МУЛЬТИПАРАДИГМЕННЕ ПРОГРАМУВАННЯ

Лекція 10

Мова ймовірнісного програмування Figaro

АТОМАРНІ ЕЛЕМЕНТИ

Атомарні елементи поділяються на дискретні і безперервні, в залежності від типу значення. Значення дискретних елементів мають такі типи, як boolean або Integer, а безперервних - тип Double. Технічно дискретність означає, що значення досить далеко відстоять один від одного. Наприклад, між цілими числами 1 і 2 немає інших цілих чисел, тому можна сказати, що вони відокремлені одне від одного. З іншого боку, значення безперервних елементів утворюють нерозривний континуум, як, наприклад, дійсні числа. Між будь-якими двома числами знайдеться ще одне дійсне число.

Дискретні атомарні елементи

Розглянемо приклади дискретних атомарних елеметпов: Flip, Select і Binomial.

Елемент Flip

Ми вже зустрічалися з дискретним атомарним елементом Flip. Його код знаходиться в пакеті `com.cra.figaro.language` разом з багатьма іншими широковживаними елементами. Ми рекомендує імпортувати цей пакет цілком на початку програми. Елемент Flip приймає один аргумент `p` - ймовірність того, що значення елемента дорівнює `true`; `p` повинно бути числом від 0 до 1 включно.

Імовірність того, що елемент приймає значення **`false`**, дорівнює **`1 - p`**. Наприклад:

```
import com.cra.figaro.language._
val sunnyToday = Flip(0.2)
println(VariableElimination.probability(sunnyToday, true))
// печатается 0.2

println(VariableElimination.probability(sunnyToday, false))
// печатается 0.8
```

Елемент **`Flip (0. 2)`** має офіційний тип **`AtomicFlip`** - підклас класу **`Element [Boolean]`**. Цим він відрізняється від елемента **`CompoundFlip`**, який буде розглянуто надалі.

Елемент Select

З елементом select ми теж зустрічалися в програмі **"Hello World"**.

Ось приклад:
```
select (0. 6 -> "Здрастуй, світ!",
  0. 3 -> "Здрастуй, всесвіт!",
  0. 1 -> "О ні, тільки не це").
```

На малюнку нижче показано, як будується цей елемент. Усередині дужок знаходиться кілька частин. Кожна частина складається з ймовірності, стрілки і можливого результату. Кількість частин довільно.

На малюнку ми бачимо три частини. Оскільки всі результати мають тип **`string`**, то елемент належить класу **`Element [String]`**. Офіційно його типом є **`AtomicSelect [String]`** - підклас **`Element [String]`**.

Природно, елемент **`Select`** відповідає процесу, можливі результати якого вибираються з зазначеними ймовірностями. Ось як це буде працювати для прикладу на малюнку:

```
val greeting = Select(0.6 -> "Здравствуй, мир!",
  0.3 -> "Здравствуй, вселенная!", 0.1 -> "О нет, только не это")
println(VariableElimination.probability(greeting,
  "Здравствуй, вселенная!"))
// печатается 0.30000000000000004
```

Відзначимо, що сума ймовірностей в **`Select`** не обов'язково дорівнює 1. Якщо вона відмінна від 1, то ймовірності нормуються, тобто множаться на деякий (один і той же) коефіцієнт, так щоб вийшла сума 1.

У наступному прикладі ймовірності вдвічі більше, ніж в попередньому, тому їх сума дорівнює 2. Нормування призводить до таких же ймовірностей, як і вище, тому результати будуть збігатися.

```
val greeting = Select(1.2 -> "Здравствуй, мир!",
  0.6 -> "Здравствуй, вселенная!", 0.2 -> "О нет, только не это")
println(VariableElimination.probability(greeting,
  "Здравствуй, вселенная!"))
// печатается 0.30000000000000004
```

Елемент Binomial

`Binomial` - ще один корисний дискретний елемент. З кожним з семи днів тижня може бути пов'язаний елемент **`Flip (0.2)`**, що визначає чи буде в цей день ясно. А нам потрібен елемент, значенням якого є число ясних днів на тижні. Для цього можна взяти елемент **`Binomial (7, 0.2)`**. Його значенням є кількість випробувань, результат яких дорівнює **`true`**, при тому, що всього проводиться сім випробувань, і кожне дає **`true`** з ймовірністю **`0.2`**. Використовується цей елемент так:

```
import com.cra.figaro.library.atomic.discrete.Binomial
val numSunnyDaysInWeek = Binomial(7, 0.2)
println(VariableElimination.probability(numSunnyDaysInWeek, 3))
//печатается 0.114688
```

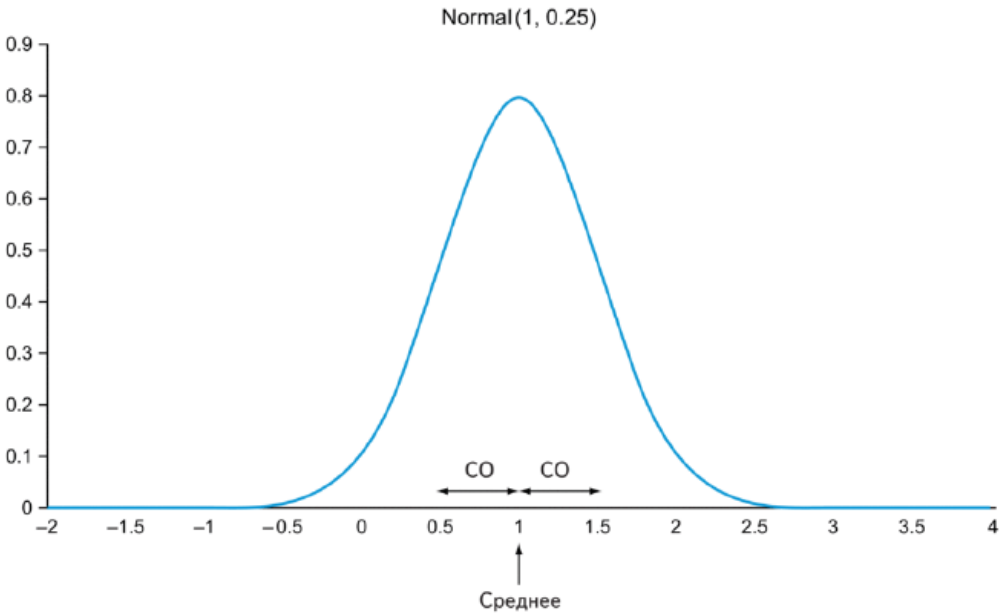
Елемент **`Binomial`** приймає два аргументи: кількість випробувань і ймовірність того, що випробування дасть результат **`true`**. У визначенні **`Binomial`** передбачається, що випробування незалежні; від того, що перше дало **`true`**, ймовірність отримання **`true`** в другому не змінюється.

БЕЗПЕРЕРВНІ АТОМАРНІ ЕЛЕМЕНТИ

У цьому розділі описані два поширених безперервних елемента: **`Normal`** і **`Uniform`**. Далі безперервні елементи розглядаються детально. Безперервне розподіл ймовірності відрізняється від дискретного тим, що задається не ймовірність кожного значення, а щільність ймовірності, що описує вірогідність інтервалу навколо значення. Проте, можна вважати, що щільність ймовірності схожа на звичайну ймовірність, тобто показує, наскільки дане значення ймовірно в порівнянні з іншими.

Елемент Normal

Ви, напевно, знайомі з нормальним розподілом ймовірності. У нього є і інші назви, наприклад: дзвіновидна крива або гауссовий розподіл.


Normal(1, 0.25)
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
–2
–1.5
–1
–0.5
0
0.5
1
1.5
2
2.5
3
3.5
4
CO
CO
Среднее

У цій функції є середнє значення - центральна точка (1 на малюнку) - і стандартне відхилення, тобто ступінь розкиду функції навколо центральної точки (0.5 на малюнку).
Приблизно в 68% відсотках випадків значення з нормальним розподілом буде відстояти від середнього не далі, ніж на одне стандартне відхилення. У статистиці і в імовірнісних міркуваннях нормальний розподіл зазвичай задається середнім і дисперсією, яка дорівнює квадрату стандартного відхилення. Оскільки на малюнку нижче стандартне відхилення дорівнює 0.5, то дисперсія дорівнює 0.25. Отже, це нормальний розподіл задається як **`Normal (1, 0.25)`**.

Figaro точно следует этому соглашению. Он предоставляет элемент `Normal`, принимающий в качестве аргументов среднее и дисперсию, например:

```
import com.cra.figaro.library.atomic.continuous.Normal
val temperature = Normal(40, 100)
```

Здесь среднее равно 40, а дисперсия 100, т. е. стандартное отклонение равно 10. Предположим теперь, что мы хотим выполнить вывод с помощью этого элемента. Увы, алгоритм исключения переменных в Figaro работает только для элементов, принимающих конечное число значений, а, значит, к непрерывным элементам он неприменим. Поэтому придется взять другой алгоритм. Мы воспользуемся приближенным алгоритмом *выборки по значимости*, который хорошо работает с непрерывными элементами. Выполнить этот алгоритм можно так:

```
import com.cra.figaro.algorithm.sampling.Importance

def greaterThan50(d: Double) = d > 50
  println(Importance.probability(temperature, greaterThan50 _))
```

Выборка по значимости – это рандомизированный алгоритм, который каждый раз дает новый ответ, близкий к истинному значению – в данном случае 0.1567.

Отметим, что запрос немного отличается от того, что мы видели раньше. Вероятность, что значение непрерывного элемента будет в точности равно заданному, например 50, очень близка к 0, поскольку значений бесконечно много, и промежутков между ними нет. Шансы на то, что процесс вернет значение, равное точно 50, а не, скажем, 50.000000000000001, бесконечно малы. Поэтому мы обычно не запрашиваем у непрерывного элемента вероятность точного значения.

Вместо этого указывается диапазон вероятностей. В данном случае в запросе задан предикат `greaterThan50`, который принимает в качестве аргумента значение типа `Double` и возвращает `true`, если аргумент больше 50. *Предикатом* называется булева функция от значения элемента. Спрашивая, удовлетворяет ли элемент предикату, мы на самом деле интересуемся, с какой вероятностью применение предиката к значению элемента вернет `true`. В этом примере запрос вычисляет вероятность того, что температура больше 50.

Елемент Uniform

Розглянемо ще елемент **`uniform`** (рівномірний розподіл), який повертає значення в зазначеному діапазоні з однаковою ймовірністю:

```
import com.cra.figaro.library.atomic.continuous.Uniform

val temperature = Uniform(10, 70)
Importance.probability(temperature, greaterThan50 _)
// печатается число, близкое к 0.3334
```

Елемент **`uniform`** приймає два аргументи: початок і кінець діапазону. Щільність ймовірності всіх значень в цьому діапазоні однакова. В наведеному вище прикладі мінімальне значення дорівнює 10, а максимальне 70, так що довжина діапазону дорівнює 60. Предикат в запиті перевіряє, чи знаходиться значення між 50 і 70; довжина цього проміжку дорівнює 20. Отже, для цього предиката ймовірність дорівнює 20/60, або 1/3, і легко бачити, що алгоритм вибірки за значимістю дає близький результат.

І наостанок одне зауваження: офіційно цей елемент називається безперервний **`uniform`**. Існує також дискретний **`uniform`**, що знаходиться в пакеті:
`com.cra.figaro.library.atomic.discrete`.
Як легко здогадатися, дискретний uniform повертає кожне значення з переданого списку з однаковою ймовірністю.

Отже, ми познайомилися з будівельними блоками, тепер подивимося, як з них збираються більші моделі.

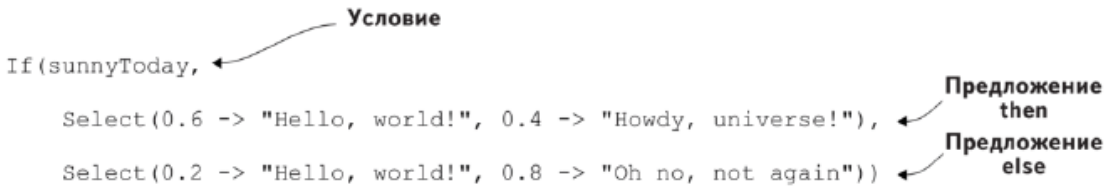
СОСТАВНІ ЕЛЕМЕНТИ

Приступимо до розгляду декількох складових елементів. Нагадаємо, що складовий елемент будує з кількох елементів можливо більш складних. Складових елементів багато, але для початку ми розглянемо тільки два: **`If`** і **`Dist`**, а потім подивимося, як використовувати складові версії більшості атомарних елементів.

Составний елемент If

Ми вже зустрічалися з одним прикладом складового елементу - **`If`**. Він складається з трьох елементів: умови, пропозиції **`then`** і пропозиції **`else`**. Випадковий процес, представлений елементом **`If`**, з початку перевіряє умова. якщо результат обчислення умови дорівнює **`true`**, то процес породжує значення пропозиції **`then`**, інакше значення пропозиції **`else`**.

На малюнку нижче показаний приклад елемента **`If`**. Як бачимо, **`If`** приймає три аргументи. Перший аргумент - що представляє
умова - має тип **`Element [Boolean]`** і в даному випадку є елементом sunnyToday. Другий аргумент - пропозиція **`then`**; якщо значення елемента-умови одно **`true`**, то вибирається цей елемент. Якщо ж значення елемента-умови одно **`false`**, то вибирається третій аргумент - пропозиція **`else`**. Типи значень пропозицій **`then`** і **`else`** повинні збігатися, і цей загальний тип є типом значення **`If`**.



Нижче показаний елемент **`If`** в дії:

```
val sunnyToday = Flip(0.2)
val greetingToday = If(sunnyToday,
    Select(0.6 -> "Здравствуй, мир!", 0.4 -> "Здравствуй, вселенная!"),
    Select(0.2 -> "Здравствуй, мир!", 0.8 -> "О нет, только не это"))
println(VariableElimination.probability(greetingToday, "Здравствуй, мир!"))
// печатается 0.27999999999999997
```

Пояснимо, чому друкується 0.28 (з деякою погрішністю). Пропозиція **then** вибирається з ймовірністю 0.2 (коли **sunnyToday** дорівнює **true**), і в цьому випадку рядок **<< Здрастуй, світ! >>** вибирається з ймовірністю 0.6.

З іншого боку, пропозиція **else** вибирається з ймовірністю 0.8, і в цьому випадку рядок **<< Здрастуй, світ! >>** вибирається з ймовірністю 0.2. Отже, повна ймовірність вибору рядка **<< Здрастуй, світ! >>** дорівнює:

(0.2 x 0.6) + (0.8 x 0.2) = 0.28.

У цьому можна переконатися, явно обчисливши два випадки:

```
sunnyToday.observe(true)
println(VariableElimination.probability(greetingToday, "Здравствуй, мир!"))
// печатается 0.6, потому что всегда выбирается предложение then
sunnyToday.observe(false)
println(VariableElimination.probability(greetingToday, "Здравствуй, мир!"))
// печатается 0.2, потому что всегда выбирается предложение else
```

Складний елемент Dist

Dist - ще один корисний складовий елемент. Він схожий на select, але вибір проводиться не з множини значень, а з множини елементів. Оскільки кожен варіант вибору сам є елементом, **Dist** виявляється складовим елементом.

Елемент **Dist** корисний, коли потрібно вибрати один з випадкових процесів. Наприклад, кутовий удар у футболі може бути коротким пасом або відразу навісом на ворота. Це можна уявити елементом **Dist**, які вибирають один з двох процесів: короткий пас і навіс на ворота.

Елемент **Dist** знаходиться в пакеті **com.cra.figaro.language**. Нижче наведено приклад його використання:

```
val goodMood = Dist (0.2 -> Flip (0.6),
0.8 -> Flip (0.2))
```

За структурою (див.малюнок нижче) він схожий на елемент **`Select`**. Різниця в тому, що наслідками не є значення, а елементи. Можна уявляти собі це так: **`Select`** вибирає одне з множини значень безпосередньо, без проміжного процесу, а **`Dist`** опосередковано - спочатку вибирається процес, а той вже генерує значення в ході виконання. В даному випадку з імовірністю 0.2 буде обраний процес, представлений елементом **`Flip (0.6)`**, і з імовірністю 0.8 - процес, представлений елементом **`Flip (0.2)`**.

Ось що виходить в разі запиту до цього елементу:

```
println (VariableElimination.probability(goodMood, true))
// печатается 0.28 = 0.2 * 0.6 + 0.8 * 0.2
```

Складові версії атомарних елементів

Ми бачили приклади атомарних елементів, які приймають числові аргументи.

Наприклад, **`Flip`** приймає ймовірність, а **`Normal`** - середнє і дисперсію. А якщо ми не знаємо точно значення числових аргументів? У **`Figaro`** відповідь проста: *зробіть їх елементами*.

Коли числові аргументи є елементами, ми отримуємо складові версії вихідних атомарних елементів. Наприклад, нижче визначено складовою **`Flip`**, до якого пред'являється запит:

```
val sunnyTodayProbability = Uniform(0, 0.5)
val sunnyToday = Flip(sunnyTodayProbability)
println(Importance.probability(sunnyToday, true))
// печатается число, близкое к 0.2548
```

Тут елемент **sunnyTodayProbability** представляє невідому ймовірність ясної погоди сьогодні; ми вважаємо, що равновероятно будь-яке значення від 0 до 0.5.

Тоді **sunnyToday** одно **true** з ймовірністю, що дорівнює значенню елемента **sunnyTodayProbability**. У загальному випадку складовою **Flip** приймає один аргумент типу **Element [Double]**, що представляє ймовірність того, що **Flip** поверне **true**.

Елемент **Normal** надає більше можливостей. В імовірнісних міркуваннях часто припускають, що дисперсія нормального розподілу відома, а от щодо середнього повної впевненості немає. Так, можна припустити, що дисперсія температури дорівнює 100, а середнє приблизно дорівнює 40, але точне його значення невідомо. Цю ситуацію можна змоделювати таким чином:

```
val tempMean = Normal(40, 9)
val temperature = Normal(tempMean, 100)
println(Importance.probability(temperature, (d: Double) => d > 50))
// печатается число, близкое к 0.164
```

З іншого боку, може бути невідома дисперсія. Припустимо, у нас є підстави вважати, що вона дорівнює 80 або 105. Тоді можна написати такий код:

```
val tempMean = Normal(40, 9)
val tempVariance = Select(0.5 -> 80.0, 0.5 -> 105.0)
val temperature = Normal(tempMean, tempVariance)
println(Importance.probability(temperature, (d: Double) => d > 50)
// печатается число, близкое к 0.1549
```

ПОБУДОВА СКЛАДНИХ МОДЕЛЕЙ

У **Figaro** є два дуже корисних для побудови моделей елементів: **Apply** і **Chain**. **Apply** дозволяє виконувати з **Figaro** довільний код на **Scala**, надаючи тим самим всю міць мови **Scala** в розпорядження розробника. **Chain** дає можливість створювати нескінченно різноманітні залежності між елементами. Складові елементи **If** і **Flip**, які ми розглядали раніше, вміють створювати тільки залежності зумовленого виду. А **Chain** дозволяє створити будь-яку потрібну вам залежність.

Елемент Apply

Елемент Apply знаходиться в пакеті com.cra.figaro.language. Він приймає на вході довільний елемент і функцію Scala. Представлений ним процес застосовує функцію до значення елемента з метою отримання нового значення.

Наприклад:

```
val sunnyDaysInMonth = Binomial(30, 0.2)
def getQuality(i: Int): String =
  if (i > 10) "хорошее"; else if (i > 5) "среднее"; else "плохое"
val monthQuality = Apply(sunnyDaysInMonth, getQuality)
println(VariableElimination.probability(monthQuality, "хорошее"))
// печатается 0.025616255335326698
```

У другому і третьому рядках визначена функція **`getQuality`**. Вона приймає аргумент типу **`Integer`**, який всередині функції іменується **`i`**, і повертає рядок.

У четвертому рядку визначено елемент **`Apply`** з ім'ям **`monthQuality`**.

Структура елемента **Apply** показана на малюнку нижче. **Apply** приймає два аргументи. *Перший* є елементом, в даному випадку це елемент **sunnyDaysInMonth** типу **Element[Int]**.

Другий - функцією, аргумент якої повинен мати той же тип, що і тип значення елемента. У нашому прикладі функція **getQuality** приймає аргумент типу **Integer**, так що ця умова дотримується. Функція може повертати значення будь-якого типу, в нашому випадку вона повертає **string**.


Элемент, к которому применяется функция
Apply(sunnyDaysInMonth, getQuality)
Функция, применяемая к значениям элемента

Елемент **Apply** визначає випадковий процес наступним чином. Спочатку генерується значення елемента, переданого в першому аргументі. У нашому прикладі це число ясних днів у місяці. Припустимо, що отримано значення 7. Потім процес застосовує до цього значення функцію, передану в другому аргументі. У нашому випадку викликається функція **getQuality** з аргументом 7, яка повертає рядок середнього. Цей рядок і стає значенням елемента **Apply**. Таким чином, значення, які повертаються **Apply**, мають той же тип, що повертається значення функції. У нашому випадку елемент **Apply** належить типу **Element[String]**.

Є багато практичних причин для застосування **`Apply`**. Назвемо лише деякі з них.

- Є елемент типу **`double`**, значення якого потрібно округлити до найближчого цілого.
- Є елемент, значення якого має тип структури даних, і потрібно якось агрегувати деякий властивість цієї структури. Наприклад, елемент визначено над списками, і ми хочемо взнати ймовірність того, що кількість елементів в списку більше 10.
- Зв'язок між двома елементами найкраще представити у вигляді фізичної моделі. У такому випадку ми можемо

скористатися функцією **Scala** для подання фізичною зв'язку і включити цю модель в **Figaro** за допомогою **Apply**.

Apply з декількома аргументами

Елемент **`Apply`** приймає також функції **`Scala`** з декількома аргументами, але не більше п'яти. Таке застосування **`Apply`** корисно, коли для впливу на певний елемент потрібно більше одного елемента. Наведемо приклад.

```
val teamWinsInMonth = Binomial(5, 0.4)
val monthQuality = Apply(sunnyDaysInMonth, teamWinsInMonth,
  (days: Int, wins: Int) => {
    val x = days * wins
    if (x > 20) "хорошее"; else if (x > 10) "среднее"; else "плохое"
  })
```

Тут **Apply** передаються в якості аргументів два елементи: **sunnyDaysInMonth** і **teamWinsInMonth**, обидва вони мають тип **Element [Int]**.

Передана в третьому аргументі функція приймає два аргументи типу **integer**, названі **days** та **wins**, і створює локальну змінну **x** зі значенням **days * wins**.

Відзначимо, що оскільки **days** та **wins** - звичайні змінні

Scala типу **integer**, то **x** також звичайна змінна, а не елемент **Figaro**. Насправді, все всередині переданої **Apply** функції - звичайний код на **Scala**. Елемент **Apply** приймає цю звичайну функцію **Scala**, що працює зі звичайними значеннями **Scala**, і "зводить" її в ранг функції, що працює з елементами **Figaro**.

Тепер висунемо запит цієї версії **monthQuality**:

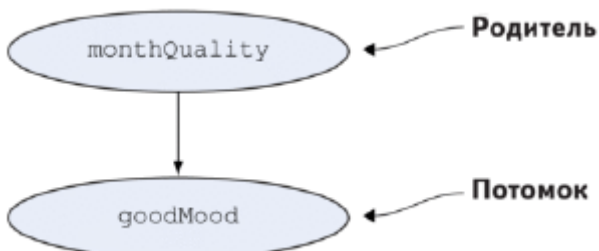
```
println(VariableElimination.probability(monthQuality, "хорошее"))
// печатается 0.15100056576418375
```

Елемент Chain

Як випливає з назви, елемент **`Chain`** служить для зчеплення елементів в модель, де один елемент залежить від іншого, який, в свою чергу залежить від інших, і так далі.

Елемент **`Chain`** також знаходиться в пакеті **`com.cra.figaro.language`**. Найпростіше пояснити принцип його роботи за допомогою картинки, де зображені два елементи; **`goodmood`** залежить від **`monthQuality`**. Відповідний випадковий процес спочатку генерує значення **`monthQuality`**, а потім на його основі генерує значення **`goodmood`**. Це простий приклад байєсівської мережі.

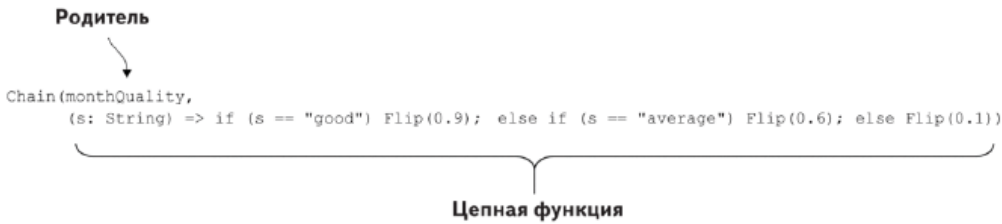
Запозичуючи термінологію байєсівських мереж, елемент **monthQuality** на малюнку називається батьком, а елемент **goodmood** — нащадком.



Оскільки **goodMood** залежить від **monthQuality**, то **goodMood** визначено за допомогою **Chain**. Елемент **monthQuality** вже був визначений в попередньому розділі. А ось як виглядає визначення **goodMood**:

```
val goodMood = Chain(monthQuality, (s: String) =>
  if (s == "good") Flip(0.9)
  else if (s == "average") Flip(0.6)
  else Flip(0.1))
```

На малюнку нижче показана структура цього елемента. Як і **Apply**, елемент **chain** приймає два аргументи: елемент і функцію. В даному випадку елементом є батько, а в ролі функції виступає так звана ланцюгова функція. Різниця між **Chain** і **Apply** полягає в тому, що передана функція в разі **Apply** повертає звичайні значення **Scala**, а в разі **Chain** - елемент. У нашому прикладі функція повертає елемент **Flip**, а який саме, залежить від значення **monthQuality**. Таким чином, ця функція приймає аргумент типу **String** і повертає екземпляр класу **Element [Boolean]**.



Випадковий процес, визначений цим елементом Chain, показаний на наступному слайді Він складається з трьох кроків. По-перше, генерується значення батька. У нас це значення середнє елемента monthQuality. По-друге, до цього значення застосовується ланцюгова функція, яка повертає результуючий елемент. В даному випадку з визначення ланцюгової функції видно, що це елемент Flip (0.6).

По-третє, результуючий елемент генерує значення - в нашому випадку true. Це значення і стає значенням нащадка.

Підіб'ємо підсумок, перерахувавши типи всіх компонентів, що беруть участь в роботі Chain. Елемент Chain параметризований двома типами: типом значення батька (позначимо його T) і типом значення нащадка (позначимо його U):

- Батько має тип Element [T].
- Значення батька має тип T.
- Ланцюгова функція має тип T => Element [U]. Це озіачает, що фуікція отримує аргумент типу T і повертає значення типу Element [U].
- Результуючий елемент має тип Element [U].
- Значення ланцюжка має тип U.
- Нащадок має тип Element [U]. Це і є тип самого елемента Chain.

В наступній лекції ми розглянемо принципи розробки застосувань в мультипарадигменій мові ймовірнісного програмування R.