Министерство науки и высшего образования Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ

«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

РАЗРАБОТКА БИБЛИОТЕКИ КОМБИНАТОРНЫХ ПАРСЕРОВ ВЫСШЕГО ПОРЯДКА НЕЧУВСТВИТЕЛЬНЫХ К ЛЕВОЙ РЕКУРСИИ

Автор: Ступников Александр Сергеевич _____

Направление подготовки: 01.03.02 Прикладная математика и информатика

Квалификация: Бакалавр

Руководитель ВКР: Забашта А.С., доцент., к.т.н. _____

Санкт-Петербург, 2024 г.

Обучающийся Ступников Александр Сергеевич
Группа М34351 Факультет ИТиП

Направленность (профиль), специализация
Информатика и программирование

Консультанты:

а) Булычев Д.Ю., канд. физ.-мат. наук, доцент _____

ВКР принята «____» _____ 20___ г.

Оригинальность ВКР _____%

ВКР выполнена с оценкой _____

Дата защиты «____» _____ 20___ г.

Секретарь ГЭК Штумпф С.А. _____

Листов хранения _____

Демонстрационных материалов/Чертежей хранения _____

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*) $\alpha \to \beta$.

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 ситанксический анализ.

ж) **Семантика языка** — это смысловое значение конструкций языка. (функция, сопоставляющую строку, принадлежащую языку, некоторому значению).

и) **Входной поток** (ввод) — линейная последовательность токенов, которая передаётся парсеру для разбора.

к) **LL(k) грамматика** — грамматика, при разборе которой на основании k токенов входного потока можно однозначно определить правило вывода, которое необходимо применить.

л) **Поиск с возвратом** (англ. *backtracking*) — техника, при которой парсер возвращает поток ввода в исходное состояние после попытки разбора каждого правила при разборе нетерминала.

м) **Longest match first** — принцип проектирования парсера, при котором правило, при разборе которой способно поглотиться наибольшее число символов входного потока, должно быть опробовано первым при разборе нетерминала языка.

н) **Lookahead** — число символов входного потока, которое парсер учитывает при выборе правила в рамках разбора нетерминала.

п) **Предметно-ориентированный язык** (англ. *domain-specific language*, *DSL*) — формальный язык, специализированный для конкретной области применения.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Актуальность работы

Задачи синтаксического анализа повсеместно встречаются в информатике. Традиционно синтаксический анализ производится с использованием генераторов парсеров, однако такой подход обладает рядом недостатков. В частности, он предполгает использование чужеродной для программы синтаксического анализа структуры, не имеющей глубокой интеграции в язык, на котором эта программа написана.

Однако для синтаксического анализа используется и другой подход, лишённый этого недостатка генераторов парсеров: комбинаторные парсеры. С их помощью можно лаконичным и естественным образом выразить синтаксис и семантику языка, кроме того они полностью интегрированы в язык, использующейся для написания программы синтаксического анализа.

Тем не менее существующие реализации комбинаторных парсеров не обладают свойством композиционности. То есть нельзя в общем случае подставлять друг в друга грамматики, закодированные с использованием комбинаторов парсеров. В частности, невозможно распознавать леворекурсивные грамматики, которые могут возникать в результате использования парсеров высшего порядка из нелеворекурсивных грамматик. Такое положение вещей не позволяет раскрыть полный потенциал комбинаторных парсеров, затрудняя использование парсеров высшего порядка, а также заставляя кодировать леворекурсивные грамматики в неестественном виде.

Цель работы

Создание интрумента синтаксического анализа, позволяющего композировать грамматики произвольным образом.

Задача работы

Создание библиотеки комбинаторных парсеров, обладающих следующими свойствами:

а) Распознавание любой однозначной контекстно свободной грамматики.
б) Нечувствительность к longest match first.
в) Нечувствительность к левой рекурсии.
г) Полиномиальное время работы.

Структура работы

а) В Главе 1 представлено описание предметной области.

б) В Главе 2 представлен обзор существующих подходов, описание выбранного подхода, детали реализации решения.
в) В Главе 3 представлена верификация полученных результатов, даны примеры практического применения разработанного подхода.

ГЛАВА 1. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

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

1.1. Подходы к синтаксическому анализу

Для выполнения синтаксического анализа используют парсеры. По сути парсер это функция из некоторого состояния в множества пар из изменённого состояния и результата, то есть функция $S \to (S \times R)^*$, 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 продвинутых подходов является использование генераторов парсеров (таких как Bison[3] или ANTLR[2]), которые создают парсер на основе описания грамматики путём использования некоторого предметно-ориентированного языка. Другим продвинутым 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 написан парсер. Это означает, что комбинаторные парсеры, являясь обычными функциями, написаны на языке компилятора. 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. Грамматики, описанные в терминах комбинаторных парсеров, можно простым образом композировать, а части описания этих грамматик переиспользовать в нескольких парсерах.

1.3. Существующие практические реализации комбинаторных парсеров и их проблемы

Простейшая реализация комбинаторного парсера использует backtracking, чтобы иметь неограниченный lookahead и, как следствие, возможность разбирать произвольные КСГ. Однако использование backtracking может приводить к экспоненциальному времени разбора некоторых слов для ряда грамматик. Например, рассмотрим грамматику на рис. 1.

$$\begin{aligned} A \;\rightarrow\; & xAa \\ & xAb \\ & \epsilon \end{aligned}$$

Рисунок 1 – Грамматика с экспоненциальной сложностью

На вводах типа "xxxxbbbb" (которые можно записать регулярным выражением $x^n b^n$) сложность разбора в соответствии с грамматикой на рис. 1 растёт экспоненциально с ростом n. Действительно, при раскрытии нетерминала A парсер с backtracking 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 backtracking, только если пользователь явно укажет, что он необходим при разборе правила некоторого нетерминала (так делает megaprsec[7] с помощью кобинатора try). Это позволяет достигнуть линейного времени работы, однако использование этого подхода приводит к тому, что парсеры приходится писать с учётом longest match first, 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 композировать такие парсеры, ведь при произвольном объединении грамматик путём композирования функций-парсеров longest match first может перестать соблюдаться. Например, при объединении грамматик, состоящих из нетерминалов A и B с рис. 2, путём задания парсера для нетерминала C строку "somebody" нельзя будет полностью распознать, потому что её префикс "some" будет поглощён правилом $A \to some$, хотя грамматика из нетерминала B способна поглотить весь поток ввода.

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

Рисунок 2 – Неправильное объединение грамматик

Другим подходом, позволяющим комбинаторным парсерам с backtracking работать за время меньшее, чем экспоненциальное, является использование мемоизации. Причина экспоненциального времени работы парсеров с backtracking (как в том числе можно видеть из примера с рис. 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 $A \to aA|eps$ на строке "aaa" вернёт

[("aaa", "")("aa", "a"), ("a", "aa"), ("", "aaa")]. Для мемоизации парсера A необходимо завести таблицу, в которую при первом вызове 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 хэш 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[1]. Мемоизированные парсеры с backtracking способны разбрирать любую КСГ за полиномиальное время.

Тем не менее большинство мемоизированных комбинаторных парсеров с backtracking 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 нетерминала.

$$\underset{t}{E \rightarrow E + E}$$

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 простой

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 **апликативными**. Можно заметить, что тип любого парсера (то есть $S \to (S \times R)^*$), принадлежит к классу монад[5]. Монадические парсеры являются подмножеством апликативных, то есть с помощью монадического парсера всегда можно закодировать аппликативный. Монадические парсеры обладают большей выразительной мощью, 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 anbncn формируется во время работы программы, а не на этапе компиляции, и зависит от количества символов 'a', которое присутсвует 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 4 – Монадический парсер для не КС языка

```
anbncn :: Parser String String
anbncn = do
    a <- some (char 'a')
    b <- replicate (length a) (char '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


    c <- replicate (length a) (char 'c')
    return (a ++ b ++ c)
```

Выводы по главе 1

В этой главе были описаны основные подходы к созданию инструментов синтаксического анализа. Были разобраны тонкости реализации комбинаторных парсеров.

ГЛАВА 2. ПРЕДЛАГАЕМЫЙ ПОДХОД

В этой главе сначала даётся обзор существующих решений, которые потенциально могут решать задачу, поставленную в рамках работы, рассматриваются недостатки существующих решений. Далее предлагается собственное решение, разбираются детали его практической реализации.

2.1. Существующие решения

На текущий момент есть несколько решений, которые частично выполняют задачу, поставленную рамках работы. А именно есть несколько алгоритмов синтаксического анализа, позволяющих разбирать леворекурсивные контекстно-свободные грамматики с использованием комбинаторных парсеров. Рассмотрим их подробнее.

— **Использование длины остатка ввода** (Frost (2006)[4])
— **Cancellation разбор** (Nederhof (1994)[8])
— **Леворекурсивный разбор PEG** (Warth (2008)[9])
— **Парсеры в стиле передачи продолжения или CPS** (Johnson (1995)[6])

2.2. Предлагаемый подход

TODO

Выводы по главе 2

В этой главе были проанализированы существующие решения.

ГЛАВА 3. ПОЛУЧЕННЫЕ РЕЗУЛЬТАТЫ

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

ЗАКЛЮЧЕНИЕ

В данном разделе размещается заключение.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 *Norvig P.* Techniques for automatic memoization with applications to context-free parsing // Computational Linguistics - COLI. — 1991. — 1 янв. — Т. 17.

2 ANTLR [Электронный ресурс]. — URL: `https://www.antlr.org/` (visited on 04/22/2024).

3 Bison - GNU Project - Free Software Foundation [Электронный ресурс]. — URL: `https://www.gnu.org/software/bison/` (visited on 04/22/2024).

4 *Frost R. A.*, *Hafiz R.* A new top-down parsing algorithm to accommodate ambiguity and left recursion in polynomial time // ACM SIGPLAN Notices. — 2006. — May. — Vol. 41, no. 5. — P. 46–54. — ISSN 0362-1340, 1558-1160. — DOI: `10.1145/1149982.1149988`. — URL: `https://dl.acm.org/doi/10.1145/1149982.1149988` (visited on 03/27/2024).

5 *Hutton G.*, *Meijer E.* Monadic Parser Combinators. —.

6 *Johnson M.* Memoization of Top-down Parsing. —.

7 megaparsec : Hackage [Электронный ресурс]. — URL: `//hackage.haskell.org/package/megaparsec` (visited on 04/22/2024).

8 *Nederhof M. J.* Linguistic parsing and program transformations. — 1994. — 206 p. — ISBN 978-90-90-07607-2.

9 *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).