Информатика

Душкин Роман Викторович
Автор нескольких книг по функциональному программированию.

Генерация рекурсивных сказок

В статье описывается одна из интереснейших прикладных задач – генерация естественно-языковых текстов на примере так называемых рекурсивных сказок из числа русских народных. Приводятся программы на функциональном языке программирования Haskell, которые производят генерацию текстов сказок «Колобок», «Теремок» и «Репка» на основе списка действующих лиц.

В настоящем номере публикуется начало статьи. Окончание ждите в следующем номере.

Введение

За сотни и даже тысячи лет своего существования русский народ создал широчайший пласт фольклорного творчества – русские народные сказки. В свою очередь, они подразделяются на несколько классов – волшебные, бытовые, сказки о животных и др. Ещё одним из таких

классов являются так называемые рекурсивные сказки, которые обычно рассказываются самым маленьким детям – постоянное повторение одного и того же сюжета с постепенным наращиванием действия позволяет детям заучивать слова, размеренный ритм сказки помогает уснуть. Рекурсивные сказки тоже подразделяются на несколько видов, некоторые из которых не так интересны (например, сказка про белого бычка, которая никогда не кончается).

Более интересны сказки с «раскручивающимся» сюжетом, в которых основное действие повторяется с каждым новым действующим лицом, пока не наступает кульминация. Таковы всем известные сказки про колобка, про теремок (в некоторых вариантах про потерянную мужиком в

лесу рукавичку), про репку. И кто в детстве не пытался проникнуть в суть таких сказок, ещё не понимая принципа *рекурсивных вычислений*, когда по собственному разумению добавлял в перечень действующих лиц новых персонажей?

Исследователи народного творчества давно приметили такую особенность рекурсивных сказок, как раскручивание сюжета до кульминации. Более того, во всех сказках выделяется одна структура, один «*паттерн*». Если описывать кратко, то такой паттерн заключается в том, что на обычную структуру повествования прозаического произведения (пролог – действие – эпилог) накладывается дополнительный рекурсивный перебор действующих лиц, иногда с заглядыванием вперёд. Первое действующее лицо осуществляет «зачин», все последующие делают своё дело, а последний персонаж приводит всё действие к своей кульминации.

На основе результатов таких исследований можно попробовать написать небольшие компьютерные программы, которые, будучи сами небольшими по размеру, позволяли бы генерировать сказки на основе входных списков действующих лиц неограниченной длины. Само собой, что для таких целей в силу рекурсивности мотива сказок целесообразно применять функциональную парадигму программирования. Ниже в настоящей статье приводятся примеры генераторов сказок на языке программирования Haskell, а также делается попытка создания обобщённого генератора *в виде функции высшего порядка*.

Все предлагаемые к изучению примеры приводятся в классическом варианте. Заинтересованный читатель может самостоятельно генерировать новые варианты соответствующих сказок, произвольно меняя список действующих лиц.

1. Колобок

Колобок был испечён бабкой по наказу деда из малой горстки муки. Не дождавшись того, что дед приступит к трапезе, колобок соскочил на тропинку около избушки и покатился в лес. В лесу он постоянно встречал различных персонажей, каждому из которых пел песенку, упоминая при этом всех предыдущих действующих лиц (*рекурсия*), чем вводил их в смятение и ускользал от хищной пасти. Наконец, колобок встретил наиболее хитрого зверя (лису), который его обманул и съел (кульминация).

Как указано во введении, вся канва сказки укладывается в простую схему «пролог – действие – эпилог». Этот факт записать на языке Haskell проще всего. С этого можно и начать:

```
kolobok :: String
kolobok = prologue ++
          act ++
          epilogue
```

Естественно, что никаких таких функций **prologue**, **act** и **epilogue** ещё нет, их необходимо только создать. Первая и последняя реализуются очень просто:

```
prologue :: String
prologue = "Жили-были дед и баба. Вот однажды дед говорит бабе:\n" ++
           "- Испеки-ка мне, старая, колобок.\n" ++
           "- Да из чего ж я его испеку-то? Муки ведь нет.\n" ++
           "- А ты по амбарам помети, по сусекам поскреби, " ++
           "авось мука и наберётся.\n" ++
           "Ну баба по амбарам помела, по сусекам поскребла, " ++
           "набрала горсти две муки. " ++
           "Замесила тесто на сметане, слепила колобок, " ++
           "изжарила его в масле и поставила на окно простынуть.\n" ++
           "Колобок лежал-лежал, скучно ему стало, " ++
           "вот он и прыгнул с окна на завалинку, " ++
           "с завалинки на лавку, с лавки на дорожку " ++
           "и покатился по ней в лес.\n"

и

epilogue :: String
epilogue = ""
```

Эпилог у этой сказки не реализован, потому что когда лиса съедает колобка, ничего больше не происходит. А вот в функции **prologue** скрыта одна неприятная логическая ошибка – в ней в виде констант упоминаются такие действующие лица как «дед» и «баба», что может привести к неприятным последствиям, если иметь желание внести их в список действующих лиц. Но пока проводится исследование внутренней структуры сказки, пусть будет так – список персонажей начнётся с зайца.

Самое интересное начинается при попытке реализовать функцию **act**, которая должна перебирать перечень действующих лиц. Для её работы прежде всего необходим как раз этот перечень, а для его создания в виде списка необходим тип, описывающий одно действующее лицо. Такой тип проще всего определить в виде структуры с именованными полями, которых будет четыре – три для наименования персонажа в именительном, родительном и дательном падежах (все три формы встречаются в тексте сказки), а также для указания пола персонажа, чтобы правильно генерировать форму глагола в прошедшем времени. Итак, тип **Actor** выглядит следующим образом:

```
data Actor
  = Actor
    {
      name :: String,
```

```
      name_g :: String,
      name_a :: String,
      gender :: Gender
    }
 deriving  Eq
```

Ну и вспомогательный тип **Gender** является простым перечислением:

```
data Gender
  = Male
  | Female
  | It
  deriving Eq
```

Поскольку пол используется для генерации окончания глагола в прошедшем времени, целесообразно сразу реализовать соответствующие функции. Их будет две, поскольку в русском языке имеется ряд нерегулярных глаголов, которые не имеют канонического окончания «-л» для мужского рода: «смог», «пренебрёг», «нёс» и т. д. – это глаголы с окончанием на «чь», «-ти» в неопределённой форме. Итак, функции:

```
ending :: Gender -> String
ending Male    = ""
ending Female  = "а"
ending It      = "о"


ending' :: Gender -> String
ending' Male    = ""
ending' Female = "ла"
ending' It      = "ло"
```

Функция **ending'** будет использоваться как раз для нерегулярных глаголов.

Теперь можно определить перечень действующих лиц (ещё раз необходимо напомнить, что используется традиционное представление этой сказки):

```
actors ::  [Actor]
actors =  [Actor "Заяц"     "Зайца"     "Зайцу"    Male,
           Actor "Волк"     "Волка"     "Волку"    Male,
           Actor "Медведь"  "Медведя"   "Медведю"  Male,
           Actor "Лиса"     "Лисы"      "Лисе"     Female]
```

Вроде бы всё готово, чтобы реализовать функцию **act**, для которой была произведена подготовительная работа. Однако если вспомнить её тип, который является простой строкой **String**, становится понятно, что эта функция должна состоять из единственного клоза, который вызывает дополнительную (и, возможно, даже локальную) функ-

цию **act'**, в которой осуществляется рекурсивный перебор уже имеющегося списка **actors**. Другими словами, реализация функции **act** использует технологию *накапливающего параметра*, передавая в дополнительную функцию список оставшихся действующих лиц список уже перечисленных персонажей, а также начальное значение результата:

```
act :: String
act = act' actors [] ""
  where
    act' :: [Actor] -> [Actor] -> String -> String
    act' []     _ result = result
    act' [a]    i result = result ++ (finalAct a i)
    act' (a:as) i result = act' as (i ++ [a]) (result ++ (interimAct a i))
```

Функции **interimAct** и **finalAct** ещё должны быть определены. Эти функции возвращают, соответственно, строки, описывающие промежуточное действие и окончательное действие последнего действующего лица (кульминацию). Как видно, обе эти функции принимают на вход текущего персонажа, а также список уже пройденных действующих лиц (обозначен параметром **i**).

Сама функция **act'** просто собирает результат рекурсивного перебора списка персонажей в параметре **result**, значение которого возвращается функцией при достижении пустого списка. В этом и состоит суть технологии использования накапливающего параметра (иначе называемого *аккумулятором*). Если посмотреть на последний клоз функции **act'**, то видно, что производится непосредственно рекурсивный вызов той же функции, а все вычисления производятся при передаче новых значений параметров. Современные компиляторы языка Haskell производят оптимизацию подобных вычислений, производя их при постоянном объёме памяти (при помощи итераций).

Просто реализовать функцию **interimAct**. В ней будет использоваться ещё одна дополнительная функция **meet**, которая возвращает стандартное для всех персонажей сказки описание встречи колобка с ними. Поскольку для последнего персонажа (лисы) описание встречи абсолютно такое же, целесообразно вынести одинаковые строки в одну строку. При этом надо вспомнить, что для первого действующего лица всё-таки имеется небольшое отличие, которое можно обработать при помощи банального условия **if**:

```
interimAct :: Actor -> [Actor] -> String
interimAct actor i = meet actor i ++
                     "Прыгнул колобок, только " ++ an ++
                     " его и видел" ++ ae ++ ".\n"
  where
    an = name actor
    ae = ending $ gender actor


meet :: Actor -> [Actor] -> String
```

```
meet actor i = "Катится колобок " ++
               (if(i== []) then "по тропинке" else "дальше") ++
               ", а навстречу ему " ++ an ++ ". " ++
               "Увидел" ++ ae ++ " " ++ an ++
               "колобка и говорит ему:\n" ++
               "– Колобок, колобок, я тебя съем.\n" ++
               "А колобок  отвечает:\n" ++
               "– Не ешь меня, " ++ an ++
               ", лучше послушай, какую я тебе песенку спою.\n" ++
               "И запел:\n" ++
               song actor i
 where
   an = name actor
   ae = ending $ gender actor
```

Самая интересная функция **song** опять выделена отдельно, но теперь уже больше для эстетических целей (она используется только в функции **meet**). В этой функции как раз и происходит рекурсивный перебор уже встреченных ранее персонажей – колобок поёт свою песенку:

```
song ::  Actor -> [Actor] -> String
song actor i =  " Я – колобок, колобок.\n" ++
                " Колобок – румяный бок.\n" ++
                " По амбарам метён.\n" ++
                " По сусекам скребён.\n" ++
                " На сметане мешён.\n" ++
                " В жаркой печке печён.\n" ++
                " На окошке студён.\n" ++
                " Я от бабушки ушёл.\n" ++
                " Я от дедушки ушёл.\n" ++
                song' actor i ""
 where
   song' actor []     result
     = result ++ "  А от тебя, " ++ (name actor) ++ ", и подавно уйду!\n"
   song' actor (i:is) result
     = song' actor is (result ++ " Я от " ++ (name_g i) ++ "  ушёл.\n")
```

Теперь осталось написать код функции **finalAct**, который, впрочем, будет состоять из вызова всё той же функции **meet** и строк с описанием кульминации:

```
finalAct :: Actor -> [ Actor] -> String
finalAct actor i = meet actor i ++
                   "А " ++ an ++ " и говорит ему:\n" ++
                   "– Ах, какая хорошая песенка. Только вот слаб" ++
                   ae ++ " я на уши стал" ++ ae ++ ". " ++
                   "Будь любезен, прыгни ко мне на нос " ++
                   "и спой свою песенку ещё раз.\n" ++
                   "А колобок и рад, что его песенку похвалили. " ++
```

```
                    "Прыгнул он " ++ ( name_a actor) ++
                    "на нос и только хотел снова запеть, " ++
                    "а " ++ an ++ " его \"Цап!\" и съел"
                    ++ ae ++ ".\n"
  where
    an = name actor
    ae = ending $ gender actor
```

Генератор сказки «Колобок» готов. Конечно, можно задаться вопросом, в чём его прелесть? А суть в том, что этот генератор будет создавать новые сказки вновь и вновь, как только меняется список персонажей. Главное – описывать их с необходимой точностью, задавая три падежа их имён и пол для генерации правильного окончания глаголов в прошедшем времени. И этот код на полторы страницы может генерировать сказки на многие и многие страницы. Конечно, этот код не без огрехов – к примеру, в трёх функциях встречаются абсолютно идентичные локальные определения. Но в целом он представляет собой достаточно простой и эффективный пример генератора естественно-языкового текста специального вида.

2. Теремок

В лесу организовалось некоторое место для жилья – теремок (то ли гнилой пень, то ли оброненный проходящим мужчиной чугунок или потерянная рукавичка). К этому теремку начали подбегать различные жители леса, которые спрашивали, кто проживает в таком знатном домике. Все, кто уже поселились в нём, выходили наружу и представлялись (*рекурсия*), после чего призывали вновь прибывшего присоединиться к весёлой компании. Наконец, к теремку подходит огромный зверь (медведь), который ну никак не мог втиснуться в и без того уже тесное пространство, а потому полезший на крышу. Это действие пагубно отражается на строении, которое ломается, а жильцы внезапно оказываются на улице (*кульминация*).

Реализацию генератора этой сказки можно основать на уже реализованных программных сущностях для сказки «Колобок». К таковым, без сомнений, относятся тип данных **Gender** и функций **ending** и **ending'**. Более того, функция **act**, которая осуществляет запуск рекурсивного перебора действующих лиц, будет также абсолютно идентична (и это не очень удивительно). Само собой, что все эти функции необходимо помещать в новый модуль, поскольку для единообразия остальные функции будут названы также, но в них, естественно, будет новое наполнение, соответствующее рассматриваемой сказке.

Дополнительно к служебным функциям необходимо реализовать функцию, которая для заданного пола персонажа возвращает соответствующее ему местоимение третьего рода единственного числа:

```
third :: Gender -> String
third Male    = "он"
third Female  = "она"
third It      = "оно"
```

Эта функция пригодится при реализации описания кульминации. Начальная же функция выглядит стандартно, в полном соответствии с упомянутой структурой сказки:

```
attic :: String
attic = prologue ++
        act ++
        epilogue
```

Реализация функций **prologue** и **epilogue** для этой сказки достаточно проста:

```
prologue :: String
prologue = "Стоит в лесу теремок, он не низок, не высок.\n"

epilogue :: String
epilogue = "И разбежались звери кто куда."
```

Теперь необходимо подумать над структурой типа **Actor**, описывающей одного персонажа. В этой сказке уже не надо предоставлять программе имена действующих лиц в родительном и винительном падежах, но зато необходимо указать, каким именно образом соответствующий персонаж перемещался по лесу, пока не наткнулся на теремок. Так что определение типа **Actor** будет таковым:

```
data Actor
  = Actor
    {
      name   :: String,
      action :: String,
      gender :: Gender
    }
```

Ну и, собственно, перечень действующих лиц для рассматриваемой сказки в одном из её классических вариантов (таковых несколько и списки персонажей различаются) выглядит следующим образом:

```
actors :: [Actor]
actors = [Actor "Мышка-норушка"        "Бежала"   Female,
          Actor "Лягушка-квакушка"     "Прыгала"  Female,
          Actor "Зайчик-побегайчик"    "Скакал"   Male,
          Actor "Лисичка-сестричка"    "Бежала"   Female,
          Actor "Волчок-серый бочок"   "Рыскал"   Male,
          Actor "Медведь"              "Шёл"      Male]
```

Набор специфических функций для генерации отдельного акта в череде рекурсивной последовательности сказки «Теремок» более сложен, чем для предыдущей сказки. Здесь надо осуществить перебор вселившихся жильцов, которые представляются вновь появившемуся претенденту один за другим. Поэтому функция **interimAct** определяется так:

```
interimAct :: Actor -> [Actor] -> String
interimAct actor i = ask actor i ++
                     "Вот и стал" ++ ae ++ " " ++ an ++ в теремке жить.\n"
  where
    an = name actor
    ae = ending $ gender actor
```

Функция **ask** возвращает строку, в которой текущее действующее лицо спрашивает о том, кто живёт в теремке. Она выглядит следующим образом:

```
ask :: Actor -> [Actor] -> String
ask actor i = (action actor) ++ " " ++ an ++ ". " ++
              "Увидел" ++ ae ++ " теремок и спрашивает:\n" ++
              "– Кто в теремочке живёт? Кто в невысоком живёт?\n" ++
              salute actor i
 where
   an = name actor
   ae = ending  $ gender actor
```

В конце определения этой функции производится вызов функции **salute**, которая как раз и генерирует перечисление всех уже вселившихся жильцов:

```
salute :: Actor -> [Actor] -> String
salute _      []     = "Никто не отзывается. "
salute actor[i]     = "Выглянул" ++ (ending $ gender i) ++
                      " " ++ (name i) ++ " и говорит:\n" ++
                      "– Я –" ++ (name i) ++ ". А ты кто?\n" ++
                      answer actor True
salute actor is     = "Выглянули жильцы и говорят:" ++
                      salute' is "" ++
                      answer actor False
  where
    salute' :: [Actor] ->String-> String
    salute' []     result = result ++ "А ты кто?\n"
    salute' (i:is) result = salute' is (result ++
                            "\n- Я – " ++ (name i) ++ ".")
```

Локальное определение **salute'** является переборной рекурсивной функцией, которая опять использует технологию накапливающего параметра. Во всех значимых клозах функции **salute** также производится вызов функции **answer**, которая выделена опять-таки больше в эстетических целях. Она генерирует ответ нового претендента на жительство в теремке:

```
answer :: Actor -> Bool -> String
```

```
answer actor one = "– А я – " ++ (name actor) ++ ". Пусти" ++
                   (if one then "" else "те") ++
                   " меня к себе жить." ++ "\n" ++
                   " –Да ты не влезешь.\n" ++
                   " –Ну я как-нибудь.\n" ++
                   " –Ну иди.\n"
```

Второй параметр этой функции является булевским и определяет, сколько жильцов живёт в теремке – один или много. Эта информация используется для правильной генерации окончания императивной формы глагола «пустить». Как видно, все три функции **ask**, **salute** и **answer** можно было бы вообще объединить в одном определении, которое вызывалось бы из функции **interimAct**. Тем не менее, для структуризации кода рекомендуется таким образом выделять логически-обособленные части программы.

Наконец, заключительная функция **finalAct** определяется следующим образом:

```
finalAct :: Actor -> [Actor] -> String
finalAct actor i = ask actor i ++
                   " Полез" ++ ae' ++ " " ++ an ++
                   " внутрь, да не смог" ++ ae' ++
                   " туда поместиться. " ++
                   " Тогда влез" ++ ae' ++ " " ++ (third actor) ++
                   " на теремок и раздавил" ++ ae ++ его.\n"
where
  an  = name actor
  ae  = ending  $ gender actor
  ae' = ending' $ gender actor
```

Опять же в построенном генераторе встречаются те же огрехи, которые были обнаружены в функции для вывода текста сказки «Колобок». Тем не менее, уже на примере этого генератора становится понятным, что объявленная во введении задача имеет типовое решение.

(Продолжение в следующем номере)