megaparsec` [11]. Так как `Megaparsec` не поддерживает левую рекурсию, грамматика для арифметических выражений предварительно была переписана в нелеворекурсивную форму тривиальным образом.

Помимо этого для нелеворекурсивной грамматики был написан парсер с использованием разработанной библиотеки, который не использует функцию

Листинг 27 – Парсер арифметических выражений

```

f :: Parser (ParserState T.Text) F
f =
  memo $
    FVal . either undefined fst . decimal <$> regex "[0-9]+"
    <|> FExpr <$> (single '(' *> expr <*> single ')')

term :: Parser (ParserState T.Text) Term
term =
  memo $
    TermVal <$> f
    <|> TermOp <$> term <*> (pure <$> oneOf ['*', '/']) <*> f

expr :: Parser (ParserState T.Text) Expr
expr =
  memo $
    ExprOp <$> expr <*> (pure <$> oneOf ['+', '-']) <*> term
    <|> ExprVal <$> term

exprStart :: Parser (ParserState T.Text) Expr
exprStart = expr <*> eof

```

мето. Это возможно, так как грамматика арифметических выражений принадлежит классу грамматик $LL(1)$, а значит, её возможно разобрать без использования поиска с возвратом. Вместе с тем переписанная грамматика нелеворекурсивна. Всё это делает использование мемоизации необязательным.

На рисунке 6 представлено время работы написанных парсеров в зависимости от длины разбираемого арифметического выражения. График «Мето CPS» показывает время работы мемоизированного CPS-парсера, график «Megaparsec» — время работы парсера на Megaparsec, график «CPS» — время работы CPS-парсера без мемоизации и возврата. По графику «Мето CPS» видно, что разработанная библиотека разбирает арифметические выражения, записанные леворекурсивно, за линейное время. Предположительно, линейное время работы будет наблюдаться на всех детерминированных грамматиках, как в случае с алгоритмом Эрли.

Тем не менее время работы мемоизированного парсера много больше времени работы парсера, закодированного с использованием Megaparsec. Такие различия объясняются необходимостью поиска в хеш-словаре, а также недетерминированностью оператора выбора ($<|>$). Вместе с тем по графику «CPS» видно, что разработанная библиотека поддерживает написание парсе-

ров в виде, при котором скорость работы сравнима со скоростью работы парсера на Megaparsec. Это достигается путём отказа от мемоизации и использования детерминированного оператора выбора.

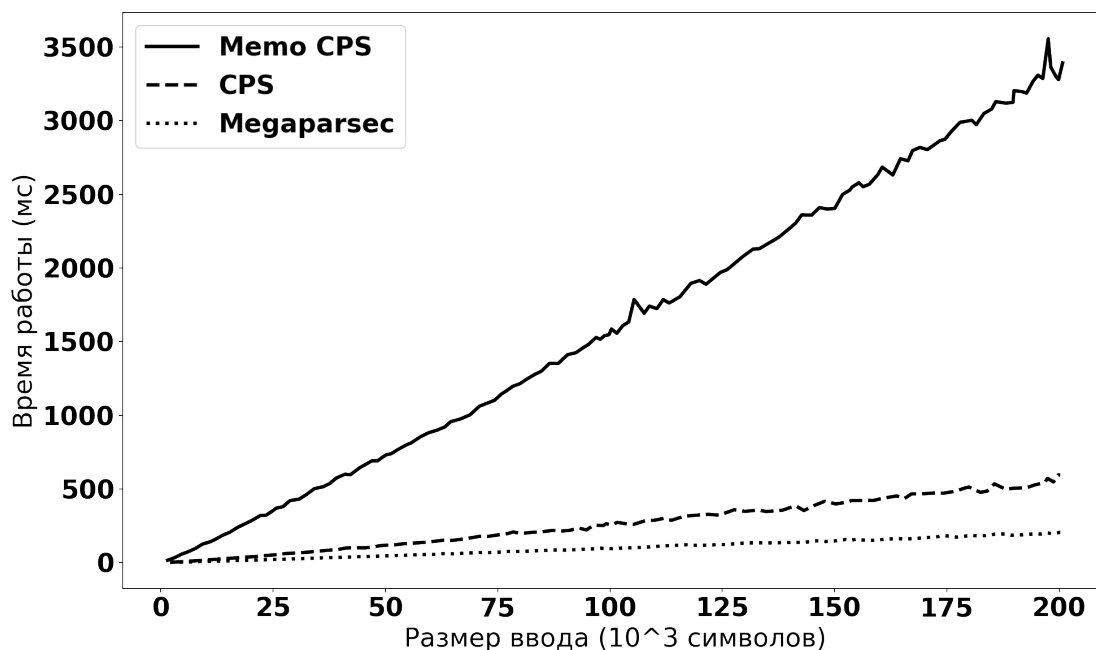


Рисунок 6 – Скорость работы парсеров арифметических выражений

Существенным недостатком мемоизированных CPS-парсеров является увеличенное потребление памяти, которое связано с необходимостью хранить промежуточные значения разбора. На таблице 1 приведено потребление памяти тремя парсерами при разборе арифметических выражений длиной 2000, 50000, 100000 и 200000 символов. По данным из таблицы видно, что потребление памяти мемоизированным CPS-парсером растёт линейно в зависимости от длины ввода.

Таблица 1 – Потребление памяти в зависимости от длины разбираемого арифметического выражения

—	Мемо CPS	Megaparsec	CPS
2000	12 МБ	6 МБ	7 МБ
50000	137 МБ	14 МБ	28 МБ
100000	271 МБ	19 МБ	43 МБ
200000	578 МБ	29 МБ	71 МБ

$$\begin{array}{l}
 A \rightarrow aAx \\
 \quad \quad aAy \\
 \quad \quad \in
 \end{array}$$

Рисунок 7 – Не $LL(k)$ грамматика

3.2. Разбор контекстно-свободных грамматик

Как показано в разделе 3.1, для арифметических выражений, чья грамматика записана в нелеворекурсивном виде, можно не использовать мемоизацию, при этом асимптотика времени работы парсера не изменится. Однако это суждение верно не для любой грамматики. Как было замечено в разделе 2.2.2 алгоритм CPS-парсеров гарантирует полиномиальное время работы парсера. Приведём пример, когда мемоизация даёт выигрыш даже для нелеворекурсивных парсеров. Рассмотрим грамматику с рисунка 7. Очевидно, что она не $LL(k)$: какое бы число символов 'а' парсер не принимал во внимание, существует строка, на которой он не сможет определить, какое правило вывода для нетерминала A применить. Время работы обычного комбинаторного парсера с возвратом при разборе такой грамматики равно $O(2^n)$ на вводах типа $a^n(x|y)^n$. Это время работы свойственно в том числе и Megaparsec, и разработанному парсеру при отсутствии мемоизации. Время работы разработанного немемоизированного CPS-парсера показано на графике «Unmemoized parser» с рисунка 8. Тем не менее при использовании мемоизированных CPS-парсеров время разбора становится квадратичным, что можно видеть на графике «Memoized parser» с рисунка 8. Графики «fit» — это графики функций, аппроксимированных на основании полученных измерений.

3.3. Разбор не контекстно-свободных грамматик

Важной особенностью написанного парсера является его монадичность, которая позволяет описывать с его помощью не только КСГ. Известно, что язык, порождаемый множеством строк, задаваемых регулярных выражением $a^k b^k c^k$ для любого натурального k , не является контекстно-свободным. Тем не менее с помощью разработанной библиотеки такой язык можно разобрать, используя парсер с листинга 28.

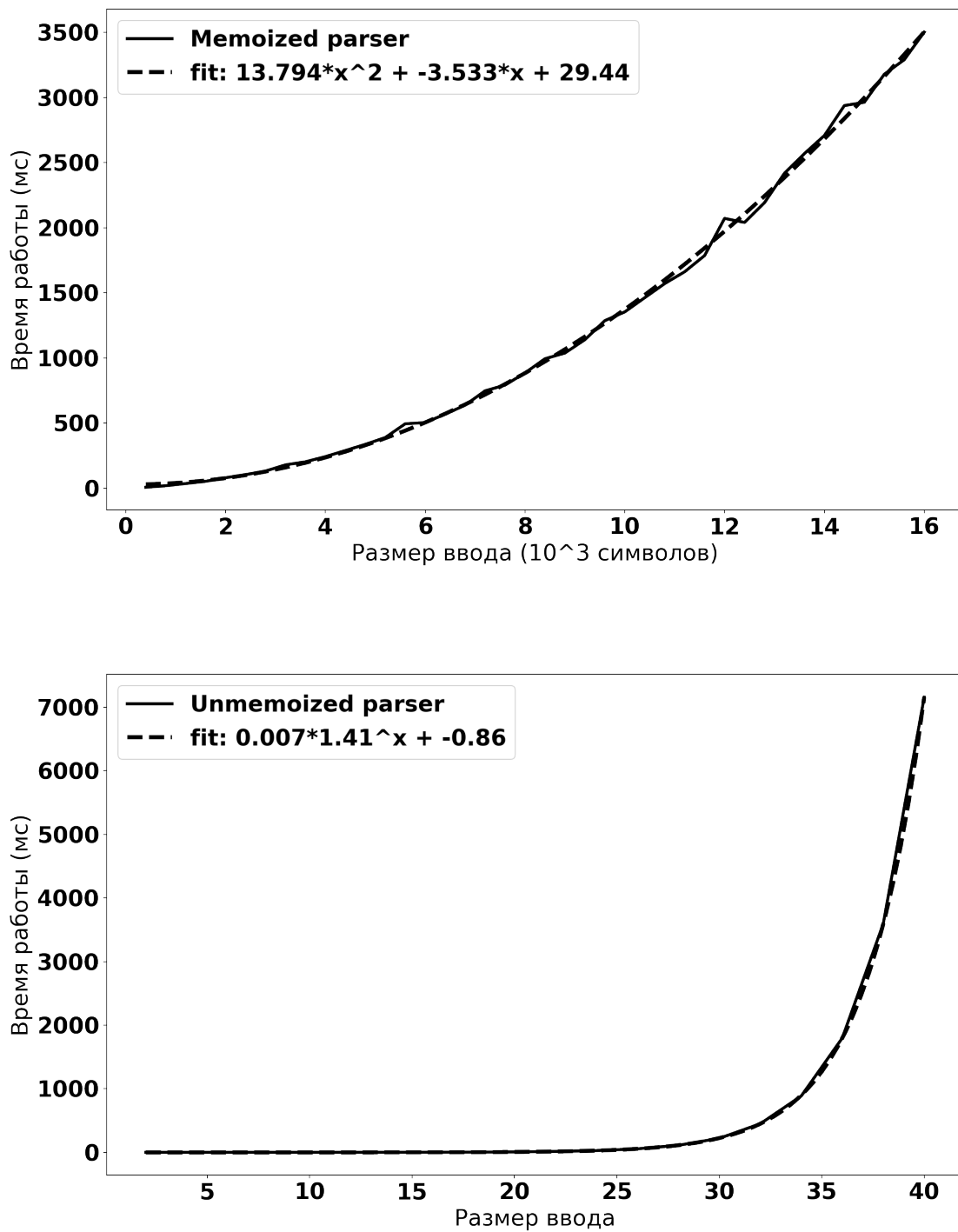


Рисунок 8 – Скорость работы CPS-парсера на не $LL(k)$ грамматике

3.4. Сравнение свойств разработанной библиотеки с существующими инструментами

В таблице 2 представлено сравнительное свойство некоторых инструментов синтаксического анализа, которые существуют на данный момент. Выбранные для сравнения инструменты либо популярны, либо обладают уникаль-

Листинг 28 – Парсер $a^k b^k c^k$

```

anbncn :: Parser (ParserState String) String
anbncn = do
  a <- some (single 'a')
  b <- replicateM (length a) (single 'b')
  c <- replicateM (length a) (single 'c')
  eof
  return $ a ++ b ++ c

```

ными свойствами. Важно заметить, что ни один из существующих комбинаторных парсеров не поддерживает левую рекурсию, пользовательскую семантику и при этом способен разбирать любую контекстно-свободную грамматику. Библиотека, написанная в рамках ВКР, называется в таблице «CPS».

Проблема таких инструментов как ANTLR и Bison заключается в том, что они являются генераторами, а не комбинаторами парсеров. Это приводит к проблемам, описанным в разделе 1.2.

Проблема Parsley [16] и подобных инструментов заключается во-первых в меньшей выразительной мощи в сравнении с монадическими инструментами анализа, а во вторых в отсутствии поддержки левой рекурсии. Инструменты типа PyParsing используют алгоритм Packrat [5], который подходит для разбора REG, что делает невозможным в общем случае комбинировать парсеры, написанные с использованием Packrat. Существующие монадические комбинаторные парсеры типа MegaParsec используют поиск с возвратом только в случае, когда это явно указано пользователем. Вместе с тем возврат без мемоизации приводит к экспоненциальному времени разбора на некоторых КСГ, что было показано в разделе 1.3.

Важно рассмотреть библиотеку Meerkat [10]. В её основе лежит тот же алгоритм, который используется для библиотеки, написанной в рамках ВКР. Тем не менее Meerkat оптимизирован для разбора неоднозначных грамматик, в связи с этим результатом разбора с использованием Meerkat служит SPPF (shared packed parse forest), что не позволяет пользователю задавать свою семантику для разбора.

Таблица 2 – Сравнительный анализ некоторых инструментов синтаксического анализа

	ANTLR	Bison	Parsley	PyParsing	MegaParsec	Meerkat	CPS
Тип	генератор		комбинатор				
Вид			аппликатив	монада			
Класс грамматик	LL(*)	КСГ	КСГ	PEG	КСГ		
Время работы	$O(n)$	$O(n^3)$	$O(n^3)$	$O(n)$	$O(n) - O(e^n)$	$O(n^3)$	$O(n^3)$
Лев. рекурсия	Нет	Да	Нет	Нет	Нет	Да	Да
Пользовательская семантика	Да	Да	Да	Да	Да	Нет	Да

ЗАКЛЮЧЕНИЕ

В рамках работы были рассмотрены недостатки существующих подходов к синтаксическому анализу формальных языков, заключающиеся в невозможности произвольной композиции парсеров. Было выявлено, что наиболее удачной формой для описания парсеров, поддерживающих композицию, являются комбинаторные парсеры. Было показано, что существующие комбинаторные парсеры не обладают в полной мере свойством композиционности в силу их чувствительности к левой рекурсии, а также неспособности большинства из них разбирать любые КСГ за полиномиальное время. В рамках работы на основе рекогнайзеров Джонсона был разработан комбинаторный парсер, поддерживающий левую рекурсию и разбирающий любые однозначные КСГ за полиномиальное время. Экспериментально было проверено, что разработанный парсер обладает этими свойствами. Таким образом, были выполнены все задачи ВКР. Разработанная в рамках ВКР библиотека обладает всеми заявленными свойствами.

Полученный результат можно применять для переиспользования частей парсеров, написанных для разбора различных языков, кроме того он открывает подход к модульному написанию парсеров, при котором парсеры более сложных языков можно собирать из парсеров для простых языков. Например, с помощью разработанной библиотеки предположительно возможно закодировать парсеры для семейства вложенных друг в друга языков с высокой степенью переиспользования кода. В дальнейшем планируется, используя написанную библиотеку, разработать парсер нетривиального языка программирования путём добавления к ядру языка дополнительных модулей. Кроме того в планах добавить обработку ошибок разбора.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

- 1 ANTLR [Электронный ресурс]. — URL: <https://www.antlr.org/> (visited on 04/22/2024).
- 2 Bison - GNU Project - Free Software Foundation [Электронный ресурс]. — URL: <https://www.gnu.org/software/bison/> (visited on 04/22/2024).
- 3 Data.Dynamic [Электронный ресурс]. — URL: <https://hackage.haskell.org/package/base-4.19.1.0/docs/Data-Dynamic.html> (visited on 05/05/2024).
- 4 *Earley J.* An efficient context-free parsing algorithm // Communications of the ACM. — 1970. — Feb. 1. — Vol. 13, no. 2. — P. 94–102. — ISSN 0001-0782. — DOI: 10.1145/362007.362035. — URL: <https://dl.acm.org/doi/10.1145/362007.362035> (visited on 05/08/2024).
- 5 *Ford B.* Parsing expression grammars: a recognition-based syntactic foundation // Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages. — New York, NY, USA : Association for Computing Machinery, 01/01/2004. — P. 111–122. — (POPL '04). — ISBN 978-1-58113-729-3. — DOI: 10.1145/964001.964011. — URL: <https://doi.org/10.1145/964001.964011> (visited on 05/11/2024).
- 6 *Frost R. A., Hafiz R., Callaghan P.* Parser Combinators for Ambiguous Left-Recursive Grammars // Practical Aspects of Declarative Languages. Vol. 4902 / ed. by P. Hudak, D. S. Warren. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2008. — P. 167–181. — ISBN 978-3-540-77441-9 978-3-540-77442-6. — DOI: 10.1007/978-3-540-77442-6_12. — URL: http://link.springer.com/10.1007/978-3-540-77442-6_12 (visited on 03/27/2024) ; Series Title: Lecture Notes in Computer Science.
- 7 Haskell/Continuation passing style - Wikibooks, open books for an open world [Электронный ресурс]. — URL: https://en.wikibooks.org/wiki/Haskell/Continuation_passing_style (visited on 04/27/2024).

- 8 *Hutton G., Meijer E.* Monadic Parser Combinators. — 1999. — Sept. 15.
- 9 *Izmaylova A., Afroozeh A., Storm T. V. D.* Practical, general parser combinators // Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (POPL '16: The 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages). — St. Petersburg FL USA : ACM, 01/11/2016. — P. 1–12. — ISBN 978-1-4503-4097-7. — DOI: 10.1145/2847538.2847539. — URL: <https://dl.acm.org/doi/10.1145/2847538.2847539> (visited on 03/27/2024).
- 10 *Johnson M.* Memoization of Top Down Parsing. — 04/25/1995. — DOI: 10.48550/arXiv.cmp-lg/9504016. — arXiv: cmp-lg/9504016. — URL: <http://arxiv.org/abs/cmp-lg/9504016> (visited on 05/11/2024).
- 11 megaparsec : Hackage [Электронный ресурс]. — URL: [//hackage.haskell.org/package/megaparsec](http://hackage.haskell.org/package/megaparsec) (visited on 04/22/2024).
- 12 *Nederhof M.-J.* A new top-down parsing algorithm for left-recursive DCGs // Programming Language Implementation and Logic Programming / ed. by M. Bruynooghe, J. Penjam. — Berlin, Heidelberg : Springer, 1993. — P. 108–122. — ISBN 978-3-540-47945-1. — DOI: 10.1007/3-540-57186-8_74.
- 13 *Norvig P.* Techniques for automatic memoization with applications to context-free parsing // Computational Linguistics. — 1991. — Mar. 1. — Vol. 17, no. 1. — P. 91–98. — ISSN 0891-2017.
- 14 System.Mem.StableName [Электронный ресурс]. — URL: <https://hackage.haskell.org/package/base-4.19.1.0/docs/System-Mem-StableName.html> (visited on 05/01/2024).
- 15 *Warth A., Douglass J. R., Millstein T.* Packrat parsers can support left recursion // Proceedings of the 2008 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation (PEPM08: Partial Evaluation and Program Manipulation). — San Francisco California USA : ACM, 01/07/2008. — P. 103–110. — ISBN 978-1-59593-977-7. — DOI: 10.1145/1328408.1328424. — URL: <https://dl.acm.org/doi/10.1145/1328408.1328424> (visited on 03/27/2024).

- 16 *Willis J., Wu N., Pickering M.* Staged selective parser combinators // Proceedings of the ACM on Programming Languages. — 2020. — Aug. 2. — Vol. 4, ICFP. — P. 1–30. — ISSN 2475-1421. — DOI: 10.1145/3409002. — URL: <https://dl.acm.org/doi/10.1145/3409002> (visited on 03/27/2024).