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

Кафедра информатики и вычислительного эксперимента

ФУНКЦИОНАЛЬНЫЙ ПАРСЕР ЛЕГКОВЕСНОГО ЯЗЫКА РАЗМЕТКИ MARKDOWN НА ОСНОВЕ КОМБИНИРОВАНИЯ МОНАД И МОНОИДАЛЬНОГО ПРЕДСТАВЛЕНИЯ ИСХОДНОГО ТЕКСТА

Выпускная квалификационная работа
на степень бакалавра

Студента 4 курса
Г. А. Лукьянова

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

Ростов-на-Дону
2015

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В работе рассматриваются технологии функционального программирования, предназначенные для описания вычислений с побочными эффектами, производится сравнение методов построения вычислений с несколькими побочными эффектами, и приводятся способы их использования для построения библиотек функциональных парсеров.

В качестве отправной точки для разработки библиотеки монадических комбинаторов парсеров используются результаты статьи [1]. Описание вычислений с несколькими побочными эффектами производится с помощью трансформеров монад [2], а также расширяемых эффектов [3] — альтернативного трансформерам монад подхода. Для построения полиморфных по входному потоку парсеров применяются специализированные моноиды, представленные в статье [4].

ГЛАВА 1

ПРЕДВАРИТЕЛЬНЫЕ СВЕДЕНИЯ

1.1. Элементы функционального программирования и теории категорий

Функциональное программирование — парадигма программирования, которая трактует программу как вычисление значения некоторой математической функции.

Корни функционального программирования уходят в λ-исчисление, формальную систему, разработанную в 1930-х годах для решения Entscheidungsproblem [5].

Одним из главных преимуществ функциональных языков программирования считается высокий уровень абстракции, выразительность и высокий коэффициент повторного использования кода. Для достижения этих свойств в языках реализуются такие средства как параметрический полиморфизм, функции высших порядков, каррирование, алгебраические типы данных, классы типов и другие. Использование неизменяемых данных позволяет существенно упростить отладку программ.

1.1.1. Функции высших порядков

Ключевой концепцией функционального программирования являются так называемые функции высших порядков (англ. higher-

order function). Они представляют собой функции, в качестве параметров которых могут выступать другие функции.

В качестве примера рассмотрим одну из функций стандартной библиотеки языка Haskell (листинг 1.1).

```
map :: (a -> b) -> [a] -> [b]

ghci> map (+3) [1,5,3,1,6]
[4,8,6,4,9]
```

Листинг 1.1: Функция для трансформации списка

Как видно из типовой аннотации, она принимает на вход функцию, которая преобразует значение типа `a` к значению типа `b`, и список значений типа `a`, возвращаемым значением является список результатов применения функции к значениям из входного списка.

Функция, рассмотренная в примере, обладает ценным свойством: она является полиморфной по своему параметру и выходному значению. Возможность написания параметрически полиморфных функций позволяет при программировании на `Haskell` повысить повторное использование кода и выразительность, чему способствует также простота синтаксиса, применяемого для описания таких функций.

1.1.2. Алгебраические типы данных

Алгебраические типы данных — минималистический, но выразительный способ описания типов данных, применяемый в функциональных языках программирования. Выделяют типы-перечисления или типы-суммы — аналог перечислений `enum` из императивных языков, и типы-контейнеры или типы-произведения — аналог структур `struct` или записей `record`.

Рассмотрим простой тип-сумму, используемый в языке `Haskell` для представления булевых констант:

```haskell
data Bool = False | True
```

Листинг 1.2: Элементарный тип-сумма

Этот тип распадается на два конструктора значений (англ. value constructors), экземпляр значения типа **`Bool`** может являться одним из возможных значений (**`True`** или **`False`**).

Теперь рассмотрим тип-произведение и пример создание константы этого типа:

```haskell
data Person = Person { firstName :: String
                     , lastName :: String
                     , age :: Int
                     }

p = Person { firstName = "John"
           , lastName  = "Lennon"
           , age = 43
           }
```

Листинг 1.3: Элементарный тип-произведение

Алгебраические типы данных могут быть параметризованы — это позволяет создавать, например, полиморфные контейнеры.

Рассмотрим определение списка, аналогичное определению из стандартной библиотеки языка Haskell:

```haskell
data List a = Nil | Cons a (List a)
```

Листинг 1.4: Определение списка

Здесь **`List`** является конструктором типа (type constructor) с одним типовым параметром, а `Nil` и `Cons` — конструкторами значений. Стоит также заметить, что **`List`** является рекурсивным типом.

1.1.3. Классы типов

Классы типов позволяют накладывать ограничения на типы, определяя некоторый набор операций, которые могут производиться

над типами, принадлежащими к некоторому классу. Иными словами, класс типов определяет интерфейс, через который можно взаимодействовать с типом.

Рассмотрим определение стандартного класса типов языка Haskell, отвечающего за возможность сравнения на равенство:

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

Листинг 1.5: Класс типов, для которых введено отношение эквиваленции

Чтобы воспользоваться возможностями класса типов для какого-то конкретного типа необходимо объявить этот тип экземпляром класса типов, во многих случаях это может быть сделано неявно, благодаря автоматическому порождению экземпляров компилятором GHC.

Возможность автоматического порождения экземпляра нужного класса типов для алгебраического типа данных:

```
data Numbers = One | Two | Three deriving Eq

ghci> One /= Two
True
```

Листинг 1.6: Автоматическое порождение экземпляров классов типов

В некоторых случаях необходимо описать экземпляр явно, такая возможность тоже существует.

```
data Foo = Foo {x :: Integer, str :: String}

instance Eq Foo where
  (Foo x1 str1) == (Foo x2 str2) = (x1 == x2) &&
                                   (str1 == str2)
```

Листинг 1.7: Явное описание экземпляра класса типов

Как уже говорилось ранее, код, написанный на языке программирования Haskell имеет очень высокий уровень абстракции. Далее будут рассмотрены два классы типов, представляющие структуры,

введённые в язык Haskell под влиянием исследований в области связи теоретической информатики и теории категорий. Эти структуры представляют абстракцию для вычислений с побочными эффектами. В дальнейшем изложении эти классы типов будут использоваться для построения парсеров.

1.1.4. Функтор

Рассмотрим определение класса типов **Functor** из стандартной библиотеки Haskell:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```

Листинг 1.8: Класс типов **Functor**

Как видно из определения, каждый тип, являющийся функтором, должен предоставлять функцию fmap.

Сформировались два основных неформальных описания функтора: контейнер, содержащий в себе значения определённого типа и вычисление, которое производится в некотором контексте. Согласно первой из этих трактовок, функция fmap применяет подаваемую ей на вход функцию к значениям в контейнере и возвращает изменённые значения, с сохранением структуры контейнера. Если же говорить в терминах вычислительных контекстов, то fmap модифицирует вычисление в контексте, но сам контекст остаётся неизменным.

Упомянутые выше списки языка Haskell являются функторами, для них в качестве fmap можно выбрать функцию **map**.

Важно заметить, что не любой тип, для которого определена функция fmap является функтором, необходимо также потребовать выполнения двух уравнений, называемых законами функтора:

Здесь **id** — тождественная функция, а . — инфиксный оператор композиции функций. Эти уравнения отражают теоретико-категорное определение функтора как отображения между катего-

```
fmap id = id
fmap (g . h) = (fmap g) . (fmap h)
```

Листинг 1.9: Законы функтора

риями, сохраняющее единичный морфизм и композицию. В случае `Haskell` рассматриваются эндофункторы над категорией `Hask` — категорией типов языка `Haskell`.

1.1.5. Аппликативный функтор

Аппликативный функтор является специализацией функтора, допускающей применение функции, находящейся внутри некоторого контекста, к значению в таком же контексте. Аппликативные функторы также иногда называют *идиомами* (англ. idioms).

В языке `Haskell` для представления аппликативных функторов используется стандартный класс типов `Applicative`, описанный в модуле `Control.Applicative`.

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

Листинг 1.10: Класс типов `Applicative`

Для всякого типа, который является аппликативным функтором, должны выполняться законы, представленные в листинге 1.11.

```
pure id <*> v = v

pure (.) <*> u <*> v <*> w = u <*> (v <*> w)

pure f <*> pure x = pure (f x)

u <*> pure y = pure ($ y) <*> u
```

Листинг 1.11: Законы аппликативного функтора

Любая монада является аппликативным функтором, но не каждый аппликативный функтор является монадой. То есть, интерфей-

```haskell
action :: IO String
action = do
  a <- getLine
  b <- getLine
  return $ a ++ b

action :: IO String
action = (++) <$> getLine <*> getLine
```

Листинг 1.12: do-нотация и аппликативный стиль

су класса типов `Applicative` удовлетворяет больше типов, чем интерфейсу класса типов **Monad**.

С понятием аппликативного функтора связан особый синтаксис описания вычислений с эффектами на `Haskell`, называемый *аппликативным стилем*. Этот стиль имеет преимущество относительно **do**-нотации для монад — компилятор имеет возможность генерировать для него более оптимальный код. Многие монадические функции могут быть переписаны в аппликативном стиле. В листинге 1.12 представлена короткая монадическая функция и эквивалентная ей аппликативная, задача которых прочитать из потока стандартного ввода две строки выполнить их конкатенацию и вернуть результат.

1.1.6. Монада

Вершиной иерархии типов, описывающих вычисления с побочными эффектами, являются монады. Возникнув как средство для введения в чистый функциональный язык возможности выполнять операции ввода-вывода, монады были обобщены и на другие вычислительные эффекты, для которых есть необходимость в построении композиции функций, и теперь являются наиболее известной абстракцией такого сорта.

Для представления монад в языке `Haskell` используется класс типов **Monad**. Кроме того, существует три закона монады, которые должны выполняться для каждого типа, имеющего экземпляр класса ти-

пов **Monad**. Эти законы отражают тот факт, что монада является моноидом в категории эндофункторов. Важно знать, что система типов не гарантирует выполнения монадических законов, ответственность за них целиком лежит на программисте, который разрабатывает экземпляр **Monad** для некоторого типа.

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

Листинг 1.13: Класс типов **Monad**

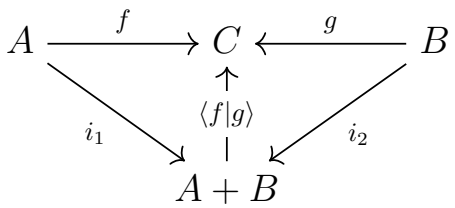
```
return a >>= k = k a
m >>= return = m
m >>= (\x -> k x >>= h) = (m >>= k) >>= h
```

Листинг 1.14: Законы монады

Монада естественным образом возникает при построении функциональных парсеров, которые описываются в третьей главе.

1.1.7. Копроизведение

Копроизведением объектов A и B в категории называется объект $A + B$ и две стрелки $i_1 : A \to A + B$ и $i_2 : B \to A + B$ такие, что для всех стрелок $f : A \to C$ и $g : B \to C$ существует единственная стрелка $\langle f|g\rangle : A + B \to C$ для которой следующая диаграмма коммутативна [6].



Стрелки i_1 и i_2 называются каноническими вложениями.

В случае категории множеств копроизведением является дизъюнктивное или размеченное объединение.

Понятие копроизведения, в случае категории типов языка `Haskell`, потребуется в главе 2, для описания расширяемых эффектов.

1.1.8. Моноиды в функциональном программировании

Моноидом является множество M с заданной на нём бинарной ассоциативной операцией $*$, и в котором существует такой элемент e, что $\forall x \in M\ e * x = x * e = x$. Элемент e называется единицей.

В языке `Haskell` для представления моноидов существует специальный класс типов.

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a
```

Листинг 1.15: Моноид в `Haskell`

Строки с операцией конкатенации и пустой строкой в качестве единицы являются моноидом. С помощью класса типов `Monoid` возможна абстрактная работа с любыми строковыми типами. Однако существуют ситуации, в которых интерфейса класса типов `Monoid` оказывается недостаточно, возникает необходимость в функции, отделяющей префикс, аналогичной функции **`head`** для списков. Для таких целей создана библиотека `monoid-subclasses` [4], предоставляющая классы типов, являющиеся моноидами специального вида, которые допускают необходимые дополнительные операции. В случае парсеров такой дополнительной операцией является аналог операции разделения списка на голову и хвост.

```
splitCharacterPrefix :: t -> Maybe (Char, t)

uncons :: [a] -> Maybe (a, [a])
```

Листинг 1.16: Отделение атомарного префикса и взятие головы списка

1.2. Существующие библиотеки функциональных парсеров

Популярной библиотекой монадических парсеров является библиотека `Parsec` [7]. Это библиотека промышленного уровня, используемая во многих проектах, например в универсальном конвертере документов `Pandoc` [8]. Преимуществами `Parsec` являются его гибкость и подробность сообщений об ошибках.

Другой популярной библиотекой является `attoparsec` [9]. При разработке `attoparsec` акцент был сделан на скорость, поэтому было принято решение ограничиться только одним типом `ByteString` и пожертвовать информативностью сообщений об ошибках. Основным предназначением `attoparsec` является анализ сетевых протоколов.

Недостатками обеих библиотек является недостаточная абстрактность в плане типов данных для представления входного потока. Применение классов типов, представленных в предыдущем подразделе, позволит получить абстрактный код способный работать с любыми строковыми типами.

Архитектура обеих библиотек использует концепцию трансформеров монад для представления вычислений с несколькими побочными эффектами. Необходимо исследовать другие концепции многоэффектных вычислений в приложении к монадическим парсерам.

ГЛАВА 2

КОМБИНИРОВАНИЕ ВЫЧИСЛИТЕЛЬНЫХ ЭФФЕКТОВ

Монады являются важным средством для построения вычислений с побочными эффектами. Каждая из стандартных монад специализирована для исполнения одной конкретной роли. В реальных программах часто требуется создавать функции, которые производят несколько разных побочных эффектов одновременно.

Решением проблемы было бы описать новый тип, имеющий свойства необходимых монад, однако этот подход нерационален и требует массированного дублирования кода. Можно также описывать функции, содержащие вложенные блоки **do**, которые эксплуатируют разные монады, подобно функции из листинга 2.1, но это подход производит сложный для понимания и поддержки код.

```
main = do -- IO
  n <- (readLn :: IO Int)
  return $ (flip evalState) (0,1) $ do -- State
    forM_ [2..n] $ \i -> do
      (x,y) <- get
      put $ (y, x + y)
    snd `fmap` get >>= return
```

Листинг 2.1: Иллюстрация использования вложенных монад: вычисление числа Фибоначчи в императивном стиле по полученному от пользователя номеру

Более рациональным решением является разработка средства комбинирования стандартных монад для порождения монад с несколькими свойствами.

В этой главе будут рассмотрены два альтернативных способа для построения вычислений с несколькими побочными эффектами: трансформеры монад и расширяемые эффекты.

2.1. Трансформеры монад

В статье [2] описаны объекты, трансформеры монад, которые можно использовать в качестве блоков для построения типов, описывающих вычисления с побочными эффектами. Каждый из трансформеров монад позволяет добавить некоторый вычислительный эффект к внутренней монаде, при этом для результирующего типа также возможно построение экземпляра класса типов **Monad**.

На данный момент трансформеры монад являются распространённым способом построения вычислений с несколькими побочными эффектами. Однако эта техника имеет недостатки: проблема невозможности автоматического *подъёма* (англ. `lift`) при наличии в стеке двух эффектов одного рода, связанная с первой проблема статически определённого порядка эффектов в стеке, а также невозможность скомбинировать две произвольные монады.

В листинге 2.2 приведён пример монадической функции, которая должна иметь два различных конфигурационных параметра. Если не произвести явный подъём, то компиляция этой функции завершится с ошибкой проверки типов вида 2.4: компилятор не в состоянии самостоятельно вывести нужный тип для функции `ask`.

Порядок монад в стеке определён статически и закодирован в типе функции, в листинге 2.3 представлена та же функция, эквивалентная по смыслу, но имеющая другой тип. Значительной деталью

```
adder :: ReaderT String (Reader Int) Int
adder = do
  str <- ask
  num <- lift ask
  return $ num + read str

runnerForAdder = runReader (runReaderT adder "2") 3
```

Листинг 2.2: Необходимость явного подъёма во внутреннюю монаду

```
adder :: ReaderT Int (Reader String) Int
adder = do
  str <- lift ask
  num <- ask
  return $ num + read str

runnerForAdder = runReader (runReaderT adder 3) "2"
```

Листинг 2.3: Функция из листинга 2.2 с другим порядком монад в стеке

также является тот факт, что в реализации этой функции `lift` применяется к другому вызову функции `ask`.

Функция `runnerForAdder` из листингов 2.2 и 2.3, запускающая вычисление, полностью определяется типом вычисления, она производит развёртку стека монад. В следующем подразделе будет рассмотрено средство комбинирования вычислительных эффектов, при использовании которого порядок на наборе эффектов устанавливается именно функцией, запускающей вычисление, а сам набор является неупорядоченным.

Следует сделать замечание относительно предыдущего примера: разнородная конфигурационная информация могла быть объединена в алгебраический тип данных, и тогда нужда в использовании двух эффектов `Reader` отпала бы, однако такое решение может не сработать для других эффектов, поэтому необходимо искать универсальный способ.

Ведутся активные поиски способов комбинирования монад, которые были бы лишены вышеперечисленных недостатков. Одним из развивающихся направлений являются расширяемые эффекты

```
No instance for (MonadReader Int (ReaderT String (Reader Int)))
      arising from a use of ask
```

Листинг 2.4: Ошибка типов при отсутствии явного подъёма

(`Extensible Effects`), которе будут обсуждаться в следующем разделе.

2.2. Расширяемые эффекты

Этот подраздел посвящён альтернативному трансформерам монад способу комбинирования вычислительных эффектов, представленному в статье [3].

2.2.1. Краткое описание библиотеки Extensible Effects

Суть подхода заключается в аналогии между вычислительными эффектами и клиент-серверным взаимодействием. Код, который собирается породить некоторый побочный эффект: совершить ввод-вывод, бросить исключение и тому подобное должен отправить *запрос* на обработку этого эффекта особой глобальной сущности — менеджеру эффектов. Запрос описывает действие, которое необходимо произвести, а также функцию-продолжение (англ. *continuation*) для возобновления работы после обработки запроса.

В ранних разработках, относящихся к такому подходу, менеджер запросов не был частью программы пользователя, а являлся отдельной сущностью, подобно ядру операционной системы, или обработчику **`IO`**-действий в `Haskell`. Этот глобальный внешний авторитет имел контроль над всеми ресурсами (файлами, памятью и другими): он обрабатывал запрос и принимал решение: исполнить его и продолжить выполнение запросившего кода, либо остановить вычисление и вернуть результат. При таком подходе нет никакой необходимости в указании явного порядка при комбинировании эффектов, однако

недостатком является негибкость внешнего интерпретатора эффектов, кроме того, эффекты никак не отражаются в типах.

Разработчики библиотеки `Extensible Effects` модифицировали концепцию:

- Глобальный обработчик запросов был заменён на средство, которое является частью пользовательской программы, некий аналог обработчиков исключений: теперь вместо единого менеджера для всех эффектов существуют локальные обработчики для каждого типа эффектов, такой подход называется *алгебраическими обработчиками* [10]. Каждый такой обработчик является менеджером для соответствующей клиентской части программы, а также и клиентом сам по себе: он пересылает запросы, которые не может обработать, менеджеру верхнего уровня.

- Разработана выразительная система типов-эффектов, которая отслеживает какие эффекты активны в данном вычислении. Эта система поддерживает особую структуру данных: *открытое объединение* (`Open Union`, индексированное типами копроизведение функторов), содержащее неупорядоченный набор вычислительных эффектов. Действие каждого обработчика отражается в типе: происходит удаление из набора эффекта, который был обработан. Таким образом система типов может отследить, все ли эффекты обработаны.

- Синтаксис для работы с эффектами построен по аналогии с синтаксисом для трансфомеров монад. Код, написанный с использованием трансформеров монад, может быть переведён на расширяемые эффекты с минимальными синтаксическими изменениями.

Есть два способа обозначить принадлежность эффекта `m` открытому объединению `r`. С помощью типового ограничения `Member m r`, которое означает, что вычисления имеет *по крайней мере* один по-

```
adder :: ( Member (Reader Int) r
         , Member (Reader String) r
         ) => Eff r Int
adder = do
  num <- ask
  str <- ask
  return $ num + read str

runAdder = run $ runReader (runReader adder "2") (1 :: Int)
```

Листинг 2.5: Пример функции с двумя средами конфигурации.

бочный эффект `m`. Иначе, можно явно указать шаблон `m :> r'` разложения набора `r` на эффект `m` и оставшийся набор `r'`, что аналогично теоретико-множественному обозначению $\{m\} \cup r'$. Тип **`Void`** играет роль $\varnothing$, то есть вычисление с типом `Eff` **`Void`** `a` является чистым, а вычисления с типом `Eff (Reader` **`Int`** `:> Reader` **`Bool`** `:>` **`Void`**`)` может иметь побочный эффект обращения к двум средам конфигурационной информации.

Авторы также указывают, что ограничение `Member (Reader` **`Int`**`)r` выглядит похоже на ограничение принадлежности классу типов `MonadReader`, более того, возможно объявить экземпляр этого класса для монады `Eff r`. Однако в этом нет необходимости, так как тип `Eff r` является более выразительным: в листинге 2.5 приводится пример работы с функцией, имеющей два эффекта `Reader`. Эта функция демонстрирует одно из преимуществ расширяемых эффектов перед трансформерами монад: пропадает необходимость явного вызова функции `lift`, которую можно наблюдать в листингах 2.2 и 2.3. Каждое вхождение функции `ask` обращается к своей собственной среде, определяемой типом.

Другое важное отличие расширяемых эффектов от трансформеров монад — неупорядоченность набора эффектов до запуска вычисления — никак не проявило себя в примере из листинга 2.5, потому что были использованы одинаковые эффекты. В листинге 2.6 рассмотрен синтетический (для краткости) пример, демонстрирующий динамическую установку порядка на множестве эффектов. Функция

```
countdown :: ( Member Fail r
             , Member (State Int) r
             ) => Eff r ()
countdown = do
  state <- get
  if state == (0 :: Int)
  then die
  else put (state - 1) >> countdown

runCountdown1 n = run $ runFail $ runState (n :: Int) $ countdown

runCountdown2 n = run $ runState (n :: Int) $ runFail $ countdown
```

Листинг 2.6: Порядок на множестве эффектов определяется динамически

из примера считает до нуля и завершается с исключением. Если запускать вычисление в порядке, представленном в `runCountdown1`, то результатом будет **`Nothing`**, сообщающий об исключении. При запуске в порядке `runCountdown2`, результатом будет пара `(0,`**`Nothing`**`)` из последнего состояния и сообщения об исключении.

2.2.2. Пример практического использования

В этом подразделе будет рассмотрена системная утилита, написанная с использованием библиотеки `Extensible Effects`. Задача этой утилиты — отслеживание изменений содержимого некоторой директории и выполнение заданных сценариев командной оболочки в качестве реакции на эти изменения. Git-репозиторий с исходным кодом на языке `Haskell` доступен по адресу [11].

На этапе проектирования приложения необходимо определить, какие вычислительные эффекты будут им порождаться. Рассматриваемая утилита будет работать с файловой системой, значит потребуется эффект **`IO`**, иметь внутреннее состояние — необходим эффект `State`, а также будет иметь конфигурационную информацию — потребуется эффект `Reader`.

```
loop :: ( Member (Reader AppConfig) r
        , Member (State AppState)   r
        , SetMember Lift (Lift IO)  r
        ) => Eff r ()
loop = do
  cfg <- ask
  currentFilesList <- lift $ getFilesInfo $ directory cfg
  handleCreate currentFilesList
  lift $ threadDelay $ refreshRate cfg
  curTime <- lift getCurrentTime
  put $ (AppState currentFilesList curTime)
  loop
```

Листинг 2.7: Главный цикл приложения

```
runApp action cfg initState =
  runLift . runState initState . runReader action $ cfg
```

Листинг 2.8: Обработка эффектов и запуск приложения

В листинге 2.7 представлена упрощённая функция главного цикла программы, типовая аннотация отражает наличие в результирующем наборе эффектов `r` трёх составляющих: `Reader`, `State` и **`IO`**.

Как уже говорилось ранее, порядок на наборе эффектов устанавливается в момент запуска вычисления, для чего служит функция из листинга 2.8.

ГЛАВА 3

МОНАДИЧЕСКИЕ ПАРСЕРЫ

В девяностые годы двадцатого века были опубликованы результаты исследований, показывающих монадичекую природу функциональных парсеров [12]. Эта их особенность приносит практические плоды: использование монадического комбинатора связывания позволяет удобно строить композиции парсеров.

Дальнейшее развитие монадического подхода к построению парсеров делает возможным выражение цельной монады для парсера в терминах более фундаментальных монад, что обеспечивает большую модульность архитектуры и позволяет варьировать семантику парсера за счёт изменения внутренней монады стека: создавать недетерминированные парсеры, парсеры, сигнализирующие о некорректности исходного текста, парсеры, генерирующие сообщения об ошибках.

3.1. Классический подход к построению монады Parser

Большая часть содержимого этого раздела является изложением первой части статьи [1] и приводятся здесь для удобства последующего изложения.

Функциональный стиль построения парсеров заключается в том, что парсер представляется функцией, которая принимает на

вход строку символов и строит по ней некое абстрактное синтаксическое дерево. Удобно считать, что не каждый парсер полностью потребляет входную строку, это даёт возможность построения результирующего парсера по частям, из примитивных. Также необходимо иметь средства для обработки некорректного входа. Для этого существуют разные способы: можно сообщать об ошибке разбора, либо заменять неудачу списком успехов [13]. Высказанные пожелания о характеристиках парсера могут быть отражены следующим типом языка Haskell:

```
type Parser a = String -> [(a,String)]
```

Листинг 3.1: Тип Parser

Мотивацией для приложения монад к построению парсеров является неудобность композиции немонадических парсеров, приводящая к появлению вложенных кортежей:

```
seq :: Parser a -> Parser b -> Parser (a,b)
p `seq` q = \inp -> [((v,w),inp'') | (v,inp')  <- p inp
                                   , (w,inp'') <- q inp']
```

Листинг 3.2: Комбинатор для построения композиции парсеров

В листинге 3.3 приведён экземпляр класса типов Monad для типа Parser:

Операция `>>=`, известная как монадическое связывание (англ. `bind`), является удобным средством для построения композиции парсеров, её использование позволяет избежать неконтролируемого возникновения вложенных кортежей, за счёт прямой передачи выхода первого парсера на вход второго.

Приведём простейшие примеры парсеров.

В листинге 3.4 представлен базовый парсер `item`, его задача заключается в отделении одного символа от входной строки. Именно его реализация требует активной эксплуатации средств, определяемых конкретным типом, представляющим парсер. Это значит, что реализация парсера `item` для цельной монады `Parser` отличается от той,

```
instance Monad Parser where
  return t = Parser $ \s -> Just (t, s)
  m >>= k  = Parser $ \s -> do
                              (u, v) <- parse m s
                              (x, y) <- parse (k u) v
                              return (x, y)
```

Листинг 3.3: Экземпляр Monad для Parser

```
item :: Parser Char
item =  Parser f
        where f []     = []
              f (x:xs) = [(x,xs)]
```

Листинг 3.4: Парсер `item` в случае цельной монады

что будет уместна в случае использования средств комбинирования монад.

3.2. Монада Parser как комбинация более простых монад

Рассмотренная в предыдущем подразделе монада `Parser` не является единственным примером монады. В статье [12] Филип Вадлер (Philip Wadler) приводит примеры типов, имеющих монадическую структуру. В статье [1] описан способ представить тип из листинга 3.3 в виде комбинации монад `State` и `[]`, для этого применяется механизм трансформеров монад.

3.2.1. Подход с трансформерами монад

Как уже говорилось ранее, монада `Parser` из предыдущего подраздела может быть представлена в виде комбинации двух более простых монад (см. листинг 3.5)

Благодаря трансформерам монад к этому типу могут быть добавлены и другие вычислительные эффекты, такие как, например,

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

Листинг 3.5: Тип Parser как комбинация монад State и []

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

item :: Parser t Char
item = do
  state <- get
  case state of
    []     -> []
    (x:xs) -> put xs >> return c
```

Листинг 3.6: Простой тип Parser

конфигурационная информация и журнализация. Внутренняя монада может быть изменена: вместо списковой монады, описывающей недетерминированный парсер, можно использовать монаду `Maybe` или `Either a` для получения парсера, выдающего сообщение об ошибке в случае неудачного разбора. Таким образом, благодаря трансформерам монад становится доступна такая гибкости управления типом парсера, достичь которой было бы тяжело при явном описании монады `Parser`.

Монадический код имеет высокий уровень абстракции. Это приносит ценные практические плоды: код парсера, написанных для одного стека монад с высокой вероятностью может быть без изменений перенесён на другой стек. Принципиальных изменений требует только самый базовый парсер `item`, который напрямую работает со структурой стека монад. В листингах 3.6 и 3.7 представлены примеры базового парсера для двух стеков монад: простого и более сложного, используемого в библиотеке для синтаксического разбора `Markdown`, которая будет описана в следующем разделе.

В парсере `item` из листинга 3.7 является обобщённым по типу входных данных: использование класса типов `TextualMonoid` из библиотеки `monoid-subclasses` позволяет использовать этот парсер для любых строковых типов. В случе неудачного разбора генерируется

```
newtype Parser t a = Parser (StateT (ParserState t)
                            (Either (ErrorReport t)) a)

item :: TM.TextualMonoid t => Parser t Char
item = do
  state  <- get
  let s = TM.splitCharacterPrefix . remainder $ state
  case s of
    Nothing -> throwError (EmptyRemainder "item",state)
    Just (c,rest) -> do
      let (c,rest) = fromJust s
      put (ParserState { position = updatePos (position state) c,
                         remainder = rest })
      return c
```

Листинг 3.7: Парсер с поддержкой моноидального входного потока и отслеживанием ошибок

ошибка, сообщающая о пустой входной строке и содержащая последнее корректное состояние парсера, которое включает в себя позицию и оставшуюся входную последовательность.

3.2.2. Подход с расширяемыми эффектами

Библиотека `Extensible Effects` предоставляют по крайней мере два способа комбинирования вычислительных эффектов для описания монадических парсеров: явное статическое описание набора эффектов и указание эффектов в типе каждой функции парсера.

В первом подходе чувствуется влияние траснформеров монад, и он не позволяет воспользоваться всеми преимуществами расширяемых эффектов. В листинге 3.8 представлен тип для парсера со статическим набором эффектов и функция для обработки эффектов.

```
type Parser a = Eff (Fail :> State String :> Void)

parse p inp = run . runFail . runState inp $ p
```

Листинг 3.8: Тип Parser как статически заданный набор эффектов

Второй способ не подразумевает описание какого-либо специального типа для парсера. Каждая функция-парсер имеет свой соб-

ственный набор эффектов, ограниченный снизу набором эффектов базового парсера `item` (листинг 3.9).

```
item :: ( Member Fail r
        , Member (State String) r
        ) => Eff r Char
item = do s <- get
          case s of
            [] -> die
            (x:xs) -> put xs >> return x
```

Листинг 3.9: Возможный вариант парсера `item` и его эффекты

На основе парсера `item` строятся другие парсеры. В листинге 3.10 представлен парсер для символа, удовлетворяющего предикату.

```
sat :: ( Member Fail r
       , Member (State String) r
       ) => (Char -> Bool) -> Eff r Char
sat p = do x <- item
           if p x then return x else die
```

Листинг 3.10: Распознавание символа по предикату

Набор эффектов парсера более высокого уровня не обязан ограничиваться эффектами парсера `item`, в листинге 3.11 представлен парсер для букв, определённый с помощью эффекта недетерминированного выбора, парсеры `lower` и `upper` определены через парсер `sat`. Эффект `Choose` предоставляет функции **`mzero`**' и **`mplus`**', аналогичные интерфейсу класса типов **`MonadPlus`**. Стандартном поведением для этих функций является недетерминированный выбор: результаты применения парсеров склеиваются в список.

Все заявленные эффекты должны быть обработаны. Библиотека парсеров, основанная на расширяемых эффектах, должна предоставлять соответствующий набор функция для запуска вычислений. В листинге 3.12 представлены две функции, позволяющие использовать описанные выше парсеры.

Построение библиотек монадических парсеров с использованием расширяемых эффектов имеет несколько иную специфику, чем

```
letter :: ( Member Fail r
          , Member (State String) r
          , Member (Choose) r
          ) => Eff r Char
letter = lower `mplus` upper
```

Листинг 3.11: Парсер с расширенным относительно `item` набором эффектов

```
parse p inp = run . runFail . runState inp $ p

parseWithChoose p inp =
  run . runChoice . runFail . runState inp $ p
```

Листинг 3.12: Функции для запуска парсеров с разными наборами эффектов

с использованием трансформеров монад. С одной стороны, расширяемые эффекты предоставляют большую гибкость: каждый парсер имеет свой собственный набор эффектов. С другой стороны, механизм трансформеров монад имеет удобную поддержку со стороны компилятора `GHC`, которая обеспечивает автоматическое порождение экземпляров нужных классов типов, таких как, например, **`MonadPlus`**, `Applicative` и `Alternative`, что избавляет разработчика от необходимости описывать некоторые вспомогательные функции. Таким образом, концептуально расширяемые эффекты являются более интересным средством, но трансформеры монад удобнее для прикладного программирования.

Экспериментальная версия библиотеки комбинаторов парсеров, основанной на расширяемых эффектах доступна в репозитории [14]

ГЛАВА 4

ФУНКЦИОНАЛЬНЫЙ ПАРСЕР ЯЗЫКА MARKDOWN

Язык Markdown — легковесный язык разметки, применяется для быстрой вёрстки небольших документов. Полезен в случаях, когда нет необходимости привлекать такие тяжеловесные форматы как HTML и LaTeX. Популярен в интернете, к примеру, на известном хостинге программного кода GitHub.

4.1. Синтаксис Markdown

В отличие от HTML или XML, Markdown не имеет стандарта. Существует неформальное, но подробное описание базового синтаксиса [15], а также нескольких расширенных версий, в числе которых так называемый GitHub Flavored Markdown.

В реализуемом парсере рассматривается подмножество базового синтаксиса, включающее в себя заголовки, параграфы, неупорядоченные списки и блочные цитаты. Дополнительной возможностью являются вставки математических формул в формате LaTeX.

4.2. Реализация парсера

Язык программирования Haskell известен своей развитой системой типов. Аппарат алгебраических типов данных предоставляет удобные средства для представления абстрактного синтаксического

```
type Document = [Block]

data Block = Blank
           | Header (Int,Line)
           | Paragraph [Line]
           | UnorderedList [Line]
           | BlockQuote [Line]
  deriving (Show,Eq)

data Line = Empty | NonEmpty [Inline]
  deriving (Show,Eq)

data Inline = Plain String
            | Bold String
            | Italic String
            | Monospace String
  deriving (Show,Eq)
```

Листинг 4.1: Типы для представления абстрактного синтаксического дерева рассматриваемого подмножества `Markdown`

дерева. Любой `Markdown`-документ представляет собой список блоков. Блок представляется типом-суммой, конструкторы которого отражают рассматриваемое подмножество `Markdown`. В листинге 4.1 представлены типы, описывающие рассматриваемую грамматику.

Блоком является либо пустой блок, либо заголовок, либо параграф, либо неупорядоченный список, либо блочная цитата. Большинство блоков имеют в своём составе список строк. Строка распадается на строчные элементы, дифференцируемые по стилю начертания: простые, полужирные, курсивом и моноширинные. В листинге 4.3 представлены парсеры для распознавания строк, парсеры `bold`, `italic` и `plain` аналогичны парсеру `monospace`, поэтому опущены для краткости.

При реализации парсеров активно применяются комбинаторы из библиотек, принцип построения которых был описан в главе 3. Стоит подробнее остановится на трёх из них, их типовые аннотации приведены в листинге 4.2:

```
many :: Parser t a -> Parser t [a]

sepby :: Parser t a -> Parser t b -> Parser t [a]

bracket :: Parser t a -> Parser t b -> Parser t c -> Parser t b
```

Листинг 4.2: Типовые аннотации основных комбинаторов парсеров

```
line :: TM.TextualMonoid t => Parser t Line
line = emptyLine `mplus` nonEmptyLine

emptyLine :: TM.TextualMonoid t => Parser t Line
emptyLine = many (sat wspaceOrTab) >> char '\n' >> return Empty

nonEmptyLine :: TM.TextualMonoid t => Parser t Line
nonEmptyLine = do
  many (sat wspaceOrTab)
  l <- sepby1 (bold <|> italic <|>
               plain <|> monospace) (many (char ' '))
  many (sat wspaceOrTab)
  char '\n'
  return . NonEmpty $ l

monospace :: TM.TextualMonoid t => Parser t Inline
monospace = do
  txt <- bracket (char '`') sentence (char '`')
  p   <- many punctuation
  return . Monospace $ txt ++ p
```

Листинг 4.3: Распознавание строк из строчных элементов.

1. Комбинатор `many` распознаёт список токенов, удовлетворяющих его парсеру-параметру.

2. Комбинатор `sepby` распознаёт последовательность токенов, удовлетворяющих его первому параметру и разделённых токенами, удовлетворяющими его второму параметру.

3. Комбинатор **`bracket`** распознаёт токены, удовлетворяющие его третьему параметру и заключенные между токенами, удовлетворяющими первому и третьему параметру соответственно.

```
header :: TM.TextualMonoid t => Parser t Block
header = do
  hashes <- token (some (char '#'))
  text <- nonEmptyLine
  return $ Header (length hashes,text)
```

Листинг 4.4: Парсер для заголовка

```
unorderedList :: TM.TextualMonoid t => Parser t Block
unorderedList = do
  items <- some (token bullet >> line)
  return . UnorderedList $ items
  where
    bullet :: TM.TextualMonoid t => Parser t Char
    bullet = char '*' <|> char '+' <|> char '-' >>= return
```

Листинг 4.5: Парсер для неупорядоченного списка

Получив возможность распознавать строки и строчные элементы, можно приступить к реализации парсеров для блоков. В листинге 4.4 представлен парсер для заголовка, а в листинге 4.5 — для неупорядоченного списка, парсеры для остальных блоков описываются схожим образом. Здесь следует сделать замечание, что текущая версия парсера не поддерживает вложенные списки, оформляемые в Markdown с помощью отступов.

Markdown применяется также для электронного конспектирования и оформления заданий. Полезным расширением грамматики Markdown являются вставки математических формул в формате LaTeX. Требуется распознать LaTeX-блок и оставить его без изменений, чтобы при последующей генерации кода по абстрактному синтаксическому дереву можно было отобразить его соответствующим образом. Для этого служит парсер из листинга 4.6.

```
blockMath :: TM.TextualMonoid t => Parser t Block
blockMath =
  (bracket (string "$$") (some (sat (/= '$'))) (string "$$")) >>=
  return . Paragraph . (: []) . NonEmpty . (: []) . Plain .
    (\x -> "$$" ++ x ++ "$$")
```

Листинг 4.6: Парсер LaTeX-блоков

В листинге 4.7 представлен парсер самого верхнего уровня, служащий для распознавания Markdown-документа как списка блоков.

```
doc :: TM.TextualMonoid t => Parser t Document
doc = many block
  where block = blank <|> header <|> paragraph <|>
                unorderdList <|> blockquote <|> blockMath
```

Листинг 4.7: Парсер, распознающий Markdown-документ

4.3. Генерация HTML-кода

Имея абстрактное синтаксическое дерево Markdown-документа можно сгенерировать по нему исходный текст на любом требуемом языке разметки. В качестве целевого был выбран язык разметри гипертекста HTML, для рендеринга LaTeX-вставок используется JavaScript -библиотека MathJax [16]. Возможно построения генераторов и для других языков разметки.

Генерация кода происходит от общего к частному. Функция serialize генерирует код по списку блоков и конкатенирует результат. Генерация каждого блока производится функцией genBlock, которая соответствующим образом обрабатывает все поддерживаемые виды блоков и генерирует нужный код. Строки и строчные элементы генерируются функциями genLine и genInline соответственно.

В листинге 4.8 представлен сокращённый кодогенератор, опущена, для краткости, обработка некоторых блоков и строчных элементов, которая происходит аналогично представленным. Этот код служит демонстрацией выразительности, которая предоставляется сопоставлением с образцом и алгебраическими типами данных.

Полный исходный код парсера Markdown, кодогенератора, а также вспомогательной библиотеки комбинаторов парсеров доступен в репозитории [17].

```
serialize :: Document -> String
serialize = concatMap genBlock

genBlock :: Block -> String
genBlock Blank = "\n"
genBlock (Header h) =
  "<h" ++ s ++ ">" ++ genLine (snd h) ++ "</h" ++ s ++ ">" ++ "\n
    "
    where s = show (fst h)
genBlock (UnorderedList l) =
  "<ul>" ++ concatMap ((++ "\n") . genOrderedListItem) l ++ "</ul
    >" ++ "\n"

genLine :: Line -> String
genLine Empty         = ""
genLine (NonEmpty []) = genLine Empty ++ "\n"
genLine (NonEmpty l)  = concatMap ((++ " ") . genInline) l

genOrderedListItem :: Line -> String
genOrderedListItem l = "<li>" ++ genLine l ++ "</li>"

genInline :: Inline -> String
genInline (Plain s) = s
genInline (Monospace s) = "<code>" ++ s ++ "</code>"
```

Листинг 4.8: Генерация HTML

СПИСОК ЛИТЕРАТУРЫ

1. *Hutton G.*, *Meijer E.* Monadic Parser Combinators // Технический отчёт NOTTCS-TR-96-4. — 1996.
2. *Sheng Liang Paul Hudak M. J.* Monad Transformers and Modular Interpreters // 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, CA. — 1995.
3. *Oleg Kiselyov Amr Sabry C. S.* Extensible Effects: An Alternative to Monad Transformers // Haskell Symposium 2013, Boston, MA, USA, 23–24 September 2013. — 2013.
4. *Blaževic M.* Adding Structure to Monoids // Haskell Symposium 2013, Boston, MA, USA, 23–24 September 2013. — 2013.
5. Entscheidungsproblem, статья из английской Википедии. — URL: `https://en.wikipedia.org/wiki/Entscheidungsproblem` (дата обр. 01.01.2015).
6. *Michael Barr C. W.* Category Theory For Computing Science. — M. : Reprints in Theory, Applications of Categories, No. 22, 2012. — URL: `http://www.tac.mta.ca/tac/reprints/articles/22/tr22.pdf`.
7. Parsec, универсальаня библиотека монадических комбинаторов парсеров. — URL: `https://github.com/aslatter/parsec`.

8. Pandoc, универсальный конвертер документов. — URL: http://pandoc.org/.
9. Attoparsec, быстрая Haskell-библиотека для разбора ByteString. — URL: https://github.com/bos/attoparsec.
10. *Andrej Bauer M. P.* Programming with Algebraic Effects and Handlers // arXiv:1203.1539 [cs.PL]. — 2012.
11. Репозиторий с кодом утилиты File Trigger. — URL: https://github.com/geo2a/file-trigger (дата обр. 23.05.2015).
12. *Wadler P.* Monads for functional programming // Advanced Functional Programming, Bastad Spring School. — 1995.
13. *Wadler P.* How To Replace a Failure by a List of Successes // Конференция по функциональному программированию и архитектуре компьютеров. Springer–Verlag. — 1985.
14. Репозиторий с кодом библиотеки пасреров, основанной на расширяемых эффектах. — URL: https://github.com/geo2a/ext-effects-parsers (дата обр. 23.05.2015).
15. Синтаксис языка Markdown. — URL: http://daringfireball.net/projects/markdown/syntax (дата обр. 29.05.2015).
16. MathJax, библиотека для отображения LaTeX в веб-браузерах. — URL: https://www.mathjax.org/ (дата обр. 29.05.2015).
17. Репозиторий с кодом библиотеки пасреров и парсера Markdown. — URL: https://github.com/geo2a/markdown_monparsing/tree/experimental (дата обр. 23.05.2015).