МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное автономное образовательное
учреждение высшего образования
ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ

Институт математики, механики и компьютерных наук
имени И. И. Воровича

Направление подготовки
Прикладная математика и информатика

МОНАДИЧЕСКИЙ ПАРСИНГ, ЧУВСТВИТЕЛЬНЫЙ К ОТСТУПАМ

Выпускная квалификационная работа
на степень бакалавра

студентки 4 курса
М. В. Втюриной

Научный руководитель:
асс. каф. ИВЭ А. М. Пеленицын

Ростов-на-Дону
2017

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Функциональные языки программирования имеют ряд преимуществ над императивными — программы обычно короче и проще для понимания, а языки имеют строгую типизацию, модульность, чистоту функций, отложенные (ленивые) вычисления.

Поведение чистых функций более предсказуемо благодаря отсутствию побочных эффектов. Стратегия вычислений, называющаяся ленивостью, заключается в том, чтобы отложить вычисление конечных значений до тех пор, пока не понадобится результат.

Среди функциональных языков *Haskell* достаточно распространён и одновременно современен. Чистые вычисления, поддержка ленивости, строгая, полная система типов — все это поддерживается языком программирования *Haskell*.

Функциональные языки используются для решения различных задач, в том числе и задачи первого этапа компиляции — синтаксического анализа или парсинга.

Популярный подход к построению функциональных парсеров состоит в том, чтобы моделировать их как функции и определять над ними функции высшего порядка (или комбинаторы). Комбинаторы, в свою очередь, реализуют грамматические конструкции, такие как последовательность, выбор и повторение. Осуществить такой подход можно с помощью монадных трансформеров — специальных типов, позволяющих комбинировать возможности нескольких монад в одной.

Реализация функционального парсера для языка программирования Gofer была предложена в статье *Monadic Parser Combinators* [1],

опубликованной Гремом Хаттоном и Эриком Мейером в 1996 году. Было принято решение модифицировать существующий подход для современной версии языка и использовать комбинаторы для решения проблемы чувствительности к отступам.

В данной работе были поставлены следующие задачи:

1. Разработать компактную библиотеку парсер-комбинаторов, чувствительных к отступам на основе монадных трансформеров
2. Создать базовые тесты библиотеки.
3. Интегрировать полученную библиотеку с парсером для подмножества языка Маркдаун.
4. Доработать парсер Маркдауна для разбора чувствительных к отсупам конструкций — вложенных списков.

ГЛАВА 1

ПРЕДВАРИТЕЛЬНЫЕ СВЕДЕНИЯ

1.1. Язык разметки Markdown

Markdown — это язык форматирования, по приницпу работы похожий на HTML, который используется для определения финального вида текста. Создан с целью написания максимально читабельного и удобного для правки текста, но пригодного для преобразования в языки для продвинутых публикаций.

Примеры выделения текста:

курсив -> *курсив*

полужирный -> **полужирный**

полужирный курсив -> ***полужирный курсив***

Markdown позволяет использовать заголовки в тексте. Доступны шесть уровней заголовков. (Начало строки с одного или более символов #)

Можно создавать маркированные списки, начиная каждую строку звездочкой и отделяя ее от текста пробелом. Аналогично создаются и нумерованные списки: каждая строка начинается с числа, после которого должны следовать точка, пробел и текст данного пункта. Допускается делать вложенные маркированный и нумерованные списки, а также смешивать их в одной структуре.

Указать, что часть текста является цитатой можно, начав каждую строку с угловой скобки (>). Этот символ выбран по той причине,

что многие почтоые программы используют именно его для выделения цитат. Результатом цитирования будет выделение абзаца отступом справа и слева.

1.2. Специальные монады

1.2.1. Reader

Монада Reader используется в случаях, когда необходимо передать какие-то настройки во множество функций, скрывая механизм передачи. [2] Вспомогательная функция *ask* получет информацию, которая передается, *runReader* запускает вычисление.

Листинг 1.2.1. Пример вычисления в монаде Reader [3]

```
newtype Reader r a = R { runReader :: r -> a }

comp :: Reader String Int
comp = ask >>= return.length

ghci> runReader comp "hello"
5
```

1.2.2. State

State — монада вычислений с сохранением и изменением состояния. Она возвращает какое-то значение и изменяет переменную состояния при необходимости. Вспомогательные функции для работы с состоянием — *put* и *get*. *Get* получает, а *put* модифицирует состояние. Чтобы запустить вычисление, необходимо вызвать *runState*.

Листинг 1.2.2. Пример вычисления в монаде State

```
newtype State s a = State { runState :: s -> (a, s) }

func :: State Int Int
func = do n <- get
put (n+2)
return n

ghci> runState func 5
(5,7)
```

Как правило, при работе с монадическими вычислениями не используют доступ к состоянию напрямую. Вместо этого применяют специальные функции для вычислений в контексте, такие как *bind* или другие специфические для конкретного монадического интерфейса методы (например, *ask* у монады *Reader*).

При создании данного проекта необходимо было получить доступ к информации, хранящейся в монадах *State* и *Reader* напрямую. Для этого были использованны функции *unS* и *unR* для *State* и *Reader* соответственно.

1.3. Функциональные парсеры

Парсер представляет собой функцию, которая принимает входной поток символов и выдает синтаксическое дерево разбора. Парсер может завершиться неудачно на входной строке, поэтому предпочтительнее, чтобы он возвращал список пар — (дерево, остаток строки), где пустой список на первой позиции означает неудачу. Различные парсеры могут возвращать различные виды деревьев.

Для парсинга в *Haskell* довольно часто используется парсер-комбинаторы, которые позволяют составлять функции более высокого порядка для генерации парсеров. Комбинаторы парсеров явля-

ются особенно выразительным шаблоном и обеспечивают быстрый и простой метод построения функциональных парсеров. Также можно встроить в парсер собственную логику.

1.3.1. Правило отступа (Offside rule)

Большую роль в обработке вложенных конструкций играет Offside rule или Правило отступа. Это правило позволяет группировать определения в программе с помощью отступов и, как правило, осуществляется лексером, который вставляет дополнительные маркеры (относительно отступа) в свой выходной поток. Другой подход к обработке данного правила — обработка с помошью специальных комбинаторов.

Листинг 1.3.1. Структура простой программы:

```
{a = b + c
 where
  {b = 10
   c = 15 - 5}
 d = a*2}
```

Суть правила заключается в следующем: последовательные определения, начинающиеся в одной колонке, считаются частью одной и той же группы **c**. Чтобы сделать парсинг проще, остальную часть каждого определения следует отнести к группе строго больше, чем *c*. Таким образом, с точки зрения Offside rule, определения *a* и *d* в программе выше сгруппированы вместе (для *b* и *c* аналогично), т.к. начинаются в одной колонне (Листинг 1.3.1).

1.4. Стек монад

Часто необходимо использовать возможности нескольких монад сразу. Специальные типы, монадные трансформеры, позволяют комбинировать несколько монад в одной. При комбинировании некоторые монады оказываются завернутыми в другие, так образуется монадный стек, по которому можно передвигаться с помощью функции *lift*, чтобы манипулировать значениями во внутренней монаде.

1.4.1. Пример с MaybeT

Листинг 1.4.1. Пример монадного трансформера MaybeT

```
newtype (Monad m) =>
        MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }

-- пример использования MaybeT
getPassword :: MaybeT IO String
getPassword = do
        lift $ putStrLn
        "Введите новый пароль:"
        s <- lift getLine
        guard (isValid s)
        return s

askPassword :: MaybeT IO ()
askPassword = do
        value <- msum $
        repeat getPassword
        lift $ putStrLn "Сохранение..."

main = runMaybeT askPassword
```

На листинге 1.4.1 представлен пример монадного трансформера MaybeT, который является оберткой вокруг m (Maybe a), где m может

быть любой монадой. Если взять в качестве m монаду IO( ), то получим трансформер ввода и вывода с возможной неудачей в вычислениях.

Далее представлено применение данного трансформера для задачи чтения пароля с клавиатуры и проверки его на устойчивость. [3]

lift - подъем функции до внутренней монады;

runMaybeT - запуск новой монады;

Для реализации моей работы понадобилось написание более сложных монадных трансформеров StateM и ReaderM. (Подробнее в п. 2.2)

1.5. Некоторые используемые расширения

MonadComprehensions — это расширение языка *Haskell*, которое обобщает представление списка до монады. Для подключения расширения необходимо добавить в файл директиву:

```
{-# LANGUAGE MonadComprehensions #-}
```

Листинг 1.5.1. Пример с заменой списковой монады на do-нотацию

```
[ x + y | x <- Just 1, y <- Just 2 ]

-- заменяется на:
do
   x <- Just 1
   y <- Just 2
   return (x+y)
```

Расширение *Multi-parameter type classes* позволяет определению класса типов иметь больше одного параметра. Директива для подключения:

```
{-# LANGUAGE MultiParamTypeClasses #-}
```

Расширение *FunctionalDependencies* используются для ограничения параметров классов типов. Оно позволяет утверждать, что в классе, содержащем несколько параметров, один из параметров может определяться из других. В представленном на листинге 1.5.2 примере *c* определяется через *a* и *b*. Директива для подключения:

```
{-# LANGUAGE FunctionalDependencies #-}
```

Листинг 1.5.2. Пример класса с параметром, который определен из двух других [4]

```
class Mult a b c | a b -> c where
  (*) :: a -> b -> c
```

Часто необходимо определить экземпляр класса, параметром которого могут быть объекты разных типов одновременно (Пример такого определения представлен на листинге 1.5.3). Данная возможность достигается при подключении расширения *FlexibleInstances*.

Директива для подключения:

```
{-# LANGUAGE FlexibleInstances #-}
```

Листинг 1.5.3. Пример определения класса с разными типами входных параметров

```
class MyClass a where
  action :: a -> Integer

instance MyClass Integer where
  action x = 1

instance MyClass Char where
  action x = 2

instance MyClass [Char] where
  action x = 3
```

ГЛАВА 2

ПОСТРОЕНИЕ ПАРСЕРА

2.1. Изменения для современной версии *Haskell*

Статья *Monadic Parser Combinators* [1], идеи из которой были использованы при создании программы, опубликованна в 1996 году Гремом Хаттном и Эриком Мейером. Все примеры, представленные в ней, были написаны на языке программирования *Gofer* — экспериметальном диалекте *Haskell*.

В настоящее время *Gofer* полностью влился в *Haskell* и не существует как самостоятельный язык, его можно встретить только в старых публикациях, поэтому было принято решение модифицировать методы, предложенные в статье. Наибольших изменений потребовала замена MonadOPlus на MonadPlus, т.к. для современной версии *Haskell*, чтобы описать экземпляр класса MonadPlus, нужно предварительно описать экземпляры Applicative и Alternative, чего не требовалось для реализации Monad0plus.

На листингах 2.1.1 и 2.1.2 представленны примеры описания экземпляра класса Monad0Plus (для *Gofer*) и MonadPlus соответственно.

Листинг 2.1.1. Пример описания Monad0Plus для *Gofer*

```
instance Monad m => Monad (StateM m s) where
    result v = \s -> result (v,s)
    stm `bind` f = \s -> stm s `bind` \(v,s') -> f v s'

instance Monad0Plus m => Monad0Plus (StateM m s) where
        -- zero :: StateM m s a
    zero = \s -> zero
        -- (++) :: StateM m s a -> StateM m s a -> StateM m s a
    stm ++ stm' = \s -> stm s ++ stm' s
```

Листинг 2.1.2. Пример описания MonadPlus для *Haskell*

```
instance Monad m => Applicative (StateM m s) where
    pure :: a -> StateM m s a
    pure = return
    (<*>) = ap

instance MonadPlus m => Alternative (StateM m s) where
    empty     = StateM $ const mzero
    s1 <|> s2 = StateM $ \s -> unS s1 s <|> unS s2 s

instance MonadPlus m => MonadPlus (StateM m s) where
    mzero = empty
    mplus = (<|>)
```

Таблица 2.1. — Изменения в *Haskell*

Gofer	Современный Haskell
result	return
bind	»=
MonadOplus	MonadPlus
zero	mzero
(++)	mplus
no equivalent	Alternative
no equivalent	Applicative
no equivalent	unR
no equivalent	unS

Таблица 2.1 содержит основные изменения, которые я встретила при работе над своим проектом.

2.2. StateM и ReaderM

Возможности языка *Haskell* позволяют определять собственные типы данных, поэтому для построения удобного типа парсера введем несколько новых типов.

StateM применяет данный конструктор типа *m* к результату вычисления.

Листинг 2.2.1. Определение StateM

```
newtype StateM m s a = StateM { unS :: s -> m (a,s) }
  deriving Functor
class Monad m => StateMonad m s | m -> s
  where
      update :: (s -> s) -> m s
      set    :: s -> m s
      fetch  :: m s
      set s   = update $ const s
      fetch   = update id
```

Экземпляры класса, описанного в листинге 2.2.1, имеют операции обновления — *update* , установки — *set* и извлечения — *fetch*.

Конструктор типа ReaderM (Листинг 2.2.2) можно сделать монадой аналогично StateM. Операция *env* возвращает состояние в результате вычисления, а *setenv* заменяет текущее состояние на новое (*env* от environment).

Листинг 2.2.2. Определение ReaderM

```
type ReaderM m s a = s -> m a

instance Monad m => ReaderMonad (ReaderM m s) s
  where
      env = ReaderM $ \s -> return s
      setenv s srm = ReaderM $ \_ -> unR srm s
```

2.3. Тип парсера

Парсер объединяет два вида вычислений: недетерминированные вычисления (результат парсера — список возможных вариантов разбора входной строки) и вычисления с состоянием (состояние — это обрабатываемая строка).

Для реализации правила отступа во время синтаксического анализа будем запоминать некоторую дополнительную информацию. Прежде всего, парсеру нужно будет знать номер столбца первого символа во входной строке и потребуется номер текущей строки. Тогда тип:

```
type Parser a = StateM [] String a
```

Если состояние парсера (строка, столбец), то:

```
type Parser a = StateM [] Pstring a
type Pstring  = (Pos,String)
type Pos      = (Int,Int)
```

Также парсеру нужно знать номер столбца текущего определения. Если правило отступа не действует, то номер этой позиции может быть отрицательным. Адаптируем тип под вышеизложенный случай:

```
type Parser a = Pos -> StateM [] Pstring a
```

Используя **ReaderM**, снова пересмотрим тип парсера:

```
type Parser a = ReaderM (StateM [] Pstring) Pos a
```

Пример парсера *item*, который успешно обрабатывает первый символ, если входная строка не пуста, представлен на листинге 3.2.2.

Вспомогательная функция *newstate* рассматривает первый символ входной строки и обновляет текущую позицию (например, если символ новой строки был поглощен, текущий номер строки увеличивается на единицу, а текущий номер столбца устанавливается на ноль). Функция *onside* проверяет, находится ли внутри позиция рассматриваемого символа относительно определения. (Листинг 3.2.1)

Листинг 2.3.1. Функции onside и newstate

```
onside :: Pos -> Pos -> Bool
onside (l,c) (dl,dc) = (c > dc) || (l == dl)

newstate :: Pstring -> Pstring
   newstate ((l,c),x:xs)
      = (newpos,xs)
        where
           newpos = case x of
                      '\n' -> (l+1,0)
                       '\t' -> (l,((c 'div' 8)+1)*8)
                       _    -> (l,c+1)
```

Листинг 2.3.2. Парсер item

```
item  = p <|> zero where
        p = do
            (pos, inp) <- update newstate
            guard $ not $ null inp
            defpos       <- env
            guard $ onside pos defpos
            return $ head inp
```

2.3.1. Реализация Offside rule

Для offside rule пробелы и комментарии не важны, но тем не менее они тоже должны быть обработанны. Это можно сделать с помощью *junk* парсера:

Листинг 2.3.3. Парсеры comment, spaces и junk

```
comment :: Parser ()
comment = [() | _ <- string "--"
            , _ <- many (sat (\x -> x /= '\n'))]

spaces :: Parser ()
spaces = [() | _ <- many1 (sat isSpace)]
    where
        isSpace x =
            (x == ' ') || (x == '\n') || (x == '\t')

 junk :: Parser ()
 junk  = [() | _ <- setenv (0,-1) (many (spaces +++ comment))]
```

Комбинатор *many1offside* разбирает последовательность определений. Вспомогательный комбинатор *off* настраивает положение для каждого нового определения последовательности (если позция столбца не изменилась). *Manyoffside* делает то же самое, что и *many1offside*, но допускает разбор пустой последовательности.

Листинг 2.3.4. Реализация many_offside

```
many1_offside  :: Parser a -> Parser [a]
many1_offside p = [vs | (pos,_) <- fetch
                      , vs      <- setenv pos (many1 (off p))

off  :: Parser a -> Parser a
off p = [v | (dl,dc)   <- env
           , ((l,c),_) <- fetch
           , c == dc
           , v         <- setenv (l,dc) p]

many_offside :: Parser a -> Parser [a]
many_offside p = many1_offside p <|> return []
```

ГЛАВА 3

ПРОГРАММНАЯ РЕАЛИЗАЦИЯ

3.1. Проект ParserCombinators

Основоной идеей для данного проекта стало приложение к статье *Monadic Parser Combinators* [1], в котором описан метод создания парсера для определения типа данных.

Код программной реализации разбит на модули:

- **StatesReaders.hs**: содержит описание монад ReaderM и StateM.
- **ParserCombinators.hs**: включает в себя описание типа парсера и базовые парсер-комбинаторы.
- **ParserDataDefinition.hs**: содержит специальные парсеры для разбора определения типа данных.
- **BasicTests.hs**: содержит базовые тесты для проекта.

Полный код проекта доступен в репозитории с исходными кодами [5].

В файле input.txt содержатся входные данные. Для работы с программой необходимо перейти в каталог проекта и задать команду **make** с ключом:

main — для сборки проекта

test — для запуска тестирования

check — для запуска программы и анализа входного файла

Могут быть распознаны функция, операция применения, переменная, конструктор типа, кортеж и список.

Листинг 3.1.1. Распознаваемый тип данных

```
data Type = Arrow Type Type -- function
          | Apply Type Type -- Application
          | Var String      -- variable
          | Con String      -- constructor
          | Tuple [Type]    -- Tuple
          | List Type       -- List
          deriving (Show, Eq)
```

Основной парсер типа данных определен рекурсивно (Листинг 3.1.4). Функции *chainr1* и *chainl1* анализируют непустые последовательности элементов, разделенные операторами, право и лево ассоциативными, соответственно. В статье для определения этих функций использовалась монада списка, которую я заменила на более простую для понимания *do*-нотацию (Листинг 3.1.3).

Листинг 3.1.2. Функции chainl1 и chainr1

```
chainr1 :: Parser a -> Parser (a -> a -> a) -> Parser a
p `chainr1` op = rec <|> p where
      rec = do
          x <- p
          f <- op
          y <- p `chainr1` op
          return $ f x y

chainl1 :: Parser a -> Parser (a -> a -> a) -> Parser a
p `chainl1` op = p >>= rest where
  rest x = one x <|> return x
  one x = do
      f <- op
      y <- p
      rest (f x y)
```

Листинг 3.1.3. Парсер типа данных

```
type0 :: Parser Type
type0 = type1 `chainr1` (symbol "->" >> return Arrow)

type1 = type2 `chainl1` return Apply

type2 = var +++ con +++ list +++ tuple
```

3.1.1. Примеры работы программы

На листингах 3.1.4 и 3.1.5 представлены два примера работы программы для разных определений — списка и дерева.

В результате получено синтаксическое дерево разбора типа данных, причем, отступы не считаются разрывами определения.

Парсинг закончен на позиции (2,0) в первом примере и (3,0) во втором. В обоих примерах строка полностью проанализирована. В случае неудачи программа выведет часть строки, которая не была разобрана.

Листинг 3.1.4. Пример 1

```
input.txt:
data List a = Nil
        | Cons a (List a)

> make check

[([("List",["a"],[("Nil",[]),("Cons",[Var "a",Apply (Con "List")
(Var "a")])])],((2,0),""))]
```

Листинг 3.1.5. Пример 2

```
input.txt:
data Tree a b =
            Leaf a
            | Node (Tree a b, b, Tree a b)

> make check

[([("Tree",["a","b"],[("Leaf",[Var "a"]),
("Node",[Tuple [Apply (Apply (Con "Tree") (Var "a")) (Var "b"),
Var "b", Apply (Apply (Con "Tree") (Var "a"))
(Var "b")]])])],((3,0),""))]
```

3.2. Интеграция с проектом Markdown-monparsing

После создания библиотеки была поставлена задача использовать полученные результаты для создания парсера Markdown, чувствительного к отступам.

За основу был взят проект Лукьянова Г. Markdown-monparsing [6] — ковертер файлов из Markdown в HTML. В данном проекте, кроме прочего, реализован парсинг некоторых простых конструкции Markdown, например, заголовки, цитаты, списки.

Было принято решение интегрировать тип парсера из п. 2.3 в проект Markdow-monparsing. Таким образом, пришлось заменить файл Parsers.hs, содержащий базовые парсеры, на два модуля моей библиотеки — StatesReaders и ParserCombinators.

Интеграция прошла успешно, все возможности проекта проверены тестированием.

ЗАКЛЮЧЕНИЕ

При реализации проекта была создана компактная библиотека парсер-комбинаторов, чувствительных к отступам. Основная идея заимствованна из статьи *Monadic Parser Combinators* [1], написанной Гремом Хаттоном и Эриком Мейером.

При создании программы активно использовались сильные стороны функционального программирования. Особенности языка *Haskell*, такие как чистота функций, поддержка ленивости, полнота и строгость системы типов, позволили решить нетривиальные задачи с помощью достаточно простого и короткого кода.

Для соответствия стандартам современного *Haskell* нужным образом были изменены экземпляры классов Monad и MonadPlus, также добавлены недостающие экземпляры классов там, где это было необходимо.

Работа демонстрирует удобство использования парсер-комбинаторов для синтаксического анализа контекстно-зависимых конструкций, в данном случае, чувствительных к отступам.

СПИСОК ЛИТЕРАТУРЫ

1. *Hutton G.*, *Meijer E.* Monadic Parser Combinators. — 1996. — URL: http://unpetitaccident.com/pub/compeng/languages/Haskell/monparsing.pdf.
2. *Липовача М.* Изучай Haskell во имя добра! — 2012.
3. Лекции по функциональному программированию. — URL: http://edu.mmcs.sfedu.ru/course/view.php?id=241 (дата обр. 09.12.2016).
4. Описание расширения MonadComprehensions. — URL: https://ghc.haskell.org/trac/ghc/wiki/MonadComprehensions (дата обр. 16.02.2017).
5. Репозиторий с кодом программы. — URL: github.com/MaryVtyurina/Parser_Combinators (дата обр. 26.05.2017).
6. Проект markdown-monparsing. — URL: https://github.com/geo2a/markdown-monparsing.