Haskell-flavoured Java

Никита Ешкеев, 29.10.2020

Автор

Никита Ешкеев - разработчик с 10 летним опытом в разработке крупных программных комплексов уровня предприятия с применением Java технологий. Имеет ряд нетривиальных коммитов в Scala 3. В настоящее время работает в компании JetBrains, где занят в команде развития поддержки языка Java в Intellij IDEA.

Telegram: neshkeev
Email: neshkeev@yandex.ru

Disclaimer

Сведения в этом докладе отражают лишь личную точку зрения автора и могут не совпадать с официальной позицией компании JetBrains.

Доклад не может рассматриваться в качестве самостоятельного учебного пособия по функциональному программированию или теории типов, поэтому многие рассматриваемые концепции сильно упрощены, а в качестве названий некоторых терминов умышленно использованы англицизмы.

Содержание

- Функциональное программирование
- Алгебраические типы данных
- Pattern matching
- Полиморфизм на уровне кайндов и типы высшего порядка
- Lightweight HKT
- Функторы и монады
- Монадичесие парсер-комбинаторы
- JSON парсер

Функциональное программирование

Минимальные сведения

Функциональное программирование

- Лямбда функция
- Замыкание (closure)
- Композиция функций
- Соглашения о записи типов функций

Лямбда-функция

Лямбда функция - это анонимная функция, которую можно присвоить переменной соответствующего типа или передать в качестве параметра в метод. Особенность лямбда-функции состоит в том, что ее определение простирается вправо на столько на сколько это возможно.

Пример:

```
Function<Integer, Integer> plus2 = x -> x + 2;
new HashMap<String, Integer>()
  .computeIfAbsent(“Hello”, s -> s.length());
```

Замыкание

Тело лямбда-функции может либо использовать только переданные параметры, либо обращаться к переменным окружения, в котором она лексически определена. Лямбда-функция, которая обращается к переменным своего окружения, называется замыканием. Замыкания являются важной частью монадического связывателя, поэтому очень важно понимать эту концепцию.

Пример:

```
Optional<String> helloWorld = Optional.of("Hello")
  .flatMap(hello -> Optional.of("World")
    .flatMap(world -> Optional.of(hello + ", " + world + "!")
));
```

Композиция функций

Каждая функция имеет параметры и результат, а следовательно, можно создавать новые функции из имеющихся, если передать результат одной функции в качестве аргументов другой функции. Это называется композицией функций

Пример

```
Function<String, Integer> length = s -> s.length();
Function<Integer, Boolean> isEven = n -> n % 2 == 0;
Function<String, Boolean> evenStringLengths = s ->
                                    length.andThen(isEven).apply(s);
Function<String, Boolean> evenStringLengthsEta = length.andThen(isEven);
```

Соглашение о записи типов

В процессе рассуждения о функциях записывать типы в привычном java стиле, вроде

```
Function<String, Function<String, Function<String, Integer>>>
```

может быть не очень наглядно, поэтому в этом докладе будет использоваться стрелочная нотация, принятая в функциональных языках, таким образом тип функции выше будет записан как

```
String -> String -> String -> Integer
```

, т.е функция принимает три аргумента `String` и возвращает `Integer`

Вопросы

(по функциональному программированию)

Алгебраические типы данных

Алгебраические типы данных

- Определение
- Тип “произведение”
- Тип “сумма”
- Соглашение о записи ADT
- Примеры ADT
- Зачем ADT

Определение

Алгебраические типы данных (ADT) - это способ представления любого составного типа данных. Термин “алгебраические” говорит о том, что любую структуру такого типа можно разложить на алгебраические операции “сумма” и “произведение”

Тип “произвдение”

Тип “произведение” (product type) - это структура, для конструирования которой необходимы объекты всех типов, которые входят в тип “произведение”. В java это равносильно понятию класс.

Тип “сумма”

Тип “сумма” (sum type) - это структура, для конструирования которой необходим хотя бы один объект из типов, которые входят в тип “произведение”. В java это отдаленно похоже на enum.

Определение типа “сумма”

Полноценное создание типа “сумма” возможно с использованием “запечатанной” иерархии:

- Определяется абстрактный класс с приватным конструктором
- Все наследники абстрактного класса определяются внутри абстрактного класса (у них есть доступ приватному конструктору абстрактного класса)
- Любой наследник абстрактного класса может быть передан туда, где ожидается абстрактный класс (Liskov substitution principle)
- Никто снаружи не может объявить новых наследников абстрактного класса, т.к. конструктор абстрактного класса является приватным

Соглашения о записи ADT

В процессе обсуждения ADT будет использоваться запись типа, принятая в Haskell

- Тип “произведение”: `ProductType = ProductType A B`
- Тип “сумма”: `SumType = A | B`
- Если тип параметризован другим типом (является конструктором типов), то параметризованный тип записывается с маленькой буквы:
  `Optional a = None | Some a`

Примеры ADT

Каждый ADT соответствует какому-то алгебраическому уравнению, например:

- Тип Unit (Синглтон): `Unit = Unit <==> y = 1`
- Тип Id: `Id a = Id a <==> y = x`
- Тип Bool: `Bool = True | False <==> y = 1 + 1 = 2`
- Тип Optional: `Optional a = None | Some a <==> y = 1 + x`
- Тип List: `List a = Nil | Cons a (List a) <==> y = 1 + x * y`
- Тип Tree: `Tree a = Leaf | Branch (Tree a) a (Tree a) <==> y = 1 + y * x * y`

Для таких уравнений справедливы ассоциативный (`a + (b + c) = (a + b) + c`) и дистрибутивный (`a * (b + c) = a * b + a * c`) законы

Зачем ADT

ADT позволяет превратить программу в математический объект, с которым можно совершать математические операции, а, следовательно, можно формально доказывать какие-то свойства программы и проводить ее трансформации, например, с целью оптимизации

Вопросы

(по алгебраическим типам данных)

Pattern matching

Pattern matching

- Мотивация
- Реализация на примере List
- Пример использования

Мотивация

ADT диктует строгую структуру, которую можно декомпозировать естественным способом. Pattern matching обобщает эту идею. Реализация pattern matching состоит в том, чтобы определить новую функцию `caseOf`, которая принимает в качестве параметров продолжения (continuations, функции) по одному на каждый компонент ADT.

Реализация на примере List

В класс `List<T>` добавляется функция `caseOf` со следующей сигнатурой:

```
public <R> R caseOf(
  Supplier<R> nilOp,
  BiFunction<T, List<T>, R> consOp
)
```

Теперь на каждом объекте `List` можно вызывать `caseOf` с обработкой каждого случая:

```
List<T> list = …
list.caseOf(
  () -> // обработать Nil
  (head, tail) -> // обработать Cons
)
```

Пример использования

Поверх функции `caseOf` для `List<T>` можно реализовать свертку:

```
public<R> R fold(BiFunction<T, R, R> reducer, R initial) {
  return caseOf(
    () -> initial,
    (head, tail) -> reducer.apply(head, tail.fold(reducer,
initial))
  );
}
```

Использование этой свертки для суммирования элементов в списке

```
List<T> list = …
int sum = list.fold((next, acc) -> next + acc, 0);
```

Вопросы

(по pattern matching)

Типы высшего порядка

Полиморфизм на уровне кайндов

Типы высшего порядка

- Мотивация
- Конструкторы типов
- Кайнды
- Конструкторы типов и конструкторы данных

Мотивация

ADT можно рассматривать как контейнер для значений, поэтому возникает желание выполнять преобразования этих значений внутри контейнера каким-то универсальным способом, не меняя структуру контейнера. Самый простой способ решения - это полиморфизм на основе перегруженных функций:

- Определяется утилитный класс
- В этом классе определяется набор методов с одинаковым именем, но разными параметрами

Это рабочий подход, но на уровне типов нет никакой возможности указать, что для какого-то произвольного контейнера есть метод, который может преобразовывать его значения.

Мотивация. Продолжение

В таких ситуациях обычно помогает *ad hoc* полиморфизм на базе интерфейсов. Но, к сожалению, в java невозможно написать конструкцию вроде:

```
interface Functor<F> {
  // (a -> b) -> f a -> f b
  <A, B> F<B> map(Function<A, B> fun, F<A> cont);
}
```

Такая запись приводит к ошибке:

```
Error: Type 'F' does not have type parameters.
```

Мотивация. Окончание

Функциональные языки Haskell и Scala предлагают решение этой проблемы через полиморфизм типов на уровне кайндов (HKT), например, аналогичная конструкция является допустимой и в Haskell, и в Scala:

Haskell

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

Scala

```
trait Functor[F[_]] {
  def map[A, B](fun: A => B, cont: F[A]): F[B]
}
```

В языке Java нет соответствующих языковых конструкций, чтобы решить эту проблему. Но есть решение: Lightweight HKT (легковесный полиморфизм на уровне кайндов)

Конструкторы типов

Если посмотреть на утилитный класс, то можно заметить, что сигнатуры функций почти одинаковые: набор джериков одинаковый, каждая принимает первым параметром функцию `A -> B`, каждая принимает контейнер типа `A` и возвращает контейнер типа `B`:

```
class MapperUtil {
  static <A, B> Id       <B> map(Function<A, B> fun, Id       <A> c) { ... }
  static <A, B> List     <B> map(Function<A, B> fun, List     <A> c) { ... }
  static <A, B> Optional <B> map(Function<A, B> fun, Optional <A> c) { ... }
  static <A, B> Tree     <B> map(Function<A, B> fun, Tree     <A> c) { ... }
}
```

Разница лишь в том, что параметризовано дженериком. Эта часть называется конструктором типа.

Кайнды

Конструктор типа позволяет строить новые типы из имеющихся типов. В java используют термин дженерики. Количество параметров конструктора типа определяет его арность (arity). Арность делит все множество конструкторов типов на классы эквивалентности. Эти классы эквивалентности называются кайндами (kind).

Конструкторы типов и конструкторы данных

Конструкторы типов отличается он конструкторов данных (обычных java конструкторов) тем, в каком контексте они могут использоваться: тогда как конструктор данных вызывается во время исполнения программы для создания новых значений, конструкторы типа вызываются во время компиляции.

Полиморфизм на уровне кайндов

Полиморфизм на уровне кайндов (Higher-kinded polymorphism, Higher-kinded types, HKT) - это способ абстрагироваться по конструкторам типа. В Haskell это краеугольный камень для определения функторов, монад и прочих функциональных конструкций. Хотя в java отсутствует прямой способ определения HKT, но есть способ достичь этого эффекта при помощи легковесного полиморфизма на уровне кайндов (Lightweight HKT)

Вопросы

(по типам высшего порядка и НКТ)

Lightweight HKT

Легковесный полиморфизм на уровне кайндов

Lightweight HKT

- Определение
- Интерфейс App
- Инъекция
- Проекция
- Кайнд класс и маркер класс

Определение

Для определения легковесного HKT необходимо определить несколько новых абстракций:

- Интерфейс `App`, который кодирует идею `F<A>`, где и `F` и `A` являются полиморфными
- Кайнд класс и маркер класс
- Конвертация конкретного типа в (инъекция) `App` и из него (проекция)

Интерфейс App

Задача интерфейса `App` состоит в том, чтобы закодировать идею применения полиморфного типа к полиморфному конструктору типа, таким образом его сигнатура следующая

```
interface App<F, A> {}
```

эта абстракция равносильна `F<A>`, где оба `F` и `A` являются полиморфными.

Можно заметить, что у интерфейса `App` нет ни одного метода, поэтому нужны конвертеры в `F<A>` и обратно в `App`, где `F` - это уже конкретный тип

Инъекция

Необходимо сделать `F<A>` наследником `App<F, A>`, но определение имеет свою особенность.

```
class OptionalK<T> implements App<OptionalK.mu, T> {
  public static final class mu {}
  ...
}
```

Здесь `OptionalK.mu` передается в `App` вместо `OptionalK`, т.к. использование обычного `OptionalK` приведет к предупреждению rawtypes.

Правила наследования java позволяют передать `OptionalK<T>` везде, где ожидается `App<OptionalK.mu, T>`. Такой подход определяет конвертацию `F<A>` в `App<F, A>`

Проекция

Конвертация из `App<F, A>` в `F<A>` выполняется при помощи нисходящего преобразования типов.

```
class OptionalK<T> implements App<OptionalK.mu, T> {
  public static final class mu {}
  ...
  static<T> OptionalK<T> narrow(App<OptionalK.mu, T> value) {
       return (OptionalK<T>) value;
  }
}
```

Кайнд класс и маркер класс

Для каждого типа, который необходимо использовать как полиморфный конструктор типов, необходимо определить кайнд класс, который реализует интерфейс `App`. Таким образом кайнд класс является оберткой над этим конструктором типов.

Каждый кайнд класс должен иметь свой маркер класс.

Пример

Этот пример демонстрирует конвертацию `OptionalK<Integer>` в `App<OptionalK.mu, Integer>`и обратно:

```
App<OptionalK.mu, Integer> appOpt = OptionalK.some(42);
OptionalK<Integer> optK = OptionalK.narrow(appOpt);

final int value = optK.getDelegate().caseOf(
    () -> 0,
    Function.identity()
);

assert value == 42 : "Original value changed";
```

Вопросы

(по Lightweight HKT)

Функторы и монады

Функторы и монады

- Интерфейс Functor
- Пример реализации OptionalFunctor
- Использование OptionalFunctor
- Монады
- Интерфейс Monad
- Пример реализации IdMonad
- Пример использования Monad

Интерфейс Functor

Интерфейс Functor теперь можно определить следующим способом

```
public interface Functor<F> {
    // (a -> b) -> f a -> f b
    <A, B> App<F, B> map(Function<A, B> fun, App<F, A> value);
}
```

Типы `F`, `A` и `B` являются полиморфными.

Пример OptionalFunctor

Теперь функтор для `Optional` можно определить следующим способом:

```
public enum OptionalFunctor implements Functor<OptionalK.mu> {
     INSTANCE;

     @Override
     public <A, B> OptionalK<B> map(Function<A, B> fun, App<OptionalK.mu, A> value) {
       final Optional<A> delegate = OptionalK.narrow(value).getDelegate();
       return delegate.caseOf(
               () -> OptionalK.none(),
              val -> fun.andThen(Optional::some).andThen(OptionalK::new)
                        .apply(val)
       );
     }
}
```

Использование OptionalFunctor

Пример добавляет строку “World” в конец элемента внутри контейнера `Optional`, если элемент существует:

```
OptionalFunctor optF = OptionalFunctor.INSTANCE;
optF.map(s -> s + “, World!“, OptionalK.some(“Hello”));
optF.map(s -> s + “, World!“, OptionalK.none());
```

Монады

Монада - это ~~моноид в категории эндофункторов~~ шаблон проектирования в функциональных языках программирования. Монада является обобщением функтора. Чтобы сделать функтор монадой, необходимо реализовать два дополнительных метода:

- `pure` - семантика заключается в том, чтобы положить любое значение (включая функции) в контейнер
- `flatMap` (монадический связыватель) - семантика заключается в том, чтобы применить функцию к значению в контейнере, произведя все эффекты контейнера

Монада является обобщением функтора, поэтому можно реализовать в общем виде метод `Functor#map` при помощи методов интерфейса `Monad`

Интерфейс Monad

Интерфейс `Monad` определяется следующим образом:

```
public interface Monad<M> extends Functor<M> {
    // a -> m a
    <A> App<M, A> pure(A a);

    // m a -> (a -> m b) -> m b
    <A, B> App<M, B> flatMap(
            App<M, A> ma,
             // a -> m b
            Function<A, App<M, B>> aToMb
    );
}
```

Пример IdMonad

Пример реализации тривиальной монады IdMonad:

```
public enum IdMonad implements Monad<IdK.mu> {
  INSTANCE;
  @Override public <A> IdK<A> pure(A a) { return IdK.of(a); }

  @Override public <A, B> IdK<B> flatMap(
      App<IdK.mu, A> ma,
      Function<A, App<IdK.mu, B>> aToMb
  ) {
    App<IdK.mu, B> result = aToMb.apply(narrow(ma).getValue());
    return narrow(result);
  }
}
```

Пример использования Monad

Пример на Java сопровождается эквивалентным кодом на Haskell для сравнения.

Java

```
<M> App<M, String> getGreet(Monad<M> m) {
 return m.flatMap(m.pure("Hello"),
  hello -> m.flatMap(m.pure("World"),
   world -> m.pure(hello + ", " + world + "!")
 ));
}
```

Haskell

```
greet = pure "Hello" >>=
 (\hello -> pure "World" >>=
  \world -> pure hello + ", " + world + "!"
 )
```

В этой задаче в произвольной монаде создается строка “Hello, World!”. Цветом выделены области действия лямбда-функций

Вопросы

(по функторам и монадам)

Монадические парсер комбинаторы

Монадические парсер комбинаторы

- Монадические парсер комбинаторы
- Структура парсера
- Определение парсера
- Вызов парсера
- Монада ParserMonad. Метод pure
- Монада ParserMonad. Метод flatMap
- Интерфейс MonadPlus
- Парсер комбинатор anyChar
- Парсер комбинатор chr
- Парсер комбинатор digit
- Парсер комбинатор opt
- Парсер комбинаторы many, many1
- Парсер комбинатор manyTill

Монадические парсер комбинаторы

Одна из задач, которая простым способом решается при помощи монад - это парсинг текста. Определяется набор примитивных парсеров, комбинация которых в свою очередь позволяет создавать более сложные парсеры.

Структура парсера

Структура парсера представляет собой функцию следующей сигнатуры:

```
String -> Optional<Result<T>>
```

, где

- На вход подается строка, которую нужно распарсить
- Результатом является тип “сумма” `Optional` с компонентами:
  - `Some`, если парсер может разобрать строку
  - `None`, если парсер не может разобрать строку
- Внутри `Optional` лежит объект `Result<T>`, который является типом “произведение” с компонентами:
  - `T` - значение, которое удалось разобрать
  - `String` - часть оригинальной строки, которая осталась после текущего парсера

Определение парсера

В java коде парсер представляется в следующей форме:

```
// String -> Optional<Result<T>>
interface Parser<T>
  extends Function<
    String,
    Optional<Result<T>>
  > {}
```

Вызов парсера

Парсер является функцией, поэтому парсер можно вызывать как обычную функцию, на вход которой нужно передать текст для разбора. Результатом функции будет тип `Optional<Result<T>>`, где:

- `None` - разбор не удался, прекратить парсинг.
- `Some<Result<T>>` - разбор оказался успешным, и результат записан в поле `Result#value`

В классе `ParserK` определен метод `parse`, который является фасадом для `Function#apply`.

Монада ParserMonad. Метод pure

Семантика метод `pure` заключается в том, что в качестве результата в `Result` вкладывается переданное в `pure` значение, и сама входная строка передается в `Result` без изменения:

```
enum ParserMonad implements MonadPlus<ParserK.mu> {
  INSTANCE;
  @Override public <A> ParserK<A> pure(A a) {
    return ParserK.of(s -> some(Result.of(a, s)));
  }
  ...
```

Монада ParserMonad. Метод flatMap

Семантика метода `flatMap` заключается в том, что комбинирует парсеры: сначала применить парсер первый, и, если разбор текста произошел успешно, применить следующий парсер:

```
// Parser<A> -> (A -> Parser<B>) -> Parser<B>
public <A, B> ParserK<B> flatMap(App<ParserK.mu, A>maK,
                                 Function<A, App<ParserK.mu, B>>aToMb) {
  Parser<A> sToAS = narrow(maK).getDelegate(); // первый парсер
  Parser<B> sToBS = s -> sToAS.apply(s).caseOf(
    () -> none(), // первый парсер потерпел неудачу, прекратить парсинг
    // первый парсер успешно отработал, запустить второй парсер
    result -> aToMb.andThen(ParserK::narrow).andThen(ParserK::getDelegate)
                   .apply(result.getValue()).apply(result.getRest()));
    return ParserK.of(sToBS);
}
```

Интерфейс MonadPlus

`MonadPlus` наследует интерфейс `Monad` и добавляет к нему метод `plus`. Семантика `plus` заключается в том, что он позволяет объединять два парсера в один:

- если первый парсер успешно разобрал строку, то его результат является результатом объединения парсеров
- если первый парсер не смог разобрать строку, то результат второго парсера является результатом объединения парсеров

Типичная задача, которая решается объединением парсеров будет такая: первая буква в строке должна быть либо “A”, либо “B”

Парсер комбинатор anyChar

Комбинатор `anyChar` позволяет получить из строки первый символ, если строка не пустая:

```
public static ParserK<Character> anyChar() {
  return of(s -> {
    if (s.isEmpty()) return none();
    return some(Result.of(s.charAt(0), s.substring(1)));
  });
}
```

Парсер комбинатор chr

Парсер комбинатор `chr` проверяет начинается ли строка с определенного символа, если да, то результатом будет этот символ:

```
public static ParserK<Character> chr(char c) {
  return of(s -> {
    if (s.isEmpty() || s.charAt(0) != c) return none();
    return some(Result.of(s.charAt(0), s.substring(1)));
  });
}
```

Парсер комбинатор digit

Парсер комбинатор `digit` при помощи комбинатора `chr` проверяет начинается ли строка с символа, который является цифрой:

```
public static ParserK<Character> digit() {
  ParserK.ParserMonad m = ParserK.ParserMonad.INSTANCE;
  return m.plus(chr('0'), m.plus(chr('1'),
         m.plus(chr('2'), m.plus(chr('3'),
         m.plus(chr('4'), m.plus(chr('5'),
         m.plus(chr('6'), m.plus(chr('7'),
         m.plus(chr('8'), chr('9')))))))));
}
```

Парсер комбинатор opt

Парсер комбинатор `opt` является мета парсером. Особенность заключается в том, что его результатом является `Result<Optional<T>>`. Парсер комбинатор `opt` применяет к входной строке другой парсер, если тот другой парсер смог разобрать строку, то его результат заворачивается в `Some` и этот `Some` становится результатом парсера `opt`. Если другой парсер не смог разобрать строку, то парсинг не прерывается как обычно, а результатом парсера (а не всего разбора) `opt` является `None`.

```
// (String -> Optional<Result<T>>) ->
//   (String -> Optional<Result<Optional<T>>>)
static <T> ParserK<Optional<T>> opt(ParserK<T> parser)
```

Парсер комбинаторы many, many1

Парсер комбинаторы `many` и `many1` являются мета парсерами, они применяют другой парсер к входной строке столько раз сколько возможно. Все результаты успешного применения накапливаются в списке. Их отличие заключается в том, что `many1` ожидает как минимум одно успешное применение парсера, тогда как `many` возвращает пустой список, если ни одного успешного применения парсера не произошло.

```
// (String -> Optional<Result<T>>) ->
//   (String -> Optional<Result<List<T>>>)
static <T> ParserK<List<T>> many(ParserK<T> parser)
static <T> ParserK<List<T>> many1(ParserK<T> parser)
```

Парсер комбинатор manyTill

Парсер комбинатор `manyTill` является мета парсером, он принимает на вход два парсера:

- Парсер, который нужно применять до тех пор, пока парсер остановки не сработает
- Парсер остановки

Результат работы парсера накапливается в списке. Парсер должен остановиться только с помощью парсера остановки, иначе весь парсинг признается неудачным.

```
// (String -> Optional<Result<T>>) -> (String -> Optional<Result<T>>) ->
//   (String -> Optional<Result<List<T>>>)
static <T> ParserK<List<T>> manyTill(ParserK<T> parser, ParserK<T> stop)
```

Вопросы

(по монадическим парсер комбинаторам)

JSON парсер

на основе монадических парсер комбинаторов

Возможные значения

Для возможных значений json документа определяется тип “сумма” Value

```
Value = String | Number | Null | Bool | Object a | Array (List Value)
```

, компоненты которого:

- `String` - строковый литерал
- `Number` - число
- `Null` - null значение
- `Bool` - логическое значение
- `Object` - объект
- `Array` - массив

Парсер комбинаторы

Для каждого компонента из типа `Value` определяется свой парсер

- `whitespace` - парсер пробельных символов
- `number` - парсер чисел как целых так и с плавающей запятой
- `string` - парсер строковых литералов
- `nullValue` - парсер значения `null`
- `fls` - парсер логического значения `false`
- `tru` - парсер логического значения `true`
- `array` - парсер гетерогенных массивов значениями, которого могут быть любые допустимые json значения
- `object` - парсер json объектов

Парсер object

Так как структура json объектов может быть самая разнообразная, поэтому в рамках данного доклада принято решение парсить объекты как список значений `Record`:

```
class Record {
        final String name;
        final Value value;
}
```

Список таких объектов напоминает `Map`, где ключом является `String`, а значением `Value` - любое json значение.

Если структура объекта известна, то можно определить парсер с определенной структурой, результатом которого будет такой объект

Парсер value

Парсер `value` является объединением всех определенных json парсеров.

```
public static ParserK<Value> value() {
  ParserK.ParserMonad m = ParserK.ParserMonad.INSTANCE;
  return m.plus(m.map(e -> e,
    string()), m.plus(m.map(e -> e,
    number()), m.plus(m.map(e -> e,
    object()), m.plus(m.map(e -> e,
    array()), m.plus(m.map(e -> e,
    nullValue()), m.plus(m.map(e -> e,
    tru()), m.map(e -> e,
    fls()
  )))))));
}
```

Применение json парсера

Использование парсера выглядит следующим образом.

```
{
  "greet": "hello",
  "what": ["world", "universe"],
  "now": true
}
```

Вызов парсера:

```
Optional<Parser.Result<T>> result = JsonParser.value().parse(json);
```

Результат:

```
Some(([{greet = hello};{what = [world;universe;]};{now = true};], ))
```

Вопросы

(по JSON парсеру)

Заключение

В текущем докладе был рассмотрен функциональный подход к разработке и проектированию программ на Java 8, напоминающий своей структурой программы на Haskell.

В докладе были рассмотрены следующие концепции:

- Алгебраические структуры данных и pattern matching
- Полиморфизм на уровне кайндов и типы высшего порядка
- Функторы и монады

В качестве применимости идей функционального программирования к задачам реального мира был реализован JSON парсер, структура которого выглядит декларативной.

Источники

- Jeremy Yallop, Leo White - Lightweight higher-kinded polymorphism - статья, описывающая идею легковесного HKT
- Arrow - библиотека общего назначения с функциональными структурами для языка Kotlin https://github.com/arrow-kt/arrow
- DataFixerUpper - библиотека, используемая в minecraft для миграции данных между версиями https://github.com/Mojang/DataFixerUpper
- avaj - библиотека для java с функциональными структурами для языка Java. В чисто функциональном стиле реализованы корутины https://github.com/neshkeev/avaj
- Функциональное программирование на Haskell (часть 2) - парсер-комбинаторы в Haskell https://stepik.org/course/693/

Исходный код

Спасибо за внимание

Вопросы