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

ПАРСИНГ ВЛОЖЕННЫХ КОНСТРУКЦИЙ В MARKDOWN НА ЯЗЫКЕ HASKELL

Курсовая работа

Студентки 3 курса
М. В. Втюриной

Научный руководитель:
ассистент кафедры информатики и вычислительного эксперимента
Мехмата ЮФУ А. М. Пеленицын

_____ _____

оценка (рейтинг) подпись руководителя

Ростов-на-Дону
2016

Содержание

Введение

В функциональном программировании популярный подход к построению парсеров рекурсивного спуска - это моделирование парсеров с помошью комбинаторов. Комбинаторы обеспечивают быстрый и простой метод построения функциональных парсеров. Большую роль в обработке вложенных конструкций играет Goffer offside rule. Это правило позволяет группировать определения в программе с использованием отступов. (Обычно осуществляется лексером, который вставляет дополнительные маркеры (относительно отсупа) в свой выходной поток)

1. Основные понятия

Прежде чем перейти к реализации программы на языке Haskell, необходимо ознакомиться с некоторыми основными понятиями функционального программирования.

1.1. Монады и парсеры

Функторы.

```
fmap :: (a -> b) -> fa -> fb
```

Аппликативные функторы.

```
(<*>) :: (Applicative f) => f(a -> b) -> f a -> f b
```

Монады.

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

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, суффикс строки), где пустой список означает неудачу. Различные парсеры могут возвращать различные виды деревьев.

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

Листинг 1.1. Пример парсера, обрабатывающего первый символ, если входная строка не пуста

```
item :: Parser Char
item  = \inp -> case inp of
                    []     -> []
                    (x:xs) -> [(x,xs)]
```

1.2. Токены и лексеры

Лексический анализатор или лексер — это программа или часть программы, выполняющая лексический анализ. Лексический анализатор обычно работает в две стадии: сканирование и оценка.

На первой стадии, сканировании, он реализуется в виде конечного автомата, определяемого регулярными выражениями. В нём кодируется информация о возможных последовательностях символов, которые могут встречаться в токенах .

Полученный таким образом токен содержит необработанный исходный текст (строку). Для того чтобы получить токен со значением, соответствующим типу (напр. целое или дробное число), выполняется оценка этой строки — проход по символам и вычисление значения.

Токен с типом и соответственно подготовленным значением передаётся на вход парсеру.

Разница между этими понятиями очень точная. Лексер распознает лексемы (токены). Например, если мы анализируем запись какого-то выражения, то отдельными токенами будут числа и знаки операций. На этом работа лексера заканчивается. Парсер же на основе списка токенов выполняет синтаксический анализ и далее уже строит какую-то модель на основании токенов.

2. Парсинг Markdown

Markdown - это язык форматирования, по приниципу 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.

2.1. Возможности Markdown

Выделение текста [1]:

курсив -> *курсив*

полужирный -> **полужирный**

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

(Выделение можно применять к отдельному слову, нескольким словам или части слова)

Markdown позволяет использовать заголовки в тексте. Доступны шесть уровней заголовков. (Начинаем строку с одного или больше символов #)

Списки [1]:

Можно создавать маркированные списки, начиная каждую строку звездочкой и отделяя ее от текста пробелом. Аналогично создаются и нумерованные списки: начинем каждую строку 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.

Цитирование [1]:

Указать, что часть текста является цитатой можно, начав каждую строку с угловой скобки (>). Этот символ выбран по той причине, что многие почтовые программы используют именно его для выделения цитат. Результатом цитирования будет выделение абзаца отступом справа и слева.

Web-ссылки [1]:

Есть два способа создать ссылку на какой-либо web-ресурс. Первый - поместить ссылку прямо в текст. Нужно заключить текст, который пользователь должен видеть и который будет выглядеть как ссылка в квадратных скобках, а URL страницы в круглые так, чтобы комплекты скобок были написаны слитно. Также можно добавить дополнительное описание, заключив его в кавычки и поместив после URL.

```
[link text](adress://link_here "Title")
```

Второй способ удобен для длинных ссылок. Чтобы не загромождать текст, можно вставлять их сносками. Добавляем короткий идентификатор с помощью второй пары квадратных скобок (идентификатор может быть поясняющим словом, фразой или числом).Затем, в любом месте документа указываем URL, связав его с идентификатором.

```
This [link example][ex] is in my document
[ex]: http://example.com/ "additional description"
```

Картинки [1]:

Картинки помещаются на страницу точно таким же образом, как и web-ссылки, но предваряются восклицательным знаком.

2.2. Реализация

Листинг 2.1. Типы данных [2]

```
type Document = [Block]

-- |Represents block entity
data Block = Blank
           | Header (Int,Line)
           | Paragraph [Line]
           | UnorderedList [Line]
           | BlockQuote [Line]
           | List [Blocks]
  deriving (Show,Eq)

-- |Represents line as list of inline elements (words)
data Line = Empty | NonEmpty [Inline]
  deriving (Show,Eq)

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

Листинг 2.2. Блочные элементы [2]

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

-- |Parse paragraph
paragraph :: TM.TextualMonoid t => Parser t Block
paragraph = do
  --l <- bracket emptyLine nonEmptyLine emptyLine
  l <- some nonEmptyLine
  return . Paragraph $ l

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


-- |Parse blockquote
blockquote :: TM.TextualMonoid t => Parser t Block
blockquote = do
  lines <- some (token (char '>') >> line)
  return . BlockQuote $ lines
```

Листинг 2.3. Парсинг всего документа. Возвращает список блоков [2]

```
doc :: TM.TextualMonoid t => Parser t Document
doc = do
  ls <- many block
  --a <- header
  --b <- blank
  --c <- paragraph
  return $ ls
  where
    block =
      blank <|> header <|> paragraph <|>
      unorderdList <|> blockquote <|> blockMath
```

3. Вложенные конструкции

3.1. Gofer offside rule

Обработка Gofer offside rule [3] — задача, осуществляющаяся с помощью лексера. 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 — обработка с помошью специальных комбинаторов.

Рассмотрим простую программу и её структуру.

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

{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.

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

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

Состояние парсера (строка, столбец).

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

Также парсеру нужно знать номер столбца текущего определения. Если offside rule не действует, то номер этой позиции может быть отрицательным.

Листинг 3.1. Окончательный тип парсера

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

Рассмотрим конструктор типа **ReaderM**

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

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

Листинг 3.2.

```
 class Monad m => ReaderMonad m s where
      env    :: m s
      setenv :: s -> m a -> m a
   instance Monad m => ReaderMonad (ReaderM m s) s where
-- env :: Monad m => ReaderM m s s
 env = \s -> result s
--setenv:: Monad m => s ->  ReaderM m s a -> ReaderM m s a
setenv s srm = \_ -> srm s
```

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

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

Парсер для **item** будет изменен: парсинг не удастся выполнить, если позиция рассматриваемого символа относительно определения не находится внутри.

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)

Листинг 3.3.

```
item :: Parser Char
item  = [x | (pos,x:_) <- update newstate
           , defpos    <- env
           , onside pos defpos]

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

Листинг 3.4.

```
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)
```

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 3.5.

```
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)

Листинг 3.6.

```
many1offside  :: Parser a -> Parser [a]
many1offside 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]
```

Листинг 3.7.

```
manyoffside :: Parser a -> Parser [a]
manyoffside p = many1offside p +++ [[]]
```

(**manyoffside** делает то же самое, что и **many1offside**, но еще парсит пустую последовательность)

В итоге, чтобы построить парсер для обработки Gofer offside rule, мы изменили тип некоторых парсеров, чтобы добавить информацию о позициях, модифицировали item и junk и определили два новых - manyoffside и many1offside. Все остальные переопределения будут сделаны автоматически системой типов Gofer.

В заключении можно сказать, что монадический характер парсеров приносит практическую пользу. Например, позволяет избежать беспорядочной манипуляции с вложенными кортежами, кроме того монадические конструкции делают парсеры более компактыми и легкими для чтения.

Библиография

1. Документация по Markdown. — URL: http://on.econ.msu.ru/help.php?file=markdown.htmlu.
2. Проект по парсингу Markdown. — URL: https://github.com/geo2a/markdown-monparsing.
3. *Hutton C.*, *Meijer E.* Monadic Parser Combinators. — 1996.