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
02.03.02 — Фундаментальная информатика
и информационные технологии

ГЕНЕРАЦИЯ ОПИСАНИЙ АЛГЕБРАИЧЕСКИХ ТИПОВ ДАННЫХ НА ОСНОВЕ JSON

Выпускная квалификационная работа
на степень бакалавра

Студента 4 курса
О. И. Маросеева

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

Допущено к защите:
руководитель направления ФИИТ _____ В. С. Пилиди

Ростов-на-Дону
2016

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

В современном мире постоянно усиливаются потоки информации, представляющей собой различного рода структурированные данные. Актуальным представляется вопрос обработки этих потоков. Для решения этой задачи появляются всё новые форматы представления данных. Одним из наиболее популярных форматов обмена структурированными данными является JSON (JavaScript Object Notation).

Постоянное изменение форматов данных вслед за меняющимся миром должно отражаться в системах типов языков программирования, которые используются для создания программ обработки потоков информации. В языках программирования со статической типизацией сложней отражать указанную изменчивость: малейшее изменение формата входных данных может повлечь необходимость больших изменений в коде для того, чтобы он стал, во-первых, компилируемым, во-вторых, рабочим.

Для статически типизированных языков программирования хотелось бы добавить гибкость в указанном отношении, это значит, что следует добавить возможность автоматизированного анализа источников данных и генерации типов на основе этого анализа.

В 2011 году была представлена концепция «поставщиков типов» (`Type Providers`) в языке `F#`. Поставщики типов являются компонентами, которые включают в программу новые методы и типы, основанные на входных данных из внешних источников данных. То есть позволяют программисту работать напрямую с данными без опреде-

ления дополнительных функций для получения и обработки данных. При этом, удобство придает и то, что поставщики типов являются плагинами к компилятору и плагинами к `IDE`. Примеры поставщиков типов: данные в виде JSON, HTML, регулярных выражений и прочие.

Поставщики типов показали свою эффективность в ряде прикладных задач [1]. Возникает вопрос о переносе принципа работы поставщиков типов на другие функциональные языки.

Цель данной работы — создание механизма для анализа входных данных JSON и генерации определений типов функционального языка программирования Haskell на основе проведённого анализа.

Для достижения данной цели в работе поставлены следующие задачи.

- Получение абстрактного синтаксического дерева (далее — AST) по исходному `JSON`.
- Преобразование AST в структуру, пригодную для генерации алгебраического типа данных `Haskell`.
- Использование сгенерированного типа данных в коде.

ГЛАВА 1

ПРЕДВАРИТЕЛЬНЫЕ СВЕДЕНИЯ

1.1. JavaScript Object Notation

JSON (`JavaScript Object Notation`) [2] — простой формат обмена данными, основанный на подмножестве языка программирования `JavaScript`. При этом рассматривемый формат независим от реализации и может использоваться любым языком программирования. Файл в формате `JSON` представляет собой неупорядоченное множество пар ключ-значение, значения которого могут иметь следующий тип:

- объекты (выделяются { ... }),
- массивы (выделяются [...]),
- строки,
- логические выражения (true | false),
- null-значения.

Листинг 1.1.1. Пример данных в формате JSON

```
{
    "firstName": "John",
    "lastName" : "Smith",
    "age" : 25
}
```

Наглядно продемонстрировать вышеобозначенную структуру можно при помощи схем, представленных на рисунках 1.1, 1.2 и 1.3.

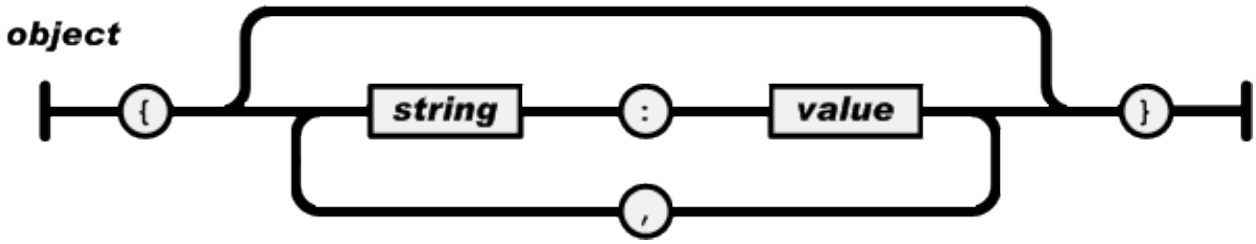

Рисунок 1.1 — Объект

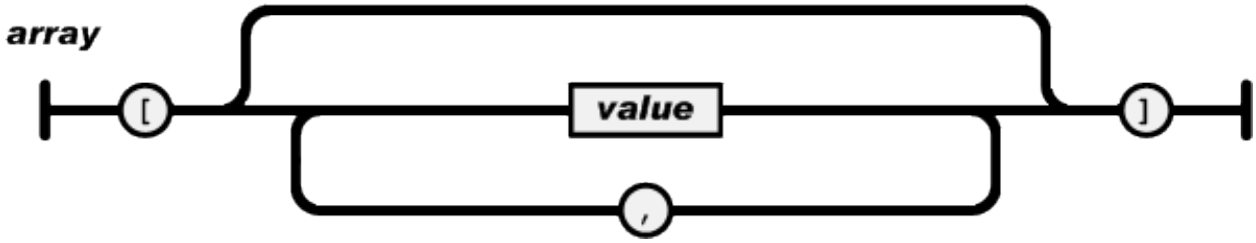

Рисунок 1.2 — Массив

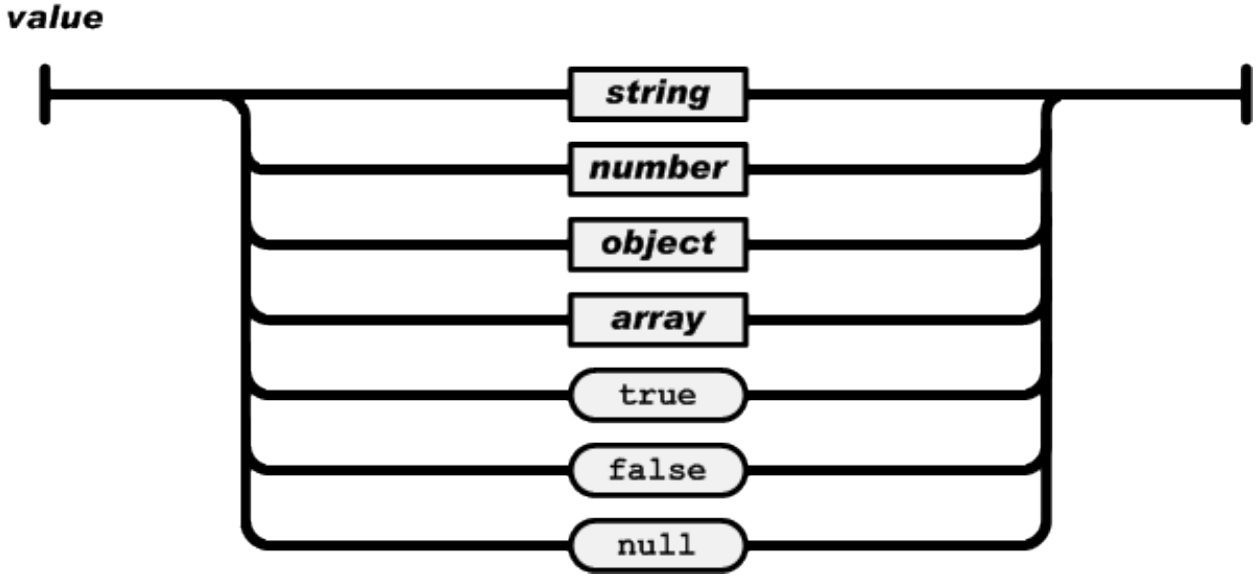

Рисунок 1.3 — Значение

1.2. Алгебраические типы данных

Алгебраические типы данных (далее — АТД) — вид составных типов, представленных типом-произведением, типом-суммой, либо комбинацией: суммой произведений. [3] Последний вариант можно проиллюстрировать на примере двоичного дерева:

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
```

В примере `Tree` является суммой произведений. Сумма определяется знаком «|». А произведение типов простым перечислением (`Node (Tree a) (Tree a)`)

Также существует другой вариант определения типа, который называется синтаксисом записи с именованными полями:

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

Вместо простого перечисления типов созданная структура наполняется полями и их значениями. Главное преимущество заключается в том, что подобный синтаксис генерирует функции для извлечения полей. Также облегчается чтение и понимание типа данных.

Стоит отметить, что с помощью АТД и стандартных типов `Haskell` (например, строковых String, списковых [a] и т. д.) можно представить `JSON`-структуру.

АТД являются фундаментальным понятием. Простота и гибкость таких типов открывают дорогу таких техникам программирования как обобщенное программирование, метапрограммирование и т. д.

1.3. Монада State

Монада `State` применима в тех случаях, когда имеется некоторое состояние, которое подвергается постоянным изменениям. Сто-

ит отметить, что при таком способе мы трансформируем состояние, но при этом не теряем «чистоту» функций.

1.3.1. Control.Monad.State

Стандартный модуль Control.Monad.State [4] определяет тип, оборачивающий вычисление с состоянием:

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

Как следует из определения, вычисление в монаде State возвращает некоторый результат и при этом в случае необходимости меняет состояние. Операции с состояниями реализованы следующими функциями: `get` (получает состояние) и `put` (изменяет состояние на заданное). В листинге 1.3.1 приведен простой пример, позволяющий продемонстрировать возможности монады `State`.

Листинг 1.3.1. Пример использования монады State

```
tick :: State Int Int
tick = do n <- get
          put (n+1)
          return n

ghci> runState tick 3
(3,4)
```

Вычисление в монаде запускается с помощью `runState` и передаваемого состояния. В функции `tick` происходит получение входного значения `3` (`get`), далее меняется состояние (`put`) и на выходе возвращается значение(`return`).

Также модуль содержит и другие полезные функции для работы с состояниями. Некоторые из них представлены в листинге 1.3.2. Функции с суффиксом -State отличаются типом возвращаемого значения. Необходимое поведение можно выбрать исходя из названия функции.

Листинг 1.3.2. Функции модуля Control.Monad.State

```
modify :: MonadState s m => (s -> s) -> m ()

execState:: State s a -> s -> s

runState:: State s a -> s -> (a, s)

evalState:: State s a -> s -> a
```

Например, функция `modify` преобразует внутреннее состояние функцией, которую получает на вход. Можно реализовать код листинга 1.3.1 через `modify` (листинг 1.3.3).

Листинг 1.3.3. Функция modify модуля Control.Monad.State

```
tick :: State Int Int
tick = do modify (+1)
          return n

ghci> runState tick 3
(3,4)
```

1.4. Библиотека Data.Aeson

1.4.1. Обзор

`Data.Aeson` — библиотека для работы с файлами в формате `JSON`, написанная на языке `Haskell`. [5] В данной библиотеке используются два основных класса типов — `FromJSON` и `ToJSON`. [6] Типы, имеющие возможность кодирования/декодирования, должны быть экземплярами классов `FromJSON`, `ToJSON`. Самый простой способ использования библиотеки заключается в определении типов данных и экземпляров `FromJSON`, `ToJSON`.

Существует возможность определить экземпляры для кодирования/декодирования по умолчанию благодаря инструкции компилятора `DeriveGeneric` и экземпляру `Generic` для кодируемых/декодируемых типов. Рассмотрим листинг 1.4.1, демонстрирующий данную возможность с условным типом данных.

Листинг 1.4.1. Создание экземпляров по умолчанию

```
{-# LANGUAGE DeriveGeneric #-}
import GHC.Generics

data DataName = DataName {
        field1::Int,
        field2::Int
     } deriving (Generic, Show)

instance ToJSON DataName

instance FromJSON DataName
```

1.4.2. Работа с AST

`Data.Aeson` имеет свой собственный тип для представления конвертируемого `JSON`-файла. Этот тип называется `Value` и имеет шесть конструкторов значения:

Листинг 1.4.2. Конструкторы Value

```
data Value
  = Object Object
  | Array Array
  | String Text
  | Number Scientific
  | Bool Bool
  | Null
```

Таким образом Aeson позволяет получить абстрактное синтаксическое дерево по JSON. Это бывает полезно в случаях, когда неизвестен тип данных, соответствующий входному файлу. Имея AST, можно написать функцию для его обхода. [6]

В качестве примера рассмотрим получение AST для двух случаев: простой (листинг 1.4.3) и более сложный со вложенными объектами (листинг 1.4.4).

Листинг 1.4.3. JSON без вложенных объектов

```
decode :: FromJSON a => ByteString -> Maybe a

ghci> decode "{\"foo\": 123}" :: Maybe Value
Just (Object (fromList [("foo",Number 123)]))
```

Листинг 1.4.4. JSON со вложенными объектами

```
ghci> decode "{\"foo\": [\"abc\",\"def\"]}" :: Maybe Value
Just (Object (fromList [("foo", Array (fromList [String "abc",
                                                 String "def"]))]))
```

1.5. Template Haskell

Template Haskell — это расширение языка Haskell, реализующее средства для метапрограммирования. [7] Оно позволяет использовать Haskell одновременно как язык исходный, так и целевой. Поскольку работа ведется с расширением, в исполняемый файл необходимо добавить директиву:

```
{-# LANGUAGE TemplateHaskell #-}.
```

1.5.1. Монада Q

Монада `Q` оборачивает значения типов, предназначенных для последующей генерации других конструкций на языке `Haskell`. [8] Такие типы полностью удовлетворяют синтаксису языка и представляют собой абстрактное синтаксическое дерево кода на `Haskell`: тип `Exp` — для генерации выражений, `Dec` — для объявлений, `Lit` — для литералов и т.д. [9]

1.5.2. Модуль Language.Haskell.TH.Syntax

Тип `Exp` определен в модуле `Language.Haskell.TH.Syntax` [10] и `Exp` имеет семнадцать конструкторов значения. В листинге 1.5.1 представлены первые четыре из них.

Листинг 1.5.1. Конструкторы Exp

```
data Exp
       = VarE Name
       | AppE Exp Exp
       | MultiIfE [(Guard, Exp)]
       | CondE Exp Exp Exp
       | ...
```

1.5.3. Вклейка (splicing)

Вклейка кода — преобразование шаблона (структура `Template Haskell`) с данным параметром в обычный `Haskell`-код во время компиляции и вклеивает его на то же место. Вклейка производится оператором `\$(...)`. Важно, чтобы между скобками и оператором `\$` не было пробелов. Также существует ограничение по использованию вклейки: ее можно применить только из другого модуля, т.е. отсутствует возможность "вклейки"шаблона в одном

Таблица 1.1. — Использование вклейки

Где используется	Тип
Выражения	Q Exp
Объявления верхнего уровня	Q [Dec]
Типы	Q Type

модуле вместе с его определением. Вклейка используется в разных компонентах языка (см. таблицу 1.1).

ГЛАВА 2

ГЕНЕРАЦИЯ АТД ПО JSON-ФАЙЛУ

2.1. Выбор средств для реализации

Изначально при реализации задачи, поставленной во введении, стоял выбор между несколькими средствами. Рассмотрим каждое из них.

2.1.1. Расширение к языку Haskell

При подобной реализации становятся очевидными некоторые преимущества, связанные с простотой использования (необходимо лишь прописать директиву, добавляющую это расширение), но при этом она представляется сложной в реализации, поскольку для ее осуществления необходимо изменить исходные коды компилятора `GHC`. При этом в данной реализации не хватало бы наглядности и синтаксис отличался бы от привычного.

2.1.2. Написание парсера

Данное решение позволяет использовать стандартные методы и наработки, связанные с парсерами, что предоставляет возможность лучше контролировать представление для генерации. При этом можно вывести готовый алгебраический тип данных. Но в данном случае

возникает проблема: как использовать этот тип незамедлительно без дополнительных манипуляций.

2.1.3. Template Haskell

Данный подход оказался предпочтительнее остальных по ряду причин. Во-первых, при его использовании отсутствуют недостатки, присущие другим подходам, в том числе существует возможность использования сгенерированного типа без дополнительных накладных ресурсов и средств. Во-вторых, предоставляется возможность генерирования данных и последующего незамедлительного их использования. Для получение AST будет использоваться ранее упомянутая библиотека `Data.Aeson`. Помимо этого, данная идея представляется более гибкой с точки зрения практической реализации. Возможно, аналогичные средства дают некоторые преимущества, но они не так существенны с точки зрения эффективности.

2.2. Программная реализация генератора АТД

2.2.1. Применение DataD (Template Haskell)

Программная реализация средствами `Template Haskell` генерирует объявление типов данных: используется конструктор значения `DataD` типа `Dec` (листинг 2.2.1). `DataD` является конструктором значения `Dec`. Тип `Cxt` определяет классы типов, которым должны принадлежать различные типы, входящие в определение АТД.

`Name` — абстрактный тип, представляющий имена в синтаксическом дереве. Используется для определения имени полей и конструкторов. Функция `mkName` (листинг 2.2.2) создает значение типа `Name` из обычной строки (`String`), с ее содержанием в качестве имени.

Листинг 2.2.1. Конструктор значения типа Dec

```
Dec
   = DataD Cxt Name [TyVarBndr] [Con] Cxt
   | ...
```

Листинг 2.2.2. Особенность: чистота функции mkName

```
mkName :: String -> Name
```

В коде реализации используется тип из листинга 2.2.1 для вклейки в объявлениях верхнего уровня. Тип `RecC` дает понять, что сгенерированный тип должен представлять собой запись с именованными полями. Использование класса типов `Generic` понадобится в будущем, когда будут генерироваться экземпляры классов `FromJSON` и `ToJSON` для вклеиваемого типа (см. раздел 1.4).

Листинг 2.2.3. Генерация Data в тексте программы

```
DataD
     []
     (mkName $ firstLetterToUpper key')
     []
     [ RecC (mkName $ firstLetterToUpper key')  (result) ]
     [mkName "Generic", mkName "Show", mkName "Eq"]
```

Важно помнить, что имена типов данных в `Haskell` начинаются с заглавных букв. Для соблюдения этого правила определим функцию `firstLetterToUpper` (листинг 2.2.4).

Листинг 2.2.4. Функция смены первой буквы на заглавную

```
firstLetterToUpper :: String -> String
firstLetterToUpper (x:xs) = (Data.Char.toUpper $ x) : (xs)
```

Листинг 2.3.1. Монадическая свертка foldlWithKeyM

```
import qualified Data.HashMap.Strict as StrHash
import qualified Data.Foldable    as FB

foldlWithKeyM :: (Monad m) => (b -> k -> a -> m b) -> b ->
                                        StrHash.HashMap k a -> m b
foldlWithKeyM f b = FB.foldlM f' b . StrHash.toList
  where f' a = uncurry (f a)
```

2.3. Использование монады State

Для аккумуляции типов `Dec` был необходим аналог глобальной переменной в императивных языках программирования. В функциональных языках, как правило, отсутствуют подобные средства в чистом виде, однако похожего поведения можно добиться, используя монаду `State`:

```
State [Dec] ()
```

Накапливаются типы для генерации с помощью функции `modify` из модуля `Control.Monad.State`. [11] Однако существует проблема: каждый раз, когда используется `modify`, порождается новый экземпляр типа `State [Dec] ()`, который теряет накопленные результаты обхода, то есть условно при таком подходе не будет глобальной переменной, в которую собираются все необходимые данные.

Для того чтобы решить возникшую проблему, нам необходима монадическая свертка по ключу и значению (см. листинг 2.3.1). Свертка реализована с помощью `foldlM` из модуля `Data.Foldable` и функции `uncurry` (преобразует каррированную функцию в функцию, принимающую пару).

Еще одно преимущество использования функционального языка `Haskell` заключается в том, что существуют полезные средства для поиска необходимых функций — поиск по сигнатуре. Данную возможность реализует сервис `Hayoo` [12]. С помощью него была найдена `foldlWithKeyM`.

2.4. Формирование АТД

Основной функцией, которая запускает проход по AST и накапливает значения в монаде `State` является:

```
convertObject:: String -> Value -> State [Dec] ()
```

Функция `convertObject` получает на вход имя типа данных и AST, полученное с помощью `decode` (`Data.Aeson`). Затем происходит спуск по дереву и анализ его вершин:

```
convertFields:: MonadState [Dec] m => Value ->
                                          m [(Name, Strict, Type)]
```

Как следует из сигнатуры, анализ и спуск происходит в монаде. Именно при проходе по вершинам мы используем написанный ранее `foldlWithKeyM`. Главная задача — учитывать, что при обходе мы можем встретить вложенные в объект объекты и вызвать функцию рекурсивно. Для этого мы используем функцию `isObject` (листинг 2.4.1). `convertFields` получает массив из `(Name, Strict, Type)`, который подставляется в (листинг 2.2.3) вместо переменной `result`.

Листинг 2.4.1. Проверка на принадлежность Object

```
isObject :: Value -> Bool
isObject (Object obj) = True;
isObject _ = False;
```

Все необходимые для генерации данные собираются за один проход AST. При анализе дерева, в том случае если алгоритм доходит до листового узла, вызывается `return`, который возвращает `[(Name, Strict, Type)]` (листинг 2.4.2). И далее продолжается обход. `MState` — квалифицированый импорт модуля `Control.Monad.State`.

Листинг 2.4.2. Простой случай при обходе

```
do
      (MState.return (((mkName $  key'), NotStrict,
                                (mkValType val' key') ) : list'))
```

Если при проходе по дереву срабатывает условие из листинга 2.4.2, происходит рекурсивный спуск — вызов функции `convertFields` с вложенным найденным объектом (листинг 2.4.3).

Листинг 2.4.3. Сложный случай при обходе

```
do
      result <- convertFields $ val'
      MState.modify ((Prelude.++) [DataD ... ])
      (MState.return (((mkName $  key'), NotStrict,
                                (mkValType val' key') ) : list'))
```

Таким образом была описана вся логика программы. Обрабатывается каждый вложенный объект и каждый лист AST. На выходе получаем структуру `[Dec]` и оборачиваем ее в монаду `Q` для вызова вклейки из другого модуля через оператор `\$(...)`.

ГЛАВА 3

ПРИМЕР ИСПОЛЬЗОВАНИЯ ПОЛУЧЕННОГО ИНСТРУМЕНТА

`Git`-репозиторий с исходным кодом на языке `Haskell` доступен по адресу [13]. Тестовые примеры запускались на компиляторе `GHC` версии `7.10.3`. Библиотека собрана в `cabal`-пакет и также доступна [13]. Для компиляции библиотеки нужно перейти в его каталог и выполнить стандартную команду:

```
$ cabal install
```

3.1. Отладка программ с Template Haskell

Компилятор `GHC` предоставляет мощные инструменты для отладки программы. [14] К примеру, можно ставить контрольные точки, получать подробное описание ошибок, генерировать полезные структуры, смотреть на AST и т.д. Для этого используются флаги компилятору.

Полезным флагом для программы, использующей `Template Haskell` служит `-ddump-splices`. Его средствами организован вывод полученного из шаблона выражения либо ошибки (информативное сообщение).

Листинг 3.1.1. Запуск отладчика с флагом -ddump-splices

```
on.hs:8:3-17: Splicing declarations
    getDataFromJSON
  ======>
    data JSONData
      = JSONData {name :: String}
      deriving (Generic, Show, Eq)
```

3.2. Простой пример

Для начала будет рассмотрен простой пример. На вход программе подается простой (без вложенных объектов) `JSON` (листинг 3.2.1).

Листинг 3.2.1. Вход программы

```
{
    "name" : "Joe",
    "age" : 25,
    "avg" : 4,
    "arr" : [1,2,3]
}
```

В итоге мы получаем тип данных с именованными полями, полностью соответствующий постановленной задаче:

```
data JSONData
        = JSONData {arr :: [Float]
                    name :: String,
                    age :: Float,
                    avg :: Float}
          deriving (Show, Eq, Generic)
```

3.3. Более сложный пример

На вход программе подается JSON со вложенным объектом (листинг 3.3.1). Стоит отметить то, как он будет представлен.

Листинг 3.3.1. Вход программы

```
{
    "name" : "Joe",
    "age" : 25,
    "avg" : 4,
    "arra" :
             {
                 "fg" : "JSONTest"
             }
}
```

После выполнения программы мы получаем два типа данных, что полностью соответствует заявленным требованиям к программе. Первый тип в своем определении использует второй.

```
data JSONData
      = JSONData {name :: String,
                  arra :: Arra,
                  age :: Float,
                  avg :: Float}
        deriving (Show, Eq, Generic)

data Arra
      = Arra {fg :: String}
        deriving (Show, Eq, Generic)
```

3.4. Соответствие синтаксису JSON

Типы данных, получаемые после работы программы полностью удовлетворяют синтаксису JSON и в полной мере покрывают все воз-

можные случаи, в том числе поля-списки. Это говорит о том, что в языке `Haskell` в полной мере могут использоваться так называемые «провайдеры типов» для `JSON`-файлов.

3.5. Развернутый пример

Более развернутый пример см. в Приложении А.

ЗАКЛЮЧЕНИЕ

В данной работе была реализована генерация алгебраических типов данных функционального языка программирования Haskell по исходным данным в формате JSON. Данная идея заимствована из языка F#, где были добавлены так называемые «поставщики типов» (Type Providers). В тексте работы приводится обзор реализации задачи на Haskell. При этом программа была опробована на нескольких вариантах входных данных.

При подготовке активно использовались сильные стороны функционального программирования. Выразительность и высокоуровневость языка Haskell придает лаконичности коду, что позволяет зачастую решать сложные задачи с помощью достаточно простого и короткого кода. В качестве основных инструментов для реализации поставленной задачи используются библиотека Data.Aeson и расширение Template Haskell. Генерация производится с помощью рекурсивного прохода по абстрактному синтаксическому дереву и накопления результатов обхода в монаде State.

Данная работа дает представление о средствах метапрограммирования на языке Haskell, а также демонстрирует возможность применения идеи поставщиков типов в других языках программирования.

СПИСОК ЛИТЕРАТУРЫ

1. JSON TypeProvider. — URL: http://fsharp.github.io/FSharp.Data/library/JsonProvider.html.
2. Стандарт обмена данными JSON. — URL: http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf (дата обр. 03.10.2015).
3. *Липовача М.* Изучай Haskell во имя добра! — ДМК Пресс, 2012.
4. Control.Monad.State, монада State. — URL: https://hackage.haskell.org/package/mtl-2.2.1/docs/Control-Monad-State.html (дата обр. 17.09.2015).
5. Aeson, библиотека для получения AST формата JSON. — URL: https://github.com/bos/aeson (дата обр. 12.12.2015).
6. Обзор Data.Aeson и примеры использования. — URL: https://www.schoolofhaskell.com/school/starting-with-haskell/libraries-and-frameworks/text-manipulation/json (дата обр. 03.02.2016).
7. Описание известных расширений Haskell. — URL: https://ocharles.org.uk/blog/pages/2014-12-01-24-days-of-ghc-extensions.html (дата обр. 01.03.2016).
8. *Tim Sheard S. P. J.* Template Meta-programming for Haskell // Proc. Haskell Workshop, 2002, Pittsburgh. — 2002.

9. Language.Haskell.TH.Syntax, абстрактные синтаксические типы данных. — URL: https://downloads.haskell.org/~ghc/7.6.3/docs/html/libraries/template-haskell-2.8.0.0/Language-Haskell-TH-Syntax.html (дата обр. 19.05.2016).
10. *Lemmer R.* Cover Haskell Design Patterns. — Packt, 2015.
11. Описание монады State. — URL: https://wiki.haskell.org/State_Monad (дата обр. 02.04.2016).
12. Сервис поиска функций по сигнатуре. — URL: http://hayoo.fh-wedel.de/.
13. Репозиторий с кодом генерации АТД. — URL: https://github.com/olegmaroseev/JSONtoDATA.
14. Debugging the compiler, GHC. — URL: https://downloads.haskell.org/~ghc/7.6.2/docs/html/users_guide/options-debugging.html (дата обр. 13.03.2016).

ПРИЛОЖЕНИЕ А

РАЗВЕРНУТЫЙ ПРИМЕР

Листинг А.0.1. JSON-файл на входе программы

```
{
  "glossary": {
    "title": "example glossary",
    "glossDiv": {
      "tit": "S",
      "glossList": {
        "glossEntry": {
          "id": "SGML",
          "sortAs": "SGML",
          "glossTerm": "Standard Generalized Markup Language",
          "acronym": "SGML",
          "abbrev": "ISO 8879:1986",
          "glossDef": {
            "para": "JS used to create markup languages such as DocBook.",
            "glossSeeAlso": [
              "GML",
              "XML"
            ]
          },
          "glossSee": "markup"
        }
      }
    }
  }
}
```

Листинг A.0.2. Полученный тип данных

```
data JSONData
      = JSONData {glossary :: Glossary}
      deriving (Generic, Show, Eq)
data Glossary
      = Glossary {title :: String, glossDiv :: GlossDiv}
      deriving (Generic, Show, Eq)
data GlossDiv
      = GlossDiv {tit :: String, glossList :: GlossList}
      deriving (Generic, Show, Eq)
data GlossList
      = GlossList {glossEntry :: GlossEntry}
      deriving (Generic, Show, Eq)
data GlossEntry
      = GlossEntry {glossDef :: GlossDef,
                    glossTerm :: String,
                    id :: String,
                    glossSee :: String,
                    abbrev :: String,
                    sortAs :: String,
                    acronym :: String}
      deriving (Generic, Show, Eq)
data GlossDef
      = GlossDef {glossSeeAlso :: [String], para :: String}
      deriving (Generic, Show, Eq)
```