Санкт-Петербургский государственный университет

Кафедра системного программирования

Курбатова Зарина Идиевна

Применение парсер-комбинаторов для разбора булевых грамматик

Курсовая работа

Научный руководитель:
асп. кафедры системного программирования Подкопаев А. В.

Санкт-Петербург
2017

Оглавление

Введение **3**

1. Постановка задачи **5**

2. Обзор предметной области **6**

2.1. Парсер-комбинаторы 6

2.2. Булевы грамматики 7

2.3. Continuation Passing Style 8

2.4. Мемоизация вычислений синтаксического анализа 9

2.5. Поддержка левой рекурсии 9

2.6. Существующие решения 10

2.6.1. Whale Calf 10

2.6.2. Meerkat 11

3. Реализация **13**

3.1. Интерфейс библиотеки 13

3.2. Поддержка булевых грамматик 15

4. Апробация **19**

Заключение **23**

Список литературы **24**

Введение

Языки программирования имеют строгую синтаксическую структуру, которую можно успешно описать с использованием формальных грамматик. Согласно иерархии Н.Хомского [8], формальные грамматики делятся на 4 вида: неограниченные, контекстно-свободные, контекстно-зависимые и регулярные. Как известно, с помощью контекстно-свободных грамматик может быть описан весьма узкий класс языков. Существует расширение контекстно-свободных грамматик – булевы грамматики, предложенное А.Охотиным [1]. Булевы грамматики более выразительны, поскольку в правых частях правил вывода появляется возможность использовать конъюнкцию и отрицание.

Задачу синтаксического анализа можно сформулировать как задачу определения принадлежности слова некоторому языку. Иными словами, синтаксический анализатор принимает на вход строку и определяет, может ли строка порождаться грамматикой задаваемого языка. Одним из популярных подходов к реализации синтаксических анализаторов является парсер-комбинаторная техника: анализаторы представляются функциями высших порядков [5]. Парсер-комбинаторы – это функции высшего порядка, принимающие в качестве параметров анализаторы и возвращающие в качестве результата анализатор. Анализатор, в свою очередь, принимает в качестве входа строку и возвращает некоторый результат. При данном подходе определяются базовые комбинаторы, а затем с их помощью определяются более сложные. Например, в качестве базовых комбинаторов можно использовать комбинатор последовательного применения и комбинатор альтернативного применения. Основной проблемой наивной реализации подхода является невозможность использования левой рекурсии. Существуют различные реализации данной техники, поддерживающие левую рекурсию. Например, в работе [4] решение основано на алгоритме обобщенного синтаксического анализа GLL.

Синтаксический анализ находит применение во многих областях. В биоинформатике одной из основных задач является поиск в геноме

особых участков, кодирующих белок, тРНК и др. Геном представляет собой последовательность нуклеотидов или, другими словами, строку над алфавитом $\{A, C, G, T\}$. Искомые подстроки генома могут быть описаны с помощью контекстно-свободной грамматики. Таким образом, необходимо находить все строки, обладающие свойством выводимости. Например, вторичная структура РНК может быть описана с помощью контекстно-свободной грамматики, поскольку является своего рода языком палиндромов. Поэтому методы синтаксического анализа применимы при решении задач биоинформатики [6].

1. Постановка задачи

Целью данной работы является изучение применимости парсер-комбинаторной техники для разбора булевых грамматик. Для ее достижения были поставлены следующие задачи:

- реализация эффективной парсер-комбинаторной библиотеки для разбора булевых грамматик с поддержкой левой рекурсии;
- апробация библиотеки на примере контекстно-свободной грамматики, булевой грамматики и грамматики из биоинформатики;
- сравнение с существующими решениями.

2. Обзор предметной области

В данной главе определены основные понятия и приведено описание техник, использованных при реализации парсер-комбинаторной библиотеки.

2.1. Парсер-комбинаторы

В функциональном программировании представление анализаторов функциями и определение функций высших порядков (комбинаторов) является популярным подходом к построению нисходящих синтаксических анализаторов. При данном подходе определяются базовые комбинаторы и примитивные анализаторы, например, для разбора пустой строки и терминального символа.

Рассмотрим несколько таких комбинаторов.

Listing 1: Комбинатор seq

```
seq         :: Parser a → Parser b → Parser (a,b)
p 'seq' q = \s → [((v,w), s'') | (v, s')   ← p s
                                , (w, s'')  ← q s']
```

Комбинатор последовательного применения **seq** принимает на вход два анализатора p, q и возвращает анализатор, который последовательно запускает анализаторp и q. В качестве результата **seq** возвращает список пар вида (x, y), где x - пара из результатов разбора p и q, y - оставшаяся часть входного потока.

Listing 2: Комбинатор bind

```
bind        :: Parser a → (a → Parser b) → Parser  b
p 'bind' f = \s → concat[f v s' | (v, s') ← p s]
```

Анализатор p применяется к входной строке s и возвращает список пар вида (значение, строка). Функция f берет значение и возвращает анализатор, который применяется к каждому значению по очереди. В результате bind вернет список пар вида (значение, строка).

Однако наивная реализация этого подхода может иметь экспоненциальную сложность разбора, еще одним недостатком является невоз-

можность использования леворекурсивных правил.

2.2. Булевы грамматики

На практике часто встречаются языки, которые нельзя описать с помощью контекстно-свободных грамматик. Например, рассмотрим язык $\{a^m b^n c^n | m, n \geq 0, m \neq n\}$, которой не является контекстно-свободным. Его можно задать с помощью булевой грамматики:

$$\begin{array}{lcll} S & \rightarrow & AB \ \& \ \neg \ DC & \\ A & \rightarrow & aA & | \ \varepsilon \\ B & \rightarrow & bBc & | \ \varepsilon \\ C & \rightarrow & cC & | \ \varepsilon \\ D & \rightarrow & aDb & | \ \varepsilon \end{array}$$

Рис.1: Булева грамматика для языка $\{a^m b^n c^n | m, n \geq 0, m \neq n\}$

Булева грамматика - это четверка $G = (\Sigma, N, P, S)$, где:

- Σ – конечное непустое множество терминалов;
- N – конечное непустое множество нетерминалов;
- P – конечное множество правил.

Каждое правило имеет следующий вид:

$$A \rightarrow \alpha_1 \& ... \& \alpha_m \& \neg \beta_1 \& ... \& \neg \beta_n$$

$$(m + n \geq 1, \alpha_i, \beta_i \in (\Sigma \cup N)^*)$$

Правило A можно интерпретировать следующим образом: если строка представима в форме $\alpha_1, ..., \alpha_m$, но не представима в форме $\beta_1., ..., \beta_n$, то она выводится из нетерминала A.

Булевы грамматики расширяют контекстно-свободные грамматики, позволяя использовать в правых частях правил вывода конъюнкцию и отрицание.

2.3. Continuation Passing Style

Наивная реализация парсер-комбинаторного подхода имеет несколько ограничений. Во-первых, при наивном подходе имеет значение порядок альтернатив: анализатор завершает работу, пройдя по первой успешной альтернативе. Во-вторых, анализатор возвращает лишь одно дерево вывода. Поскольку потенциально анализатор может успешно завершить работу несколько раз на одной позиции во входном потоке, для разбора неоднозначных грамматик необходим исчерпывающий поиск, т.е. поиск всех возможных выводов строки. Одним из способов достичь этого является техника программирования в стиле передачи продолжений (Continuation Passing Style). Идея техники заключается в передаче управления через механизм продолжений. Продолжение представляет собой состояние программы, в которое может быть осуществлен переход из любой точки программы.

Рассмотрим простой пример – определение функции, возводящей число в квадрат.

Listing 3: Обычное определение функции

```
square :: Int → Int
square x = x*x
```

Listing 4: Определение функции в стиле передачи продолжений

```
square_cps :: Int → (Int → c) → c
square_cps x k = k (x*x)
```

У каждой функции есть дополнительный аргумент – функция, которой будет передан результат. В примере выше k – это продолжение, которому передается результат возведения числа в квадрат. Например, в качестве продолжения может быть передана функция, выводящая результат в консоль.

2.4. Мемоизация вычислений синтаксического анализа

Использование мемоизации при построении нисходящих синтаксических анализаторов для избежания экспоненциального времени анализа впервые было предложено в работе [9]. Идея техники заключается в сохранении результата работы анализатора для предотвращения повторых вычислений: если анализатор уже вызывалась на `i`-ой позиции, возвращается результат из таблицы, в противном случае анализатор запускается на позиции `i`, а результат затем помещается в таблицу.

2.5. Поддержка левой рекурсии

При наивной реализации парсер-комбинаторного подхода вызов анализатора на грамматике вида $E \to E\ E \mid a \mid \varepsilon$ не завершится, поскольку анализатор зациклится. В связи с этим становится актуальным вопрос поддержки левой рекурсии. В данном разделе описан подход к поддержке леворекурсивных правил. Впервые он был представлен в работе [7].

Заведем функцию `memo`, которая принимает на вход анализатор `f` и возвращает его мемоизированную версию, которая каждый раз при вызове на `i`-ой позиции обращается к таблице мемоизации. Если анализатор уже вызывался на `i`-ой позиции, то возвращается соответствующее значение, в противном случае вызывается функция memoresult с аргументом `f(i)`. Благодаря стратегии call-by-name анализатор `f` не запускается на позиции `i`, что гарантирует единственность вызова. Функция memoresult возвращает объект, у которого есть доступ к двум спискам: `Rs` хранит позиции, на которых работа анализатора завершилась успешно, `Ks` хранит все продолжения, переданные немемоизированному анализатору при вызове на `i`-ой позиции. Если анализатор вызван впервые, то текущее продолжение добавляется к списку `Ks`. При вызове анализатора на позиции `j`, которой нет в `Rs`, сначала `j` добавляется в `Rs`, затем запускаются все продолжения из `Ks` на `j`-ой позиции. Если

анализатор уже вызывался, то текущее продолжение добавляется к `Ks` и вызывается для каждой позиции из `Rs`.

Теперь, когда мемоизированный анализатор вызывается на `i`-ой позиции, его завершение будет гарантировано, поскольку не будет вызван на `i`-ой позиции больше одного раза.

2.6. Существующие решения

В данном разделе приведен обзор парсер-генераторной библиотеки с поддержкой булевых грамматик и двух парсер-комбинаторных библиотек.

2.6.1. Whale Calf

Whale Calf[1] - инструмент для разбора булевых грамматик, реализованный А.Охотиным на языке `C++`. Инструмент состоит из двух компонент: парсер-генератор, который преобразует текстовое описание конъюнктивных грамматик в исполняемый код, и библиотека, которая используется для анализа грамматик. В библиотеке реализовано несколько алгоритмов синтаксического анализа: табличный алгоритм для грамматик в бинарной нормальной форме, табличный алгоритм для грамматик в линейной нормальной форме, табличный алгоритм для произвольных грамматик, конъюнктивный LL, конъюнктивный LR и алгоритм, основанный на эмуляции автоматов, эквивалентных линейным конъюнктивным грамматикам.

Рассмотрим пример описания грамматики в контексте данной библиотеки.

Listing 5: Описание грамматики для языка $\{\omega c \omega | \omega \in a, b^*\}$

```
algorithm=LR;
terminal a, b, c;

S → C & D;
C → a C a | a C b | b C a | b C b | c;
```

[1] `http://users.utu.fi/aleokh/whalecalf/`

```
D → a A & a D | b B & b D | c E;
A → a A a | a A b | b A a | b A b | c E a;
B → a B a | a B b | b B a | b B b | c E b;
E → a E | b E | e;
```

Whalf Calf обрабатывает файл с описанием грамматики и создает два файла с определением анализатора на языке `C++` – `filename.cpp` и `filename.h`. Рассмотрим несколько методов, представленных в примере ниже:

- `read()` используется для передачи входной строки;
- `recognize()` используется для определения принадлежности строк языку;
- `parse()` используется для построения дерева вывода.

Listing 6: Пример использования анализатора

```
int x[] = {0, 1, 2, 0, 1}; // input "abcab"
whale_calf.read(x, x + 5);
WhaleCalf::TreeNode *tree = whale_calf.parse();
if (tree) whale_calf.print_tree(ofstream("parse_tree.dot"));
```

Данная реализация имеет сложность $O(n^4)$ по времени. Утверждается, что на практике работает быстрее, чем реализации других алгоритмов, имеющих в худшем случае сложность $O(n^3)$.

2.6.2. Meerkat

Meerkat[2] - парсер-комбинаторная бибилиотека с поддержкой всех контекстно-свободных грамматик, в том числе содержащих леворекурсивные правила, написанная на языке `Scala`. Библиотека позволяет строить дерево разбора за линейное время на грамматиках реальных языков программирования и за кубическое время в худшем случае. Помимо прочего, строится SPPF (Shared Packed Parse Forest) – графовое представление семейства деревьев разбора. Описание подхода к поддержке левой рекурсии описано авторами в работе [3].

[2] `http://meerkat-parser.github.io/`

Listing 7: Описание грамматики арифметических выражений в библиотеке Meerkat

```
val E: OperatorNonterminal
= syn ( right ( E ~ "^" ~ E )
      |> "-" ~ E
      |> left ( E ~ "*" ~ E
      | E ~ "/" ~ E )
      |> left ( E ~ "+" ~ E
      | E ~ "-" ~ E )
      | "(" ~ E ~ ")"
      | "[0-9]".r
      )
```

В языке Scala функции и переменные могут состоять из любых символов, например, можно создать функцию с именем $\sim$. В примере выше | - комбинатор альтернативного применения, $\sim$ - комбинатор последовательного применения. Для указания приоритета используется |>, для указания левой и правой ассоциативности используются left и right соотвественно. Терминальные символы могут быть представлены строками - ‘‘$-$″, ‘‘$+$″, ‘‘$*$″ и т.д.

3. Реализация

В данном разделе описана реализация парсер-комбинаторной библиотеки.

В качестве языка для реализации выбран Kotlin[3]. Kotlin - это молодой язык программирования, поддерживающий объектно-ориентированную и функциональную парадигмы, разрабатывается компанией JetBrains[4]. Поскольку Kotlin базируется на JVM, на ранних этапах разработки библиотеки была актуальна проблема переполнения стека из-за большого количества функциональных вызовов. Решить проблему удалось, воспользовавшись техникой программирования в стиле передачи продолжений (Continuation Passing Style).

3.1. Интерфейс библиотеки

Анализатор имеет тип Recognizer<A>, принимает на вход позицию во входном потоке и возвращает функцию типа CPSResult<A>. Эта функция принимает на вход продолжение типа K<A> и возвращает Unit. Тип Unit эквивалентен типу `void` в других языках программирования. Продолжение представляет собой следующий этап синтаксического анализа. Анализ производится вне зависимости от порядка альтернатив, таким образом обеспечивается исчерпывающий поиск.

Listing 8: Основные типы

```
typealias Recognizer<A> = (Int)      -> CPSResult<A>
typealias CPSResult <A> = (K<A>)     -> Unit
typealias K<A>          = (Int, A)   -> Unit
```

В библиотеке реализовано множество различных анализаторов и комбинаторов, среди которых можно выделить несколько базовых:

- terminal - принимает на вход строку s и проверяет, начинается ли входная строка с подстроки s;

[3] http://kotlinlang.org/

[4] http://jetbrains.com/

- eps – анализатор для пустой строки;
- rule – комбинатор альтернативного применения;
- seq – комбинатор последовательного применения;
- map – принимает на вход анализатор p и функцию f, применяет f к результату работы p;
- fix – комбинатор неподвижной точки, используется для определения рекурсивных правил;
- number – анализатор чисел;
- symbol – анализатор последовательности букв и чисел;
- paren – принимает на вход анализаторf и проверяет, заключен ли результат f в круглые скобки во входной строке.

В качестве примера работы с библиотекой рассмотрим реализацию анализатора для языка While. While – простой язык программирования, в котором определены следующие конструкции: присваивание, последовательная композиция, условный оператор и while.

Определение анализатора для разбора выражений приведено в листинге 9. Оператор "/" представляет собой синтаксический сахар для комбинатора альтернативного применения.

Listing 9: Анализатор выражений для языка While

```
1  val exprParser: Recognizer<Expr> = fix {
2      val corep = (number map { Expr.Con(it) as Expr }) /
3                  (symbol map { Expr.Var(it) as Expr }) /
4                  paren( sp(it) )
5      val op1p = rightAssocp(sp(terminal("^")), corep) {
6                      l, op, r → Expr.Binop(l, op, r)
7            }
8      val op2p = rightAssocp(sp(terminal("*") / terminal("/") /
9                 terminal("%")), op1p) {
10                     op, e1, e2 →
11                 Expr.Binop(op, e1, e2)
```

```
12             }
13     val op3p = assocp(sp(terminal("+") / terminal("-")), op2p) {
14                 op, e1, e2 ->
15             Expr.Binop(op, e1, e2)
16             }
17     return@fix op3p
18 }
```

3.2. Поддержка булевых грамматик

Для поддержки булевых грамматик необходимо определить две операции – конъюнкцию и отрицание. Комбинатор And, реализующий операцию конъюнкция, должен находить пересечение языков, порождаемых поступившими на вход анализаторами. Если входная строка принадлежит пересечению, то анализ завершается успешно. Комбинатор AndNot, реализующий операцию отрицание, завершает работу успешно, если входная строка принадлежит одному языку, а другому – нет. Ниже приведено подробное описание реализации обоих комбинаторов.

Комбинатор And принимает в качестве параметров два анализатора p1 и p2 и возвращает анализатор, который работает следующим образом:

- хранит два списка пар для p1 и p2 соответственно вида <Int, A>, где первый параметр - номер позиции во входном потоке, второй параметр - обработанный символ входного потока;
- запускает оба анализатора на i-ой позиции с продолжением, которое принимает пару вида <Int, A>, где первый параметр - pos - позиция во входном потоке, второй параметр - res - обработанный символ входного потока. Добавляет пару в соответствующий список пар для анализатора p1(p2). Проверяет, есть ли ли в соответствующем списке для анализатора p2(p1) пара, в которой первый элемент соответствует равен pos. Если да, то исходному продолжению передается пара, состоящая из pos и пары, которая состоит из pos и res;

- анализатор сообщает, что строка принадлежит пересечению конъюнктов, если порождается и анализатором p1, и анализатором p2.

Listing 10: Код комбинатора And

```
1  class And<A, B> (
2  val p1: Recognizer<A>, val p2: Recognizer<B>
3  ) : Recognizer<Pair<A, B>>() {
4      override fun invoke(p: Int): CPSResult<Pair<A, B>> {
5          val passedByp1: ArrayList<Pair<Int, A>> = ArrayList()
6          val passedByp2: ArrayList<Pair<Int, B>> = ArrayList()
7          val executed  : ArrayList<Pair<Int, Pair<A, B>>> = ArrayList()
8
9          return { k: K<Pair<A, B>> →
10             p1(p)({ pos: Int, res: A →
11                 val pair1p = Pair(pos, res)
12                 passedByp1.add(pair1p)
13                 passedByp2.forEach { r →
14                     if (pair1p.first == r.first) {
15                         val kpair = Pair(pair1p.second, r.second)
16                         k(pos, kpair)
17                         executed.add(Pair(pos, kpair))
18                     }
19                 }
20             })
21
22             p2(p)({ pos: Int, res: B →
23                 val pair2p = Pair(pos, res)
24                 passedByp2.add(pair2p)
25                 passedByp1.forEach { r →
26                     if (pair2p.first == r.first) {
27                         val kpair = Pair(r.second, pair2p.second)
28                         k(pos, kpair)
29                         executed.add(Pair(pos, kpair))
30                     }
31                 }
32             })
33         }
34     }
35 }
```

Строка выводится по правилу $S \to A\&\neg B$, если она может быть порождена из нетерминала A и не может быть порождена из нетерминала B. Таким образом, нужен комбинатор AndNot, проверяющий это условие. В качестве параметров комбинатор AndNot принимает два анализатора p1 и p2 и возвращает анализатор, который работает следующим образом:

- хранит два списка пар для p1 и p2 соответственно вида <Int, A>, где первый параметр - номер позиции во входном потоке, второй параметр - обработанный символ входного потока;
- запускает анализатор p1 на i-ой позиции с продолжением, которое добавляет пару типа <Int, A> в соответствующий список для p1. Эта пара передается продолжению k, если анализ прошел успешно;
- затем запускает анализатор p2 на i-ой позиции;
- если анализатор p2 не разобрал входную строку до той же позиции, что и анализатор p1, то анализ входной строки завершается успешно.

Listing 11: Код комбинатора AndNot

```
1  class AndNot<A, B> (
2  val p1: Recognizer<A>, val p2: Recognizer<B>
3  ) : Recognizer<A>() {
4      override fun invoke(p: Int): CPSResult<A> {
5          val passedByp1: ArrayList<Pair<Int, A>> = ArrayList()
6          val passedByp2: ArrayList<Pair<Int, B>> = ArrayList()
7          val executed   : ArrayList<Pair<Int, A>> = ArrayList()
8          return { k: K<A> →
9              p2(p)({ pos: Int, res: B →
10                 val pair2p = Pair(pos, res)
11                 passedByp2.add(pair2p)
12                 passedByp1.forEach { r →
13                     if(passedByp2.find{ it.first == r.first } == null){
14                         k(pos, r.second)
15                         executed.add(Pair(pos, r.second))
```

```
16                     }
17                  }
18              })
19
20              p1(p)({ pos: Int, res: A →
21                  val pair1p = Pair(pos, res)
22                  passedByp1.add(pair1p)
23                   k(pos, res)
24              })
25          }
26      }
27  }
```

4. Апробация

В данном разделе приведены результаты тестирования производительности парсер-комбинаторной библиотеки. Для тестирования были выбраны несколько грамматик. Эксперименты проводилось на машине со следующими характеристиками:

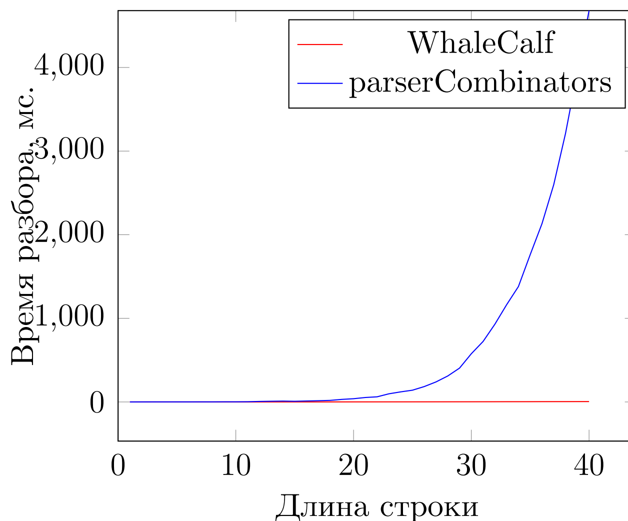
- OS – Ubuntu 16.04;
- CPU – Intel Core i5-4200M;
- RAM – 4GB.

Первая грамматика сильно неоднозначная, содержащая леворекурсивные правила, реализует худший случай для анализатора.

Listing 12: Грамматика для языка $\{a^*\}$

```
S → SSS | SS | a
```

Рис.2: Сравнение времени работы



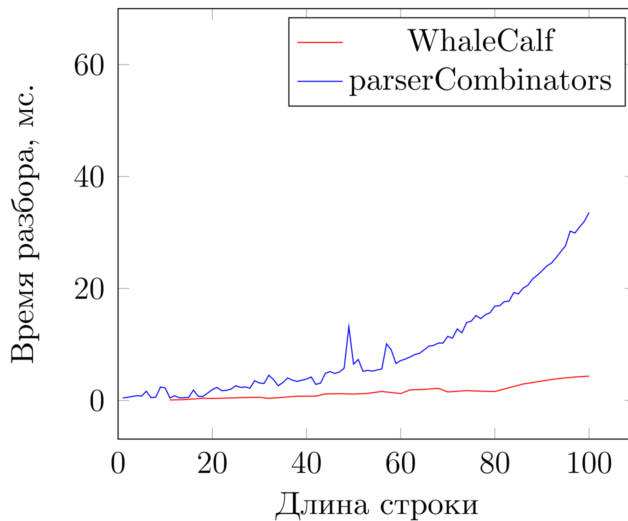
Результаты измерений представлены на рис.2. В библиотеке WhaleCalf реализовано несколько алгоритмов синтаксического анализа. Лучший результат демонстрирует алгоритм GLR, поэтому для сравнения производительности использовался он. Видно, что время работы алгоритма GLR растет значительно медленнее, чем время работы парсер-комбинаторов, с ростом длины входной строки.

Следующий эксперимент проводился на конъюнктивной грамматике, которая определяет требование declaration before use. Грамматика взята из работы [2]. Результаты измерений представлены на рис.3. Видно, что решение, основанное на алгоритме GLR, быстрее парсер-комбинаторов.

Listing 13: Грамматика для языка $\{u_1...u_n|n \geq 0, \forall i\ u_i \in da^* \ or \ u_i = ca^k \ and \ u_j = da^k \ \forall j < i,\ k \geq 0\}$

$$
\begin{array}{lllllll}
S & \to & SdA & | & ScA \ \& \ EdB & | & \varepsilon \\
A & \to & aA & | & \varepsilon & & \\
B & \to & aBa & | & E \ c & & \\
E & \to & EcA & | & EdA & | & \varepsilon
\end{array}
$$

Рис.3: Сравнение времени работы

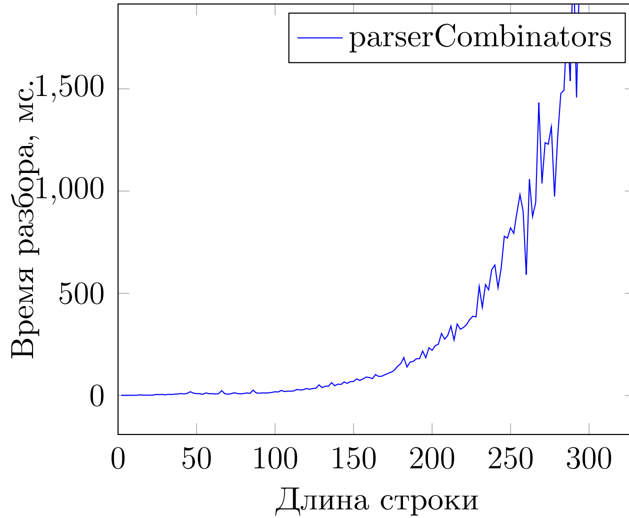


Также тестирование проводилось на булевой грамматике для языка $\{\omega\omega | \omega \in \{a, b\}^*\}$. Результаты представлены на Рис. 4.

Listing 14: Булева рамматика для языка $\{\omega\omega | \omega \in \{a, b\}^*\}$

$$
\begin{array}{lllll}
S & \to & \neg AB \ \& \ \neg BA \ \& \ C & & \\
A & \to & XAX & | & a \\
B & \to & XBX & | & b \\
X & \to & a & | & b \\
C & \to & XXC & | & \varepsilon
\end{array}
$$

Рис.4: Время работы

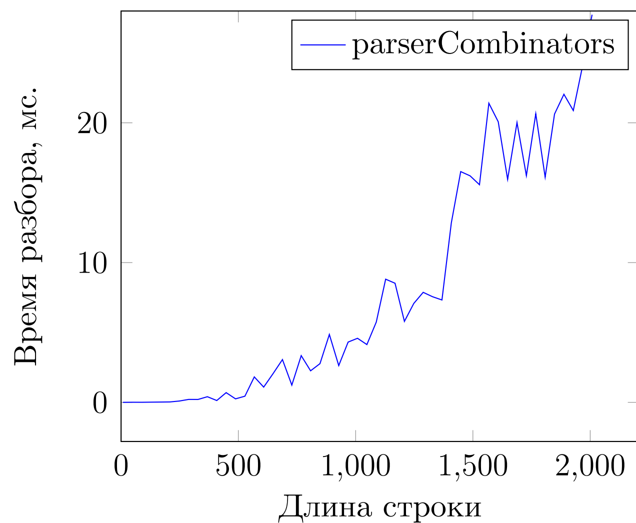


Базовая структура псевдоузла (pseudoknot), элемента вторичной структуры РНК, может быть представлена в виде языка $L_2 = \{\omega | \omega = [^i \ (^j \ ]^i \ )^j\}$, где пара скобок представляет собой пару оснований (base pair). Язык L_2 не является контекстно-свободным, однако он может быть представлен в виде пересечения языков $\{\omega | \omega = [^i \ (^* \ ]^i \ )^*\}$ и $\{\omega | \omega = [^* \ (^j \ ]^* \ )^j\}$.

Listing 15: Конъюнктивная рамматика для языка L_2

```
S  → S1 & S5
p1 → [
p2 → ]
p3 → (
p4 → )
S1 → S2 S3
S2 → p1 S2 p2 | S4
S4 → p3 S4 | ε
S3 → p4 S3 | ε
S5 → S6 S7
S7 → p3 S7 p4 | S8
S6 → p1 S6 | ε
S8 → p2 S8 | ε
```

Рис.5: Время работы



Результаты измерений представлены на Рис.5. Видно, что время работы растет достаточно быстро.

Заключение

В ходе работы получены следующие результаты:

- реализована парсер-комбинаторная библиотека с поддержкой булевых грамматик, в том числе содержащих леворекурсивные правила;
- проведена апробация библиотеки на примере контекстно-свободной грамматики, булевой грамматики и грамматики из биоинформатики;
- проведено сравнение с существующими решениями;
- результаты работы представлены на конференции "СПИСОК-2017".

Исходный код доступен в репозитории [10], автор вел работу под учетной записью *onewhl*.

В текущем виде библиотека не готова к промышленному использованию, поскольку необходимы оптимизации, способствующие повышению скорости работы парсер-комбинаторов.

Список литературы

[1] Alexander Okhotin. Boolean grammars // Information and Computation. — 2004. — Vol. 194, no. 1. — P. 19–48.

[2] Alexander Okhotin. Conjunctive and Boolean grammars: the true general case of the context-free grammars // Computer Science Review. — 2013. — Vol. 9. — P. 27–59.

[3] Anastasia Izmaylova, Ali Afroozeh, Tijs van der Storm. Practical, general parser combinators // Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation / ACM. — 2016. — P. 1–12.

[4] Daniel Spiewak. Generalized Parser Combinators. — 2010.

[5] Graham Hutton, Erik Meijer. Monadic parser combinators. — 1996.

[6] Manfred J Sippl. Biological sequence analysis. Probabilistic models of proteins and nucleic acids, edited by R. Durbin, S. Eddy, A. Krogh, and G. Mitchinson. 1998. // Protein Science. — 1999. — Vol. 8, no. 3. — P. 695–695.

[7] Mark Johnson. Memoization in top-down parsing // Computational Linguistics. — 1995. — Vol. 21, no. 3. — P. 405–417.

[8] Noam Chomsky. Three models for the description of language // IRE Transactions on information theory. — 1956. — Vol. 2, no. 3. — P. 113–124.

[9] Peter Norvig. Techniques for automatic memoization with applications to context-free parsing // Computational Linguistics. — 1991. — Vol. 17, no. 1. — P. 91–98.

[10] parserCombinators [Электронный ресурс]. — URL: `https://github.com/anlun/parserCombinators/` (дата обращения: 19.05.2017).