**Уважаемый читатель!**

Все переводные материалы, представленные в каталоге **«Переводы»**, выложены в свободный доступ для свободного использования в образовательных и научных целях. Я, как переводчик, рад, что мои труды не пропадают даром и используются хорошими людьми. Однако я был бы ещё больше рад, если бы хорошие люди благодарили меня за мой непростой труд. Благодарность можно выразить при помощи переводов на следующие счета в различных платёжных системах:

- Яндекс.Деньги: **4100137733052**.
- WebMoney: **R211895623295**.
- PayPal: **darkus.14@gmail.com**.

Ваша благодарность поможет мне и дальше заниматься переводами и выкладывать их в свободный доступ.

---

*Перевод цикла записей в блоге Криса Смита: Введение, 1, 2, 3, 4, 5, 6, 7, 8.*

# Язык Haskell для детей

*Смит К., Шпенглер С.*

## Введение

В течение нескольких недель я собираюсь еженедельно публиковать запись под общим названием темы «Язык Haskell для детей». Данный раздел является введением с описанием того, что это вообще такое.

### Что это такое и для чего всё это нужно?

В этом учебном году у меня появилась прекрасная возможность обучить языку Haskell группу детей 12 —13 лет. Я крайне рад этому. Я обучаю детей в качестве учителя в Little School в Vermijo по соседству с моей соседкой Сью. Мне нравится эта идея, которая заключается в том, что школа располагается непосредственно в жилом доме — в классах немного детей, а люди в округе постоянно вносят свою долю в учебный процесс. В прошлом году я раз в неделю обучал детей математике, и это было очень здорово. В этом году я пытаюсь преподавать программирование в объёме двух часов в неделю.

### Что, реально язык Haskell?

Да, я выбрал язык Haskell в качестве языка программирования для обучения. Это, конечно, спорно, но у меня есть веские причины:

- Это то, что меня очень волнует! Мне кажется, что это лежит глубже каких-либо технических деталей, поскольку каждому важно учить других тому, что его волнует, и

передавать свою взволнованность другим.

- Бен Липпмейер (Ben Lippmeier) написал отличную библиотеку Gloss, которая является прекрасным инструментом для обучения. Эта библиотека позволяет создавать графические приложения при помощи высокоуровневого, компонуемого и семантически осмысленного API.
- Мне кажется, что язык Haskell является наилучшим средством для изучения и практического применения абстрактных шаблонов мышления, которые будут полезны в математике, логике и чётком понимании окружающего мира. Смотрите, меня не волнует то, что кто-либо после моих уроков выберет карьеру программиста. Если кто-нибудь из учеников станет поэтом, я буду надеяться, что мои уроки тоже сыграли в этом деле не последнюю роль.

Когда я обсуждал свои планы с другими людьми, я слышал возражения о том, что дескать язык Haskell «сложен». Мне кажется, что это слишком сильное упрощение. Если внимательно посмотреть на причины, по которым язык Haskell считается «сложным», то большинство их сводится к: аз) он отличается от других языков программирования, которые используют люди; буки) при помощи него люди делают сложные вещи. Ни одна из этих причин не применима к данному курсу обучения.

Но это совсем не означает, что всякий должен выбирать для себя язык Haskell. Если вы собираетесь обучать детей программированию, то необходимо следить за тем, чтобы избегать *случайных* сложностей. Подходит любой язык программирования, который позволяет вам выражать то, что вы хотите донести. Если бы я не использовал для этих целей язык Haskell, я бы использовал язык Python. Но я принимаю во внимание свою взволнованность, так что выбор пал на Haskell!

## Зачем публиковать это всё в блоге?

Обучение будет проводиться по утрам вторника и среды каждой недели. В процессе планирования курса я столкнулся с серьёзной проблемой — в природе нет книг, повествующих об обучении языку Haskell двенадцатилетних детей. Так что вместо учебника предлагаю свой собственный план:

1. Конечно, без специальных приготовлений никуда. Я сделал наброски нескольких десятков тем, наметил зависимости между ними, придумал идеи для упражнений в Gloss для каждой темы. Я бы не хотел выносить это на всеобщее обсуждение, поскольку я бы не хотел, чтобы сроки или программа обучения была зафиксирована ещё до начала.
2. Каждую неделю сразу после проведения уроков я буду структурировать и обобщать полученный опыт, записывая всё это дело в своё блоге под меткой «Язык Haskell для детей».
3. Своим ученикам я поручу приходить читать эти записи и оставлять под ними комментарии относительно того, чему интересному они научились, какие новые идеи пришли им в голову, и что они могли бы сделать со своими программами, которые казались прикольными.

Я надеюсь, что при помощи такой структуры курса я достигну ещё нескольких целей.

Первой и наименее интересной целью является создание письменного источника для всех учеников, чтобы у них была возможность в любое время прийти и обновить свои знания. Честно говоря, я надеюсь, что это применение записей будет не так уж часто и

востребовано. Если моим ученикам постоянно требуется ретроспекция, то это является прекрасным показателем того, что я слишком спешу в своём обучении. Тем не менее, наличие записей просто полезно ещё и потому, что они помогут тем ученикам, которые отсутствовали на уроках, либо просто ими можно будет пользоваться для повторного ознакомления с тем, что было пройдено ранее.

А второй, более важной целью я полагаю подключение своих учеников к большому сообществу. Всех, кто ещё читает этот блог, независимо от того, только ли учите вы язык Haskell или уже являетесь полноправным программистом, призываю присоединяться к обсуждениям. Я знаю, что по крайней мере три или четыре человека, с кем я разговаривал о планируемом курсе и кто хотел бы получать информацию о его продвижении, смогут получать эту информацию здесь. Это ваш шанс!

## Краткий обзор

Я организую процесс обучения на базе структуры библиотеки Gloss, которая в свою очередь основана на интересном прогрессе идей в программировании. Так что лучшим способом разъяснить структуру библиотеки является перечисление четырёх точек входа библиотеки Gloss. Вот они:

`displayInWindow`

Это простейшая точка входа библиотеки Gloss. Вы даёте ей описание изображение, и она просто выводит это изображение в окне. Мы будем использовать эту функцию для изучения базовых типов данных (чисел, булевых значений и т. п.), структурированных типов (например, списков и кортежей), выражений, приоритета и порядка операций, переменных и генераторов списков.

`animateInWindow`

Эта точка входа библиотеки Gloss позволяет отобразить в окне анимацию, представленную в виде функции, которая отображает время в картинку. Мы будем использовать её для обсуждения таких понятий, как определение функции, условное выражение, сопоставление с образцами и определение типов.

`simulateInWindow`

Третья точка входа библиотеки Gloss запускает симуляцию, которая отличается от простого анимирования наличием состояния. Эту точку входа мы будем использовать при определении и обсуждении своих собственных типов данных.

`gameInWindow`

Четвёртая и последняя точка входа библиотеки Gloss осуществляет симуляцию на основе данных пользователя, и это называется «игрой». Мы будем использовать эту точку входа для обсуждения проблем, возникающих в крупномасштабных программах: рекурсия в коде и данных, модульное программирование, простые структуры данных и создание сложных типов данных. На этом этапе я также ожидаю, что буду больше времени уделять практике, поощряя каждого ученика к созданию чего-либо собственного.

Точное распределение времени для каждой части курса будет определяться в течение

года, так что посмотрим, как это всё пойдёт. Если вам интересна эта тема, держитесь около!

# Неделя 1

## Погнали!

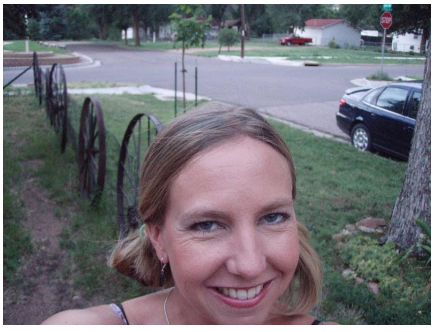
Ещё раз хотелось бы поблагодарить тех, кто поддержал этот проект и был его частью. Сегодня состоялся мой первый урок, посвящённый языку программирования Haskell, и это было замечательно! Это сообщение является моим первым еженедельным обзором того, что мы делаем на неделе.

## Сперва хотелось бы представиться

Поскольку мы учим нескольких детей, то начнём с краткого представления.

- **Я**: меня зовут Крис Смит, и я обучаю школьников программированию в Little School в Vermijo, где всё это началось. Я являюсь большим поклонником языка Haskell, и я действительно взволнован тем, что имею возможность поделиться своими знаниями с другими людьми.

- **Сью**: Сью Шпенглер является основателем, ведущим педагогом, руководителем, суперинтендантом, смотрителем, секретарём и поваром Little School в Vermijo. Школа является её личным проектом, в котором она вытворяет много прекрасных вещей. Мне пришлось покопаться в архивах, чтобы найти её фотографию, но мне кажется, что эта ей понравится.



- **Мои ученики**: сегодня у меня в классе присутствовали: Грант, София, Марчелло и Эви

(надеюсь, что я произносил их имена правильно). Я попросил их представиться в комментариях к этой записи, так что смотрите ниже.

- **Все остальные**: любых других детей, которые хотят участвовать в обучении, также прошу представиться в комментариях. Вы можете поздороваться, а если хотите, можете дать ссылку на фотографию или видео.

Я надеюсь, что у каждого найдётся чуточку времени, чтобы представиться и поздороваться с остальными. Процесс обучения всегда идёт веселее, когда можно общаться с другими людьми.

## План

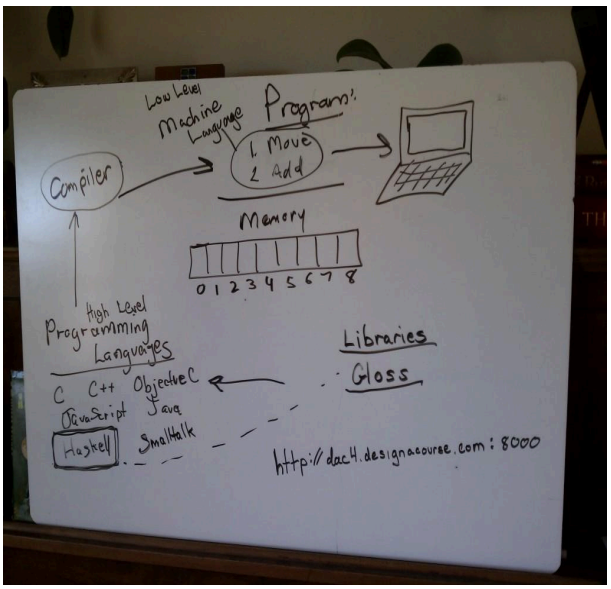
Мы говорили о том, куда направляемся. В частности:

1. Будем учиться создавать компьютерные программы, которые рисуют картинки.
2. Будем изменять наши компьютерные программы таким образом, чтобы картинки двигались!
3. Будем создавать игру на свой собственный выбор.

Эта программа займёт весь учебный год. И это не потому, что ученикам надо будет запоминать многие вещи относительно написания компьютерных программ, а потому, что мы будем творцами, будем создавать нечто, за что потом будем гордиться. Так что у нас будет много свободного времени для того, чтобы играть и пробовать различные идеи в наших программах. Мы будем учить язык программирования Haskell, однако в конце обучения ученики будут уметь управлять компьютером, проектировать и реализовывать что-нибудь действительно прикольное «с нуля», а не просто помнить всякую фигню про язык Haskell.

## Организация инфраструктуры и процесса программирования

Сперва мы поговорили о том, что такое компьютерная программа и как идеи совмещаются друг с другом. Вот снимок нашей доски после того, как мы закончили:




А вот некоторые из идей, о которых мы говорили:

- *Как работает компьютер*. Основная часть компьютера состоит из устройства, выполняющего инструкции («процессор»), и устройства, запоминающего информацию («память»).
- *Машинный язык*. Конечно, компьютер не разговаривает по-русски. Он выполняет инструкции, записанные на языке, называемом «машинным языком». Этот язык очень простой для компьютера, но очень сложный для человека, и писать на нём программы практически невозможно.
- *Компиляторы*. Вместо того, чтобы писать программы на машинном языке, мы пишем их на других языках, а потом используем компьютер для того, чтобы перевести их. Программа, которая это делает, называется компилятором.
- *Некоторые языки программирования*. У нас есть выбор: мы можем использовать различные языки программирования для написания программ. Мы немного помозговали на тему того, о каких языках ребята когда-либо слышали. Этими языками оказались: Java, C, C++, Objective C, JavaScript и Smalltalk (да, Марчелло слышал о языке Smalltalk, я под впечатлением). А язык, который мы будем изучать на этих занятиях, называется Haskell.
- *Библиотеки*. Библиотеками называются программы или части программ, которые какие-либо люди написали для нас, так что нам не придётся начинать с чистого листа. Мы затратили некоторое время на обсуждение шагов, которые надо сделать для того, чтобы заставить компьютер выполнить то, что нам хочется. Например, какие самые мельчайшие шаги надо совершить, чтобы нарисовать окно. Сколько кругов, прямоугольников, линий, букв и т. д. можно обнаружить в одном окне на компьютере? Библиотеки позволяют один раз описать какие-либо последовательности действий для того, чтобы каждый раз не делать этого.

После этого мы поговорили о том, как мы будем использовать язык программирования Haskell и библиотеку Gloss.

## Немного игры

На этом этапе мы все используем web-сайт для написания несложных компьютерных программ. Его адрес: http://dac4.designacourse.com:8000/.

Мы начали с действительно простых программ. Например, похожих на следующие.

Нарисовать круг:

```
import Graphics.Gloss
picture = circle 80
```

Нарисовать прямоугольник:

```
import Graphics.Gloss
picture = rectangleSolid 200 300
```

Нарисовать слова:

```
import Graphics.Gloss
picture = text "Hello"
```

Все эти программы обладают несколькими одинаковыми частями:

- Во всех программах первой строкой является «`import Graphics.Gloss`». Это говорит компилятору то, что для рисования картинок вы хотите использовать библиотеку Gloss. Вы должны написать эту строчку только один раз в самом начале своей программы.
- Далее во всех программах есть строка, начинающаяся с «`picture =`». Это обозначает то, что наша программа предназначена для рисования картинки, название которой будет «`picture`». После этого тот web-сайт, который мы используем для программирования, берёт эту картинку, какой бы мы её не определили, и рисует её на экране. Мы разговаривали о том, как в будущем мы сможем определять другие картинки, давая им другие названия, но на этом этапе мы просто согласились с тем, что вот так мы будем говорить компилятору о том, какую картинку мы хотим нарисовать.
- После символа (=) описываются сами картинки, которые мы хотим нарисовать. Есть несколько различных видов картинок, и мы только что увидели три из них. Все виды картинок, которые можно создать, включены в библиотеку Gloss.
- За исключением последнего примера, в первых двух используются некие числа. Например, число 80 в первом примере представляет собой радиус круга (то есть расстояние от его центра до границы). Вы можете сделать это число больше для того, чтобы нарисовать больший круг, или меньше, чтобы нарисовать меньший круг. То же самое вы можете сделать и с длиной и высотой прямоугольника.

Некоторые люди сообщили нам о каких-то проблемах с web-сайтом. Если у вас возникают проблему, то прошу убедиться, что вы используете последнюю версию браузера. И ещё... Web-сайт не работает с программой для скачивания браузеров (Internet Explorer), так что используйте её по прямому назначению и попробуйте что-нибудь из этого списка: Chrome, Firefox, Opera или Safari. Однако не беспокойтесь насчёт проблем с браузером, поскольку скоро вы установите себе на компьютер компилятор языка Haskell, так что вам больше не нужен будет никакой web-сайт, чтобы запускать свои программы. Мы использовали web-сайт только для того, чтобы быстро начать заниматься.

## Нарисуем несколько штучек

К этому моменту несколько учеников спросили, могут ли они рисовать несколько фигур на одной и той же картинке. Для этих целей вы можете использовать функцию `pictures` (обратите внимание на окончание). Например:

```
import Graphics.Gloss
picture = pictures [circle 80,
                    rectangleWire 200 100]
```

Рассмотрим внимательнее:

- Слово «`pictures`».
- Открывающая квадратная скобка.
- Список интересующих нас изображений, разделённых запятой.
- Закрывающая квадратная скобка.

Мы размышляли о том, как было бы полезно в наших программах иметь возможность писать на новых строках. Это можно, но единственно, что вам надо принимать во внимание, так это то, что каждая новая строка должна начинаться с нескольких пробелов. Видите, как третья строка в программе сдвинута от начала?

## Изменение ваших изображений

Библиотека Gloss имеет несколько инструментов для изменения ваших изображений.

При помощи функции `color` вы можете изменять цвет:

```
import Graphics.Gloss
picture = color red (circleSolid 80)
```

Обратите внимание, как мы вызываем функцию «`color`» тогда, когда хотим указать цвет, а потом записываем изображение в круглых скобках. Круглые скобки обозначают то же самое, что и в математике — трактовать то, что внутри них, в качестве единого объекта (в нашем примере, в качестве изображения).

Мы говорили о том, какие цвета знает библиотека Gloss. Вот исчерпывающий список: `black`, `white`, `red`, `green`, `blue`, `yellow`, `magenta`, `cyan`, `rose`, `orange`, `chartreuse`, `aquamarine`, `azure`, и `violet`. Мы все смеялись над тем, как Сью нашла у себя в голове какое-то жуткое название цвета и спросила: «А она знает цвет `chartreuse`?». Да, знает. Прикольное совпадение!

Также вы можете двигать изображения по экрану:

```
import Graphics.Gloss
picture = translate 100 50 (rectangleSolid 50 50)
```

Если вы хотите сдвинуть ваше изображение, то библиотека Gloss использует для этого функцию «`translate`». Да, это непривычное слово, но оно обозначает, что что-либо надо просто сдвинуть вправо, влево, вверх или вниз. Первое число обозначает то, как сдвигать по горизонтали. Положительные числа сдвигают изображения вправо, а отрицательные влево. Соответственно, второе число определяет то, насколько сдвигать изображение по вертикали. Положительные числа задают сдвиг вверх, а отрицательные — вниз.

Имейте в виду, что в языке Haskell отрицательные числа записываются в скобках. Если вы запишете выражение «`translate -100`», то язык Haskell подумает, что вы хотите отнять число 100 из слова «`translate`». Он не знает, как вычесть число из функции (я тоже не знаю), поэтому он сразу же сдастся. Но вместо этого вам следует написать «`translate (-100)`», и всё будет в порядке.

Также вы можете поворачивать вещи. Для этого используется функция «`rotate`». Например, нарисуем-ка ромб:

```
import Graphics.Gloss
picture = rotate 45 (rectangleWire 100 100)
```

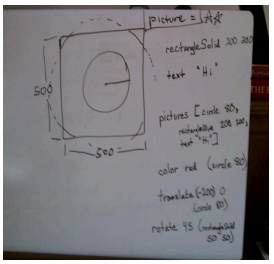
Вы поворачиваете изображения на некоторый градус. Помните, что поворот на 360 градусов обозначает поворот к исходному состоянию, как будто бы ничего и не поворачивалось. 45 градусов составляют половину прямого угла. Видите, по каким причинам получился ромб?

Наконец, вы можете изменять размеры изображения, для чего используется функция «`scale`»:

```
import Graphics.Gloss
picture = scale 2 1 (circle 100)
```

Этот код нарисует *эллипс*, который похож на круг, но в два раза шире, чем его размер по высоте.

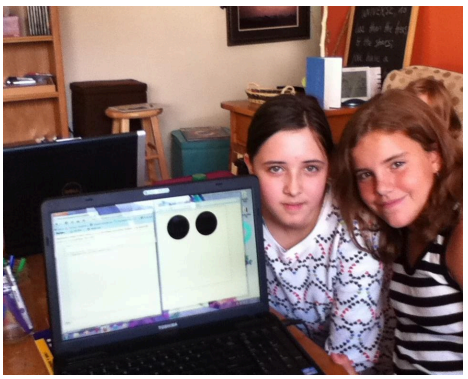
Не переживайте, если для вас всё это ещё имеет мало смысла. Мы ещё потратим много времени для изучения того, как комбинировать вместе все эти штуки. Вот фотография нашей доски после того, как мы изучили всё это:



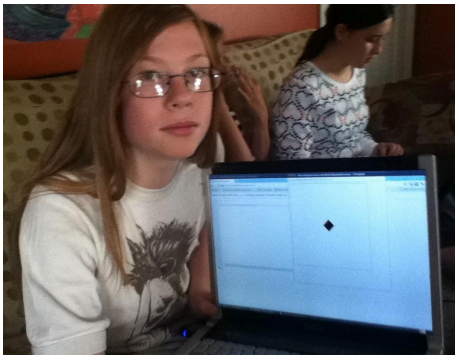

## Время для экспериментов

Мы потратили много времени на то, чтобы каждый ученик нарисовал то, чего бы желал. Наиболее комфортно делать это во время игры. Делайте много мелких изменений и смотрите, к чему они приводят. Сначала попытайтесь предугадать, что сделает изменение, потом попробуйте и посмотрите, оказались ли вы правы или нет.

А вот несколько картинок ребят с их проектами:

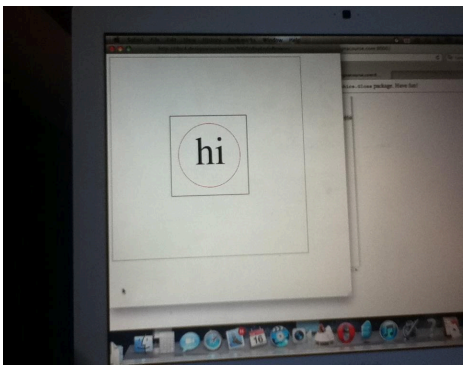


Вот София и Эви гордятся своими кругами. У них внезапно получились глаза на лице.



А вот Грант со своим ромбом. Ромб выглядит несколько лучше после того,

как он растянул его в вертикальном направлении.




Это изображение Марчелло... Центровка слова внутри круга была непростой задачей. Если вы попробуете её решить, то вы увидите, что текст пишется не ровно в центре экрана, как рисуются другие фигуры, так что Марчелло затратил много времени на эксперименты с функцией «`translate`» для точной центровки в круге.

А вот очень воодушевлённая София после того,
как смогла расположить свои «глаза» в нужном месте.

### Домашнее задание

Вашим заданием, если вы примете его, будет планирование и рисование некоторого изображения, которое вам интересно. Возможно, что это рыба, сад, космическая станция или дракон. Просто убедитесь, что вы можете это нарисовать при помощи кругов и прямоугольников различных цветов, а также при помощи перемещений, поворотов и растягиваний. Здесь в Little School мы будем выполнять это задание всю оставшуюся неделю, а также будем учиться этому на занятиях на следующей неделе. Потратьте немного времени и возвращайтесь с тем, чем вы могли бы гордиться.

## Неделя 2

Пришло время описать вторую неделю наших занятий по программированию (вот ссылка на предыдущее описание).

### Изображения из домашних заданий прошедшей недели

Помните, на прошлой неделе домашним заданием было выбрать какое-либо

интересное для вас изображение и нарисовать его. Вот некоторые результаты:

- Марчелло: http://pilotkid.edublogs.org/2011/08/18/hakell-rocks/
- София: http://sally2135x6.edublogs.org/2011/08/18/gloss-programmingupside-down-elephant-drawing/
- Грант: http://brugdush.edublogs.org/2011/08/24/haskell-dart-board/
- Миссис Сью: http://mssuesun.edublogs.org/2011/08/17/haskell-is-cool/
- Моя собственная картинка: http://cdsmith.wordpress.com/2011/08/23/haskell-for-kids-my-project/

## Тема недели: порядок

Темой этой недели является *порядок*. Одна из проблем, которые не позволяют быстро войти в тему компьютерного программирования, заключается в том, что вы пишете программы, которые так захватывающи и сложны, что вы попадаете в тупик: программа даже может работать, однако она становится очень сложной для понимания, и вы боитесь её трогать даже для того, чтобы немного подправить. Мы потратили немного времени для того, чтобы обсудить, как организовывать ваши компьютерные программы так, чтобы их было легко понимать.

## Части программ

В качестве примера программы, которую можно было бы написать в первую неделю обучения, рассмотрим такую:

```
import Graphics.Gloss
picture = color red (pictures
                      [circle 80,
                       translate 50 100 (rectangleSolid 100 50)])
```

Мы тщательнее изучили то, из чего состоят программы. Вот их части.

### Оператор импорта

Первая строка представляет собой оператор импорта. Помните, при помощи неё мы говорим компьютеру о том, что мы будем использовать библиотеку Gloss. Если бы этой строки не было бы, то у компьютера не было бы даже никаких идей о том, откуда ему брать информацию о таких словах, как «`color`», «`red`», «`pictures`», «`cirlce`» и т. д. И вам надо импортировать библиотеку только один раз, независимо от того, сколько раз вы используете программные сущности из неё.

### Определения

После оператора импорта ваша программа определяет несколько новых слов, которые называются *функциями*. Единственной функцией, которую мы определяли в наших первых программах, была функция «`picture`». На текущий момент мы определяем только одну функцию именно с таким именем, чтобы компьютер понимал, что именно мы хотим, чтобы он нарисовал. Что мы не знали на прошлой неделе, так это то, что кроме одной функции можно определять и другие. И после этого их можно использовать абсолютно так же, как и такие функции, как «`circle`» и другие слова из библиотеки Gloss.

Например, посмотрите-ка на такую программу:

```
import Graphics.Gloss

picture = snowman

snowman = pictures [translate 0 ( 50) top,
                    translate 0 (  0) middle,
                    translate 0 (-75) bottom]

top    = circleSolid 50
middle = circleSolid 70
bottom = circleSolid 90
```

Эта программа определяет пять функций: «`picture`», «`snowman`», «`top`», «`middle`» и «`bottom`». Опять же, компьютер нарисует только то, что определено в функции «`picture`», однако остальные функции нам нужны для того, чтобы организовывать нашу программу по блокам. Вот два важных правила программирования, которые являются частью языка Haskell:

1. Каждое новое определение должно начинаться с самого первого символа строки. Оно не должно иметь отступов.
2. Если вы продолжаете определение на следующей строке, то вторая строка уже должна иметь отступ. Не важно, какого размера, хотя бы немного.

Так вышло, что сначала я определил функцию «`picture`», затем функцию «`snowman`» и т. д., но некоторым людям может показаться, что надо было бы определить эти функции в другом порядке. Прекрасно! Порядок определений вообще не играет никакой роли, так что чувствуйте себя свободней при написании своих программ. Например, сначала определяйте функцию «`picture`», а потом одну за одной определяйте те функции, которые в ней используются (иногда это называется «разработка сверху вниз»), либо начните с определения всех своих вспомогательных частей, а когда они будут готовы, определите функцию «`picture`» (а это называется «разработка снизу вверх»). Вы даже можете разместить все ваши определения в алфавитном порядке. Компьютеру всё равно. (На самом деле, я не рекомендовал бы использовать алфавитный порядок, но вы могли бы его использовать).

### Выражения

Части ваших программ, стоящие в определениях после символа равенства (=), называются *выражениями*. Это то место, где вы говорите компьютеру то, что обозначают ваши функции. На прошлой неделе мы изучали простейшие выражения типа вывода геометрических фигур, а также чуть более сложные типа изменения цвета или месторасположения.

На второй неделе мы обсуждали выражения более подробно. Посмотрите на следующее определение:

```
picture = translate 10 20 (circleSolid 50)
```

Выражение в правой части определения функции «`picture`» состоит из четырёх

блоков, выставленных друг за дружкой. Когда вы видите что-то вроде этого, то должны понимать, что это *вызов функции*, причём первый блок — это имя функции, а остальные блоки — её *параметры*. Так что здесь у нас:

1. `translate` — это имя функции.
2. `10` — это первый параметр.
3. `20` — это второй параметр.
4. `(circleSolid 50)` — вся эта сложная штука является третьим параметром.

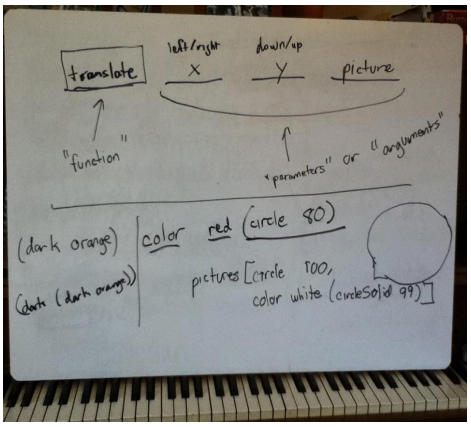
Обратите внимание на круглые скобки! Они обозначают: «воспринимай всё, что внутри скобок, как один объект».

Каждая функция ожидает определённое количество параметров. Например, функция «`translate`» всегда ждёт три параметра: первым идёт расстояние для сдвига по горизонтали, вторым — по вертикали, а третьим — изображение, которое надо сдвинуть. Если вы дадите этой функции только два параметра или пять параметров, она не будет работать. (Если хотите, попробуйте и посмотрите, что получится).

А что можно сказать о части программы, приведённой в скобках? Это просто ещё одно выражение. Можете ли вы понять, какая его часть является функцией, а какая параметром?

Есть ещё одна важная деталь. Вы могли заметить, что функция «`pictures`» другая. Часть программы, которая идёт после функции «`pictures`», заключена в квадратные скобки, в которых находится столько выражений, сколько вам хочется, главное, чтобы все они разделялись запятыми. Это просто другой вид выражений, называемый *списком*. Пока мы видели только функцию «`pictures`», которая использует списки, но скоро мы увидим ещё некоторые функции.

Вот наша доска после того, как мы обсудили части программ, функции и выражения:




Пузырь для слов относится к другому обсуждению, в котором мы рассматривали возможность рисования интересных фигур при помощи наложения белых фигур сверху других фигур.

## Некоторые советы

Большинство нижеследующих советов собраны с занятий второй недели. На этой неделе мы рассуждали о хороших советах по поводу написания компьютерных программ.

Советы следующие.

### Порядок превыше всего

Если вы не поддерживаете порядок, то вы не сможете написать интересные компьютерные программы. Ваша программа должна быть спроектирована так, чтобы состоять из небольших кусочков, и вы должны создавать эти кусочки осмысленными. Вот пример того, как не надо писать свои программы:

```
import Graphics.Gloss
pictures = [translate 71 123 (rectangleSolid 30 200),
            rotate 45 (translate 10 1 (color green (circle 50))),
            translate 100 100 (scale 2 2 (rectangleWire 10 10)),
```

и т. д., и т. п.

Это вполне допустимо, если вы экспериментируете или играете. Но в конце концов вам придётся разбить программу на управляемые части. Не очень интересно читать длинный список из 40-ка различных операций рисования, равно как и совершенно непросто потом вносить в него исправления.

Вместо этого попробуйте сделать примерно так, как ранее показано в примере с функцией «`snowman`»: части программы разбиты на более мелкие части так, что думать об одной части очень легко. Конечно, не только в частях дело. Наверняка вы не захотите, чтобы одна часть рисовала траву, одну ногу и солнца, а другая часть рисовала другую ногу и дерево. Попытайтесь найти такие части вашего изображения, чтобы в них был смысл. У программистов есть для этого прикольное слово: мы называем такие части *когерентными*, а само явление *когерентностью*.

Ранее мы уже упоминали о двух подходах к разработке программ: сверху вниз и снизу вверх: вы можете начать с определения всего изображения, а потом плавно переходить к его частям; а можете начать с определения мелких кусочков и строить из них более объёмные вещи. На данный момент вы можете пользоваться тем подходом, какой найдёте более лёгким для себя, однако в этот момент обратите внимание и на противоположный подход — возможно, что для некоторых задач его использование принесёт больше пользы.

### Будьте хорошим писателем

Второй совет, о котором мы говорили, заключается в необходимости быть хорошим писателем. Вы же не просто пишете компьютерную программу для компьютера и потом запускаете её, вы также пишете что-то такое, что потом будут читать другие люди. Вы можете показать свою программу другим ученикам в классе, или учителю, или тысячам других людей.

Хорошим способом общаться с другими людьми при помощи других людей является использование правильных имён для своих функций. Если вы рисуете цветок, то хорошими названиями для функций будут «`leaf`» (лист), «`stem`» (стебель) и «`petal`» (лепесток), а не «`picture1`», «`picture2`» и «`picture3`».

Ещё одним методом общения с читателями своих программ будет использование комментариев. Компьютер будет полностью игнорировать любую часть программы, которая

заключена между символами `{-` и соответствующим `-}` (то есть сначала открывающая фигурная скобка, за которой сразу следует дефис, а потом дефис, за которым сразу следует закрывающая фигурная скобка). Более короткие комментарии можно оформлять при помощи двух дефисов `--`, и компьютер будет игнорировать всё, что находится после них до конца строки. Так что вы можете написать что-то типа этого:

```
{-
    Изображение моего пса, исполнено Крисом Смитом.

    Это изображение моего пса Руби. Он наполовину колли.
-}

picture = pictures [circleSolid 50,       -- Это её тело.
                    rectangleSolid 20 100 -- Это её хвост.
                   ]
```

(Не, конечно, это изображение не похоже на собаку, но простите мне это, поскольку это всего лишь пример).

При помощи выбора хороших имён и снабжения своих программ комментариями вы сделаете программы более лёгкими для чтения как для себя, так и для других людей. Вот, что мы подразумеваем, когда говорим, что надо быть хорошим писателем.

### Скрывайте неважные идеи

Человек не может запомнить все вещи, так что другим хорошим советом при написании компьютерных программ будет сокрытие всей фигни, которая не является важной, что позволит сфокусироваться на ключевых идеях программы.

Мы обсуждали возможности языка Haskell, которые помогают программисту в этом деле, — это ключевые слова **`let`** и **`where`**. Вот пример использования слова **`where`** в программе:

```
import Graphics.Gloss

picture = snowman

snowman = pictures [translate 0 ( 50) top,
                    translate 0 (  0) middle,
                    translate 0 (-75) bottom]
  where
    top    = circleSolid 50
    middle = circleSolid 70
    bottom = circleSolid 90
```

Это практически та же самая программа для рисования снеговика, за исключением одного отличия. Функции «`top`», «`middle`» и «`bottom`» определены после ключевого слова **`where`**, а также внутри определения функции «`snowman`». (Возможно, что это не совсем очевидно, но здесь строки с ключевым словом **`where`** и все остальные, что после неё, имеют отступ, так что они всё ещё относятся к определению функции «`snowman`»).

Это требуется для того, чтобы функции «`top`», «`middle`» и «`bottom`» имели смысл только внутри определения функции «`snowman`». Так что вы можете их использовать при рисовании снеговика (функция «`snowman`»), но если вы попытаетесь использовать их где-либо ещё, то это не будет работать. Компилятор выдаст сообщение об ошибке «Вне области видимости».

Идея *области видимости* играет важную роль в программировании. Областью видимости функции является та часть программы, где она может использована. Когда вы пишете более возвышенные программы, вы обычно не хотите, чтобы каждая из сотен функций была видна повсеместно в программе. Когда программа для отрисовки снеговика будет увеличиваться, мы, возможно, захотим определить функцию «`top`», которая будет обозначать что-то совершенно иное для рисования в других частях изображения. И это вполне приемлемо, поскольку функция «`top`» находится в области видимости определения функции «`snowman`».

Мы рассуждали о такой аналогии. Если вы находитесь в Англии, вы можете сказать: «Сегодня я видал королеву». Но вы не сможете сказать то же самое в США, поскольку это будет иметь мало смысла в виду отсутствия здесь королевы. Слово «королева» имеет различный смысл в зависимости от того, где мы находимся. Так что если бы я отвечал так же, как отвечает компилятор языка Haskell, я бы ответил на вашу фразу о виденной сегодня королеве: «Вне области видимости: королева».

Кстати, а вот способ для написания той же самой программы для рисования снеговика, но при помощи использования ключевого слова **`let`**:

```
import Graphics.Gloss

picture = snowman

snowman = let top    = circleSolid 50
              middle = circleSolid 70
              bottom = circleSolid 90
          in  pictures [translate 0 ( 50) top,
                        translate 0 (  0) middle,
                        translate 0 (-75) bottom]
```

Всё это означает ровно всё то же самое, но у вас просто есть выбор между определением частей до (при помощи ключевого слова **`let`**) или после (при помощи ключевого слова **`where`**) основного определения функции.

### Обращайте внимание на детали

Наконец, последний совет для программистов заключается во внимании к деталям. Есть много ситуаций, в которых обычные люди могут просто догадаться, что вы имеете в виду, однако компьютер не умеет догадываться, и он не будет работать, если вы не расскажете ему в деталях, что надо делать.

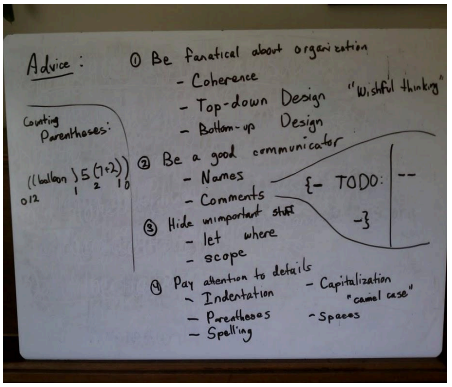
Мы более внимательно рассмотрели некоторые из этих деталей:

- *Отступы*. Вы не можете делать отступов в первой строке определения функции, однако обязаны делать отступ в каждой следующей. Это закон для языка Haskell, и если вы не

будете ему следовать, ваша программа не будет работать. А если вы используете ключевые слова **`let`** или **`where`**, то все определения внутри них должны иметь одинаковый отступ.

- *Скобки*. У многих людей, когда они только-только начинают программировать, скобки вызывают проблемы. (Из-за этого многие из нас всё ещё делают ошибки при использовании скобок, хотя программировали до этого уже годами). Все ваши открывающие и закрывающие скобки должны соответствовать друг другу, или ваша программа не будет работать. Мы рассуждали о некоторых вещах, которые могут помочь в этом деле. Некоторые инструменты для написания программ, в том числе и используемый нами на данном этапе web-сайт, помогают следить за скобками, выделяя их. Зайдите на web-сайт и поставьте курсор сразу за скобкой, и соответствующая ей тут же будет обведена в серый прямоугольник. Это может помочь, но только тогда, когда используемый вами инструментарий умеет это делать. Мы также рассуждали о подсчёте скобок. Есть трюк, который используется многими программистами для контроля скобок в их программах: надо начать с 0, затем читать свою программу от начала до конца и прибавлять единицу в случае обнаружения открывающей скобки, а вычитать единицу в случае обнаружения закрывающей скобки. Если все скобки соответствуют друг другу, то в конце программы вы должны получить 0. Если получен не 0, то необходимо внимательно просмотреть код снова, чтобы обнаружить не совпадающую скобку.
- *Орфография*. Если вы неправильно написали наименование функции (при её вызове, а не определении), то программа не будет работать.
- *Заглавные буквы*. Для языка Haskell очень важно, начинается ли название с заглавной буквы или со строчной. Так, например, слово «`red`» прекрасно работает для обозначения цвета, а слово «`Red`» работать не будет. Мы говорили о соглашении об именовании, которое позволяет использовать заглавные буквы внутри названий, даже если они начинаются со строчной буквы, как в названии функции «`circleSolid`». Это называется верблюжьим стилем, поскольку заглавные буквы в середине слов напоминают горбы верблюда.
- *Пробелы*. В большинстве случаев вы сможете использовать пробелы так, как вам нравится, но часто пробелы просто необходимы. Вы не можете слить вместе наименование функции и какое-либо число, это не будет работать. А многие программисты любят использовать пробелы для выравнивания своих программ в приятные столбцы.

Вот общий вид нашей доски после выполнения этой части занятия:




Внимание к деталям очень важно и очень поможет вам, когда вы станете писать

большие программы.

### Слова замечательных людей

Сегодня мы закончили чтением цитат различных людей, имеющих отношение к программированию. Мы обсуждали цитаты и даже смеялись.

> *Любой дурак напишет программу, которую поймёт компьютер. Хороший программист пишет программы, которые понимают другие люди.*
>
> ***— Мартин Фаулер***

В этой цитате говорится о необходимости быть хорошим писателем, что подчёркивалось в этой записи ранее.

> *Управление сложностью является сутью программирования.*
>
> ***— Брайан Керниган***

Брайан Керниган был одним из двух людей, которые изобрели язык программирования C, один из наиболее популярных языков программирования в мире. Здесь он говорит о необходимости следования порядку и сокрытия неважных деталей так, то вам не нужно будет заботиться о том, что ваши программы становятся всё сложнее и сложнее. Фактически он говорит о том, что это наиболее важная вещь в деле программирования.

> *Любой достаточно серьёзный баг ничем не отличается от особенности.*
>
> ***— Рич Кулавец***

Помните, что «баг» — это ошибка в программе. А её особенность — это то, что программа делает, и предполагается, что она должна это делать. Я добавил эту цитату для Софии, которая в первый день обучения обнаружила, что если на моём web-сайте для проверки программ попытаться растянуть заполненный круг, то в середине экрана останется дыра, которую София решила приспособить в качестве рта для лица, которое она хотела изобразить в своей первой программе. Но на следующий день я починил web-сайт, после чего её программа перестала работать. Это определённо был баг, который превратился в особенность.

> *Если вы хотите добавить в свою программу комментарий, то лучше спросите себя, что вам надо улучшить в программе, чтобы комментарий перестал быть нужным.*
>
> ***— Стив Макконнелл***

Это отличная идея! Мы говорили о том, что комментарии используется для того, чтобы объяснять что делается в вашей программе. Но лучшим, чем комментарии, решением является упорядочивать вашу программу так, чтобы становилось очевидным то, что она делает, так что комментарии вам больше не требуются. Это, конечно же, не обозначает, что вообще не надо писать комментариев, однако если вы можете писать программы просто, комментировать их особо и не надо.

> *Большая программа требует фанатичной преданности красоте. Если вы посмотрите внутрь хорошей программы то обнаружите, что части, которые*

> *вообще не предполагались для чьего-либо глаза, также являются красивыми. Я не претендую на то, что я разрабатываю превосходные программы, однако я стараюсь вести себя при этом так, что в обычной жизни обычно ведёт к получению лекарств по рецептам. Меня сводит с ума плохой код: неправильно расположенные пробелы или ужасные имена функций.*
>
> ***— Пол Грэм***

Несмотря на то, что Пол Грэм шутил, он указал на очень хорошую вещь. Эта цитата касается первого и последнего советов, перечисленных ранее.

> *Есть два способа писать компьютерные программы. Первый заключается в написании настолько простых программ, что в них с очевидностью не бывает ошибок. Второй предполагает написание таких сложных программ, что в них нет очевидных ошибок.*
>
> ***— Чарльз Хоар** (тот самый, чья сортировка)*

Последняя цитата приведена немного изменённой, поскольку в оригинале написана более сложным, техническим языком.

### Домашнее задание

Вашим домашним заданием будет очистка ваших программ. Представьте себе, что вы хотите поделиться с кем-нибудь из своих друзей не только картинкой, которую рисует компьютер, но и самой программой для этого. Попробуйте очистить свою программу настолько хорошо, насколько вы сможете: разбейте её на части наиболее логичным образом, подбирайте наиболее приемлемые названия функций, исправьте все детали и пишите свою программу так, чтобы другим людям её было приятно читать.

До следующей недели.

## Неделя 3

### Где мы находимся?

Мы многое сделали в течение двух прошедших недель. Вот краткая информация о том, что мы уже знаем и умеем:

- Узнали о том, как происходит процесс программирования: что такое компиляторы, языки программирования, библиотеки, и как всё это взаимодействует друг с другом.
- Создавали компьютерные программы для рисования изображений. Мы описывали простые изображения, такие как круги и прямоугольники. Мы изменяли изображения посредством их перемещения, поворота и растягивания. И наиболее важным этапом является то, что мы изучили, как комбинировать простые изображения в сложные.
- Использовали идентификаторы для обозначения частей наших программ, таких как отдельные части наших изображений. Например: *слон*, *солнце* и т. д.
- Поняли для себя, как важно оставаться организованным и держать свои программы в порядке, давать функциям правильные наименования, хранить все связанные идеи и

*комментировать* свой программный код (*комментарии* — это такие части наших программ, которые игнорируются компилятором) для того, чтобы записывать свои заметки прямо в программах.

Если вы хотите, то можете вспомнить все эти идеи, прочитав описания занятий на первой и второй неделях. Давайте, мы подождём...

## Отрезки и многоугольники

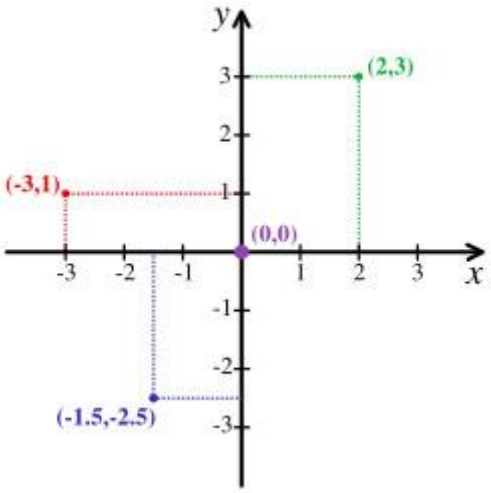
Мы очень много времени потратили на то, чтобы рисовать круги и прямоугольники. Это было прикольно, но это очень даёт очень ограниченные возможности. Что делать, если вы хотели бы нарисовать звезду? Возможно, что вам это удалось бы при помощи растягиваний и поворотов прямоугольников, но это было бы очень нелегко. Для решения этой задачи мы много времени затратили на изучение новых инструментов: отрезком и многоугольников.

### Декартовы координаты

Для того чтобы понять, что такое отрезки и многоугольники, нам надо изучить некоторые специфические понятия из геометрии. Изучать их мы будем при помощи базового понятия *декартовых координат*. Идея состоит в том, чтобы рассуждать о некоторой точке на плоскости при помощи двух чисел:

1. Насколько далеко отстоит точка от центра изображения по горизонтали. Мы называем это число *координатой x*.
2. Насколько далеко отстоит точка от центра изображения по вертикали. Мы называем это число *координатой y*.

Вот картинка, которая может помочь прояснить эти понятия (взята отсюда):




Мы записываем точку как (*x*, *y*), где *x* и *y* — координаты, как было описано раньше: *x* обозначает сдвиг от начала координат вправо или влево, а *y* — вверх или вниз. Точка в самом центре изображения имеет координаты (0, 0), поскольку она лежит ровно по середине как в горизонтальном, так и в вертикальном направлении. Координаты других точек могут быть найдены при помощи отсчёта расстояния от центральной точки по горизонтальной линии (или «*оси x*») до точки, а потом отсчёт вдоль вертикальной линии (или «*оси y*») вниз или вверх до самой точки. Необходимо отметить, что для обозначения движения влево или

вниз мы используем отрицательные числа.

***Упражнение***. Потратьте несколько минут на то, чтобы выбрать на вышеприведённой диаграмме ещё несколько точек и определить их координаты, то есть записать их в виде пары чисел в круглых скобках, как это было показано чуть ранее.

Если вы помните, мы говорили о том, что область для рисования на web-сайте Gloss имеет размер 500 на 500 точек. Это значит, что центр этой области всё равно имеет координату (0, 0), а координаты точек на ней могут изменяться в интервале от -250 до 250. (Почему? Потому что 250 — это половина 500, то есть половина размера области для рисования в обе стороны от центра). На этом этапе может помочь такое упражнение — найдите и возьмите кусок миллиметровой бумаги и нарисуйте на ней квадрат размером 500 на 500 точек, после чего потренируйтесь определять координаты или находить точки по координатам. Например, найдите такие точки как (100, 100) или (-200, 50). В качестве специального бонуса для вас на web-сайте имеется следующая возможность. Во время работы программы наведите на область для рисования курсор мыши, и на всплывающей подсказке будут написаны координаты точки, над которой находится курсор.

## Рисование отрезков

Давайте поиспользуем декартовы координаты для того, чтобы порисовать. Попробуйте следующую программу:

```
import Graphics.Gloss
picture = line [(-100, -100), (-200, 50)]
```

Можете ли вы догадаться, каков будет результат? Запустите и проверьте себя. Эта программа нарисовала отрезок от одной точки до другой. Вы также можете нарисовать больше отрезков и даже зубчатые линии из них, просто перечисляя в списке большее количество координат.

```
import Graphics.Gloss
picture = line [(-200, -200),
                ( 200, -200),
                ( 200,  200),
                (-200,  200),
                (-200, -200)]
```

Постарайтесь представить, где перечисленные точки находятся на экране, и каков будет результат исполнения программы. Теперь запустите программу. Получилось ли то, что вы ожидали?

## Рисование многоугольников

Рисовать-то можно не только отрезки, но и закрашенные области. Такие области называются многоугольниками, и это слово обозначает замкнутую область с прямыми границами. Предыдущий пример можно легко преобразовать в многоугольник:

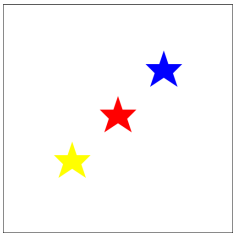
```
import Graphics.Gloss
picture = polygon [(-200, -200),
                   ( 200, -200),
```

```
                  ( 200,  200),
                  (-200,  200)]
```

Прошу отметить, что мне не надо повторять последнюю точку, как это было для отрезков, поскольку любой многоугольник должен быть замкнут в любом случае, так что библиотека Gloss сама подразумевает, что заканчивается периметр многоугольника там же, где и начинается. Конечно, прямоугольники не слишком впечатляют, так что вот вам программа, написанная миссис Сью:

```
import Graphics.Gloss

picture = stars

stars = pictures [translate ( 100) ( 100) (color blue   star),
                  translate (-100) (-100) (color yellow star),
                  translate (   0) (   0) (color red    star)]

star = polygon [(  0,  50),
                ( 10,  20),
                ( 40,  20),
                ( 20,   0),
                ( 30, -30),
                (  0, -10),
                (-30, -30),
                (-20,   0),
                (-40,  20),
                (-10,  20),
                (  0,  50)]
```

Вы можете определить то, что она рисует? Вот результат её исполнения:



## Думаем как компьютер

А вот, что мы делали в течение прошедшей недели. На той неделе мы размышляли о том, как думать о программах с точки зрения компьютера. Мы заново просмотрели всё то, чему уже научились, однако на этот раз мы старались представить самих себя компьютерами во время чтения и интерпретации программ. Вот темы, о которых мы говорили.

### Модули

Когда мы пишем наши компьютерные программы в web-браузере, мы просто печатаем текст программы в поле ввода, после чего нажимаем кнопку «Старт». Всё вместе, что мы

вводим в поле ввода, называется *модулем*. Так что пока вы можете думать о слове «модуль», как о слове, которое обозначает программу целиком. (Мы немного поговорили на тему того, что одна программа может состоять из нескольких модулей. Фактически, большая часть программ во всём мире состоит из нескольких модулей. Однако для нас пройдёт ещё некоторое время, прежде чем нам потребуется разделять наши программы на модули).

Модули содержат внутри себя пару вещей, которые мы уже видели:

- **Декларации импорта**. Они всегда должны находиться в самом начале модуля, и они говорят нам, какие модули (обычно из сторонних библиотек) мы используем. Например, свои программы для рисования мы всегда начинаем с импорта модуля `Graphics.Gloss`, поскольку это позволяет нам использовать всю мощь библиотеки Gloss, все эти круги, цвета и т. д. Обычно мы импортируем одну библиотеку, но если вам требуется, то вы можете импортировать несколько модулей.
- **Определения**. Определение говорит компьютеру, что обозначает то или иное слово. Например, если вы посмотрите на предыдущую программу, то там будет три определения для наименований «`picture`», «`stars`» и «`star`».

Декларация импорта *всегда* начинается со специального слова «**`import`**». Определение *всегда* начинается со слова, которое определяется, за которым следует знак равенства (=), после которого описывается определяемое слово. Иногда в прошлом мы видали людей, которые пытались вставить внутрь модуля какие-то другие вещи, а не определения. Например, может быть даже вы пытались написать что-то вроде этого:

```
import Graphics.Gloss
circle 80
```

Если вы пытались написать подобное, то уже знаете, что это не работает! Вы не можете поставить описание круга в любом месте своей программы. Ошибка, которую вы получите при этом, гласит: «Голое выражение на верхнем уровне». Когда вы видите такую ошибку, вы должны понимать, что просто забыли наименование функции, которую пытались определить.

### Каково быть компилятором

Вот, то делает компилятор: он читает вашу программу и запоминает все слова, которые вы определили. Когда он находит специфическое слово, он пытается понять его значение. В наших программах таким специфическим словом является наименование «`picture`». Так что он смотрит определение этого слова в вашей программе. Если в этом определении встречаются какие-либо слова, значений которых компилятор не знает, он также пытается их найти в программе. И так далее — пока существуют слова, значений которых компилятор не знает, он и ищет. В общем, когда вы пишете программу, вы просто составляете для компилятора словарь для поиска в нём значений слов.

Попробуйте представить в уме, что сделает следующая программа:

```
import Graphics.Gloss

picture = color myFavoriteColor (circle 100)

myFavoriteColor = blue
```

```
stars = pictures [translate ( 100) ( 100) (color blue   star),
                  translate (-100) (-100) (color yellow star),
                  translate (   0) (   0) (color red    star)]

star = polygon [(  0,  50),
                ( 10,  20),
                ( 40,  20),
                ( 20,   0),
                ( 30, -30),
                (  0, -10),
                (-30, -30),
                (-20,   0),
                (-40,  20),
                (-10,  20),
                (  0,  50)]
```

Если вы являетесь компилятором и работаете над этой программой, то вот, что вам придётся сделать:

- Вы видите декларацию импорта и помечаете себе, что в программе используется библиотека Gloss. После этого вы сможете пойти в саму ту библиотеку и посмотреть, как там определяются такие слова, как «`circle`», «`rectangleSolid`», «`polygon`» и «`green`». Теперь вы знаете, что обозначают эти слова.
- После этого вы смотрите на определения в предоставленном модуле. Их три: «`picture`», «`stars`» и «`star`».
- Поскольку в реальной жизни вам надо понять, что обозначает слово «`picture`», вы читаете его определение и пытаетесь его понять.
- Ох ты ж! Вы не знаете, что обозначает слово «`myFavoriteColor`». Но это не проблема, поскольку вы смотрите на определение и понимаете, что это просто синий цвет, «`blue`».
- Теперь вы знаете всё, что вам нужно, поэтому просто рисуете синий круг.

Вы можете удивиться — а зачем тогда слова «`stars`» и «`star`»? Что ж, вы знаете, что они обозначают... Но они никогда не используются в программе. Однако если слово находится в словаре, это не значит, что вам обязательно надо использовать его. Помните, что компилятор использует только те слова, которые ему необходимы для понимания смысла слова «`picture`». Если вы хотите посмотреть на какую-либо фигуру, вам надо встроить её в определение функции «`picture`» (или в какую-либо другую функцию, которая используется в «`picture`» и т. д.).

### Выражения (и я не имею в виду выражений лица)

Также мы немного рассуждали о том, что может идти в определениях после знака равенства. После знака равенства идёт то, что называется *выражением*, и имеется несколько форм выражений, которые мы использовали. Вы уже неоднократно все их видели, но будет лучше, если мы сейчас кратко их повторим. Вот они:

- **Константные функции**. Имя константной функции (то самое слово, которое вы обычно определяете) может быть выражением. Для того чтобы понять, что это слово обозначает, вы смотрите в словарь. Примерами таких функций являются «`picture`» и «`green`».

- **Функции (с аргументами)**. Другим типом выражений, который мы постоянно и повсеместно используем, являются функции со своими *аргументами* (также называемыми *параметрами*). Примерами таких выражений являются «`circle 100`» и «`translate 50 50 star`». Помните, что каждая функция ожидает на вход определённое количество параметров, и вы должны передать её на вход требуемое число.
- **Списки**. Списком является набор выражений в квадратных скобках, разделённых запятыми. Мы уже дважды видели их в разных местах — в качестве параметра функции «`pictures`» и в виде списка других изображений. Параметром функций «`line`» и «`polygon`» является список точек.
- **Точки**. Точка описывается парой координат в круглых скобках, разделённых запятой. Точки нужны вам для рисования отрезков и многоугольников, причём из точек надо составлять списки.
- **Числа**. Число записывается так, как оно есть само по себе. Например, *5* или *1.75*.
- **Строки**. Помните, что для функции «`text`» мы передавали строку, заключённую в кавычки. *Строка* — это тоже выражение.

Вы должно быть заметили, что не всякое выражение представляет собой описание изображения. Некоторые из них представляют числа, строки или списки. Вы должны сами следить за тем, чтобы подставлять правильные выражения в нужные места. Например, если вы определите, что слово «`picture`» является числом *7*, то такая программа не будет работать. Это потому, что число *7* не является изображением, это просто число.

А вот теперь посмотрим на выражения ещё с одной, более интересной стороны. Некоторые выражения состоят из меньших частей, и эти части, в свою очередь, также являются выражениями. Так что мы конструируем выражения из других, более мелких выражений.

А что насчёт примера? Давайте ещё раз посмотрим на программу миссис Сью, которую мы уже видели раньше. Вот она снова, вам не надо прокручивать текст назад:

```
import Graphics.Gloss

picture = stars

stars = pictures [translate ( 100) ( 100) (color blue   star),
                  translate (-100) (-100) (color yellow star),
                  translate (   0) (   0) (color red    star)]

star = polygon [(  0,  50),
                ( 10,  20),
                ( 40,  20),
                ( 20,   0),
                ( 30, -30),
                (  0, -10),
                (-30, -30),
                (-20,   0),
                (-40,  20),
                (-10,  20),
```

```
                ( 0,  50)]
```

Тут есть три определения: «`picture`», «`stars`» и «`star`».

Определение слова «`picture`» крайне незамысловато. Мы говорим, что слово «`picture`» обозначает одну простую вещь — функцию «`stars`». Так что эти два слова обозначат одно и то же. Вы можете определять синоним именно таким образом тогда, когда вам это требуется. Марчелло заметил, что в можете написать:

```
move = translate
```

И после этого вам больше не надо будет вспоминать, что обозначает слово «`translate`». (Однако, всё же, я думаю, что вам надо запомнить, что обозначает это слово, поскольку определение синонима для него поможет в краткосрочном варианте, когда вы пишете одну программу. Как только вам надо будет написать вторую программы, вам снова надо будет вспомнить про слово «`translate`»).

Теперь обратим свой взгляд на слово «`star`», определение которого несколько более длинное. Выражение, использованное для определения слова «`star`», является функцией с аргументами — функцией является слово «`polygon`», а в качестве аргумента ей передан список точек. Помните, что аргумент сам по себе является выражением. В частности, здесь мы видим список, содержащий набор значений, разделённых запятыми, а сам список заключён в квадратные скобки. Мы можем заглянуть глубже, ведь каждое значение в списке само является выражением, более маленьким: в данном случае это точка. И, наконец, каждая координата точки является ничем иным, как *ещё одним* выражением, числом. Так что у нас есть выражение внутри выражения, которое само находится внутри выражения, и которое опять находится внутри самого верхнеуровневого выражения. И для программирования это нормально: вы всегда строите более крупные выражения из имеющихся выражений.

Я оставил самое сложное определение на десерт. Давайте посмотрим на слово «`stars`». На этот раз я опишу его выражение при помощи маркированного списка, и вы увидите, почему я выбрал такую форму представления.

- На самом верхнем уровне слово «`stars`» определено как функция с аргументами.
  - Функцией является слово «`pictures`», которая не состоит из частей.
  - Аргументом является список, то есть набор значений, разделённых запятыми и заключённых в квадратные скобки.
    - Первое значение в списке является функцией с аргументами.
      - Функцией является слово «`translate`», которое не состоит из более мелких частей.
      - Первым аргументом является 100, это просто число и не состоит из частей.
      - Вторым аргументом также является число 100, которое тоже не состоит из составных частей.
      - Третьим аргументом, который указывает, что именно рисовать, является функция с аргументами.
        - Функция называется «`color`».
        - Первым аргументом является слово «`blue`», константная функция.
        - Вторым аргументом является слово «`star`», которое мы уже рассматривали.
    - Второе значение в списке является функцией с аргументами.

        - Функцией является слово «`translate`», которое не состоит из более мелких частей.
        - Первым аргументом является -100, это просто число и не состоит из частей.
        - Вторым аргументом также является число -100, которое тоже не состоит из составных частей.
        - Третьим аргументом, который указывает, что именно рисовать, является функция с аргументами.
            - Функция называется «`color`».
            - Первым аргументом является слово «`yellow`», константная функция.
            - Вторым аргументом является слово «`star`», которое мы уже рассматривали.
    - Третье значение в списке является функцией с аргументами.
        - Функцией является слово «`translate`», которое не состоит из более мелких частей.
        - Первым аргументом является 0, это просто число и не состоит из частей.
        - Вторым аргументом также является число 0, которое тоже не состоит из составных частей.
        - Третьим аргументом, который указывает, что именно рисовать, является функция с аргументами.
            - Функция называется «`color`».
            - Первым аргументом является слово «`red`», константная функция.
            - Вторым аргументом является слово «`star`», которое мы уже рассматривали.

Это было непросто! Однако теперь вы видите, как можно дробить выражения на подвыражения, и каждое подвыражение в свою очередь является выражением, которое, возможно, тоже можно раздробить на части.

## Понимание программ

Последняя тема, которой мы занимались на прошедшей неделе, и которая также была домашним заданием, заключалась в попытках предугадать, что будет делать программа. Как надо работать?

1. Вам необходима миллиметровая бумага — такая, с мелкой квадратной сеткой.
2. На бумаге нарисуйте квадратную область, например 10 клеток в ширину и высоту.
3. Нарисуйте точку в самом центре вашей области, это будет начало координат (0, 0).

Если вы нарисовали квадрат шириной 10 клеток, то каждая клетка обозначает 50 точек. Попробуйте найти некоторые точки в этом квадрате.

- Найдите точку (250, 250).
    - (Вы должны попасть в верхний правый угол квадрата).
- Найдите точку (0, -250).
    - (Вы должны попасть в середину нижней стороны квадрата).

Теперь вашей задачей является прочитать следующие программы и нарисовать то, что они бы нарисовали. Вы можете проверить свои ответы при помощи web-сайта. *Однако будьте осторожны*. Если вы запустите одну из этих программ, компьютер забудет ту программу, над которой вы работали, так что сначала сохраните её копию где-нибудь.

### Упражнение 1

```
import Graphics.Gloss
picture = rotate 45 (rectangleWire 150 150)
```

### Упражнение 2

```
import Graphics.Gloss
picture = rotate 135 (pictures [circle 50,
                                rectangleWire 10 100])
```

### Упражнение 3

```
import Graphics.Gloss
picture = translate 50 0 (rotate 45 triangle)
triangle = line [(0, 0), (100, 200), (-100, 200), (0, 0)]
```

### Упражнение 4

```
import Graphics.Gloss
picture = rotate 90 (translate 100 0 (circle 100))
```

После того как вы выполните все эти упражнения, попробуйте сами написать какую-нибудь программу, а потом предсказать то, что она нарисует. На это упражнение мы потратили некоторое время, поскольку перед тем, как мы перейдём к анимированию, нам надо понимать, что происходит при работе программ.

На этом всё. До следующей недели.

# Неделя 4

Приветствую в нашем еженедельном отчёте об обучении детей программированию на языке Haskell. На прошедшей неделе мы постигли три главные вещи, так что давайте с головой в них окунемся.

## Новый ученик

Перво-наперво в нашем классе появился новый ученик, так что поприветствуем Скайлера! Мы уже прозанимались три недели, но это событие даёт нам возможность осуществить краткий анализ того, что мы прошли. Вот основные идеи:

1. На первой неделе мы говорили о том, что такое компьютерная программа, что такое компилятор, и зачем нам нужны библиотеки. Мы также начали немного программировать на языке программирования Haskell при помощи библиотеки Gloss. Мы познакомились с самыми азами. Вот ссылка на описание Недели 1.
2. На второй неделе мы общались на тему структур компьютерных программ, включая: что такое оператор импорта *import*, как писать определения функций, что такое выражение. Мы также изучили вопрос о том, как полезно быть организованным и держать свои программы в порядке — всё в программе находится там, где оно должно быть. Вот ссылка на описание Недели 2.
3. Наконец, на третьей неделе мы разговаривали о декартовых координатах (*x*, *y*), отрезках и

многоугольниках, а также мы играли «в компьютер», ставя себя на место компьютера и предугадывая, что он будет делать с той или иной программой. Вот ссылка на описание Недели 3.

Присоединение к нашему коллективу Скалера позволило нам вновь пройтись по изученным темам и потренироваться в объяснении этих тем другим людям. Я думаю, что большинство людей также узнало что-то новое.

## Практикуемся в разработке сверху вниз

Другой темой, которую мы рассматривали на этой неделе, было несколько идей, которых мы касались ранее только поверхностно:

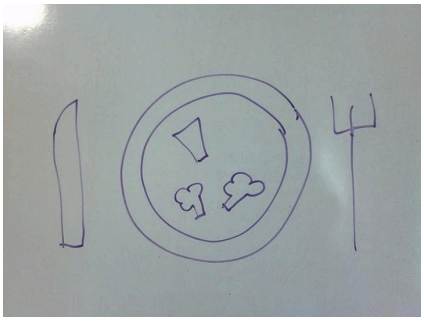
- Разработка сверху вниз.
- Организация и именование объектов.
- Написание заглушек.

Мы проработали ту тему при помощи примера, выполненного во всех деталях в классе. Я попытаюсь воспроизвести эти упражнения здесь. Вы можете, как всегда, испытывать код на web-сайте http://dac4.designacourse.com:8000/ . Помните, важной частью упражнений является изучение всех шагов, которые мы предприняли, а не просто прокручивание до конца и изучение результатов финальной версии программы. Мы изучали технику, которую вы сможете использовать при написании своих собственных программ.

Техника основана на повторяющихся шагах, так что ниже я выделяю полужирным шрифтом (кроме последнего листинга) части программ, которые являются новыми. Так что обычно вы можете игнорировать те части, которые написаны обычным шрифтом, — то, что вы уже видели ранее.

### Цель

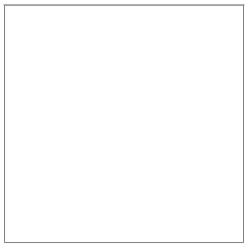
Нашей целью является составление программы, которая нарисует изображение тарелки с ужином. Вот рисунок на доске, иллюстрирующий цель:



Мы начнём с написания базовой программы для рисования изображения. Мы ещё не использовали эту функцию, но она называется «`blank`» и рисует просто пустое изображение.

```
import Graphics.Gloss
picture = blank
```

Если мы запустим эту программу, то результат будет более, чем скучным:



Действительно, скучно. Однако здесь есть нечто, на что надо обратить внимание. Определение изображения с помощью функции «`blank`» называется *заглушкой*. Мы просто добавили в программу нечто, о чём мы знаем, что это нечто ещё не завершено, однако сама программа может уже использоваться. Это вполне приемлемый подход, пока мы помним все места, где использовали заглушки, чтобы позже прийти и наполнить их содержимым.

Первой вещью, что надо сделать, является определение того, что изображение является тарелкой с ужином, «`dinner`». Это выглядит так. Мы также начинаем использовать комментарии (помните, это такие части программы, которые игнорируются компилятором), чтобы запомнить для себя, где мы используем заглушки.

```
import Graphics.Gloss

picture = dinner
dinner = blank  -- stub
```

Это выглядит так же, как и раньше. Мы просто указали, что наше изображение является тарелкой с ужином. А уже функция «`dinner`» определена как пустое изображение «`blank`». Это выглядит бесполезным шагом, однако нельзя недооценивать технику называния объектов правильными именами. Впоследствии это нам очень пригодится.

Теперь давайте исправим заготовку для функции «`dinner`», записав в неё все основные части изображения, которые мы хотим нарисовать. Если вы посмотрите на эскиз, то вы увидите, что, образно говоря, изображение состоит из тарелки, какой-то еды, ножа и вилки. Добавим сюда же и стол, который просто является общим фоном всего изображения. Вот попытка улучшить нашу программу:

```
import Graphics.Gloss

picture = dinner

dinner = pictures [table,
                   plate,
                   food,
                   fork,
                   knife]

table = blank -- stub
plate = blank -- stub
food = blank -- stub
fork = blank -- stub
```

```
knife = blank -- stub
```

Это может выглядеть не совсем выразительно, но сейчас мы работаем над техникой, позволяющей разбивать нашу программу на всё меньшие и меньшие части. Теперь давайте попробуем нарисовать несколько простых частей. Во-первых, стол — это прямоугольник размером 500 на 500 точек (другими словами, вся область для рисования), закрашенный хорошим цветом для стола, например, коричневым.

```
import Graphics.Gloss

picture = dinner

dinner = pictures [table,
                   plate,
                   food,
                   fork,
                   knife]

table = color brown (rectangleSolid 500 500)
plate = blank  -- stub
food  = blank  -- stub
fork  = blank  -- stub
knife = blank  -- stub
```

Давайте попробуем откомпилировать и запустить эту программу.

```
Not in scope: `brown'
```

Ох ты ж! Мы использовали цвет «`brown`», который не определён в библиотеке Gloss. Но мы же сами можем определить этот цвет. В общем, коричневый цвет — это что-то типа тёмно оранжевого, так что определим как-то так:

```
import Graphics.Gloss

picture = dinner

dinner = pictures [table,
                   plate,
                   food,
                   fork,
                   knife]

table = color brown (rectangleSolid 500 500)
plate = blank  -- stub
food  = blank  -- stub
fork  = blank  -- stub
knife = blank  -- stub

brown = dark orange
```

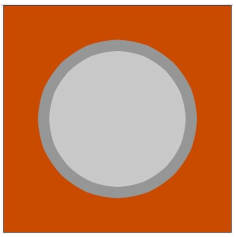
Попробуем снова:

Выглядит хорошо! Пришло время добавить изображение тарелки. Мы хотели бы нарисовать на тарелке ободок, так что тарелка будет представлять собой две окружности — одна чуть поменьше над другой. Нам ещё понадобятся два новых цвета: серый («`gray`») и светло-серый («`lightGray`»). Это не сложно, так что просто напишем:

```
import Graphics.Gloss

picture = dinner

dinner = pictures [table,
                   plate,
                   food,
                   fork,
                   knife]

table = color brown (rectangleSolid 500 500)

plate = pictures [color gray (circleSolid 175),
                  color lightGray (circleSolid 150)]

food  = blank  -- stub
fork  = blank  -- stub
knife = blank  -- stub

brown     = dark orange
lightGray = dark white
gray      = dark lightGray
```

Я тут произвёл с цветами-то кое-какой трюк. Я мог бы определить серый цвет «`gray`» как двойное затемнение белого цвета «`dark (dark white)`», но проще использовать уже определённый ранее цвет «`lightGray`». Оба таких определения будут обозначать одно и то же. Вы можете понять, почему?

Так что теперь мы откомпилируем и запустим нашу программу. Посмотрим:



Как вы видите, всё идёт, в общем-то, неплохо. Время пришло поработать над серебряными приборами, лежащими по бокам тарелки. Что мы об этом знаем? Перечислим:

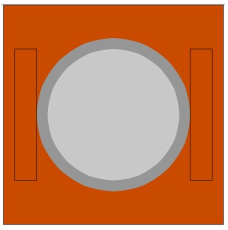
- Нам нужны оба прибора — нож и вилка.
- Мы хотели бы, чтобы они были расположены в середине рисунка по бокам тарелки, где-то на расстоянии 200 символов.
- Они должны быть намного выше, чем толще. Примерно 300 точек в высоту и 50 точек в ширину.

Сейчас мы лучше остановимся, чем продолжим работу, пытаясь сделать всё сразу. Помните, мы пытаемся научиться технике разработки программ при помощи написания за один раз маленьких вещей. Так что сейчас мы просто нарисуем два прямоугольника, игнорируя реальные формы приборов. Из-за того, что мы используем слова «`fork`» и «`knife`» в списке для функции «`dinner`», то есть не в том месте, где они реально определяются, то перемещения, должны производиться именно в определении функции «`dinner`». Результат выглядит так:

```
import Graphics.Gloss

picture = dinner

dinner = pictures [table,
                   plate,
                   food,
                   translate ( 200) 0 fork,
                   translate (-200) 0 knife]

table = color brown (rectangleSolid 500 500)

plate = pictures [color gray (circleSolid 175),
                  color lightGray (circleSolid 150)]

food  = blank  -- stub
fork  = rectangleWire 50 300  -- stub
knife = rectangleWire 50 300  -- stub

brown     = dark orange
lightGray = dark white
gray      = dark lightGray
```

Заметьте, что мы всё ещё определяем сами функции «`fork`» и «`knife`» в виде заготовок. Да, это всё ещё заготовки. Они ещё выглядят совсем не так, как вилка и нож

выглядят на самом деле. Но этот метод помогает нам рассмотреть те области, в которых нам надо будет рисовать эти приборы. Давайте откомпилируем и запустим получившуюся программу. Вот результат:

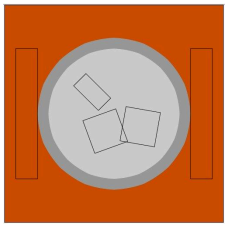


Пришло время для рисованию еды. Мы начнём с разделения еды на три главных части, определяя эти части заготовками в виде прямоугольников, нежели будем сразу рисовать правильные изображения. Каждый кусочек еды перемещён и немного повёрнут.

```
import Graphics.Gloss

picture = dinner

dinner = pictures [table,
                   plate,
                   food,
                   translate ( 200) 0 fork,
                   translate (-200) 0 knife]

table = color brown (rectangleSolid 500 500)

plate = pictures [color gray (circleSolid 175),
                  color lightGray (circleSolid 150)]

food = pictures [translate (-50) ( 50) (rotate ( 45) carrot),
                 translate (-20) (-40) (rotate ( 20) broccoli),
                 translate ( 60) (-30) (rotate (-10) broccoli)]

carrot   = rectangleWire 40 80   -- stub
broccoli = rectangleWire 80 80   -- stub
fork     = rectangleWire 50 300  -- stub
knife    = rectangleWire 50 300  -- stub

brown     = dark orange
lightGray = dark white
gray      = dark lightGray
```

Не переживайте, если у вас не получается понять, что обозначают все эти числа. Когда мы реализовывали это в классе, мы много времени тратили на обсуждение и подгонку фигур на нужные места. Поскольку мы можем сделать это до того, как посвятим время детальной проработке того, как должна выглядеть брюссельская капуста («`broccoli`»), мы и используем такие вот заготовки. (Также не переживайте, если вы даже не знаете, как читается слово «`broccoli`», мне самому пришлось смотреть его в словаре).

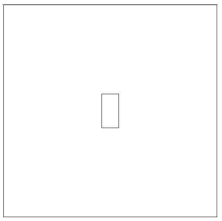
Давайте запустим программу и посмотрим, что получилось:



А сейчас давайте рассмотрим ещё один трюк. Сфокусируем свои усилия на рисовании моркови («`carrot`»). Но поскольку изображение моркови перемещено и повёрнуто, нам будет трудно сфокусироваться на нём. Помните, мы ранее написали определение `picture = dinner`? Я обещал, что вскоре это нам пригодится. Сейчас мы просто изменим это определение на `picture = carrot`. Теперь наша программа выглядит так:

```
import Graphics.Gloss

picture = carrot

dinner = pictures [table,
                   plate,
                   food,
                   translate ( 200) 0 fork,
                   translate (-200) 0 knife]

table = color brown (rectangleSolid 500 500)

plate = pictures [color gray (circleSolid 175),
                  color lightGray (circleSolid 150)]

food = pictures [translate (-50) ( 50) (rotate ( 45) carrot),
                 translate (-20) (-40) (rotate ( 20) broccoli),
                 translate ( 60) (-30) (rotate (-10) broccoli)]

carrot   = rectangleWire 40 80   -- stub
broccoli = rectangleWire 80 80   -- stub
fork     = rectangleWire 50 300  -- stub
knife    = rectangleWire 50 300  -- stub

brown     = dark orange
lightGray = dark white
gray      = dark lightGray
```

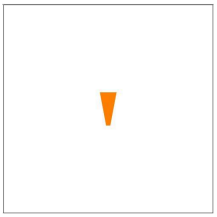
Когда мы ранее запускали эту программу, мы просили компьютер нарисовать нам всё для ужина, все эти вилки и ножи, и всё остальное. А теперь мы просим его нарисовать нам одну только лишь морковь. И это прекрасно. Компьютер чувствует себя счастливым, когда может проигнорировать все определения, которые не используются в программе. Результат выглядит так:



Пока мы отложили всё, что не относится к моркови, а она, как мы хотели, выглядит как прямоугольник прямо в центре экрана. Убрав все эти вилки и кочаны брюссельской капусты, нам будет довольно-таки несложно определить, чем является морковь. Это просто оранжевый многооугольник. После некоторых экспериментов вы можете остановиться на следующем варианте:

```
import Graphics.Gloss

picture = carrot

dinner = pictures [table,
                   plate,
                   food,
                   translate ( 200) 0 fork,
                   translate (-200) 0 knife]

table = color brown (rectangleSolid 500 500)

plate = pictures [color gray (circleSolid 175),
                  color lightGray (circleSolid 150)]

food = pictures [translate (-50) ( 50) (rotate ( 45) carrot),
                 translate (-20) (-40) (rotate ( 20) broccoli),
                 translate ( 60) (-30) (rotate (-10) broccoli)]

carrot   = carrot = color orange (polygon [( -5, -40),
                                           (-20,  40),
                                           ( 20,  40),
                                           (  5, -40)])

broccoli = rectangleWire 80 80   -- stub
fork     = rectangleWire 50 300  -- stub
knife    = rectangleWire 50 300  -- stub

brown     = dark orange
lightGray = dark white
gray      = dark lightGray
```

Это выглядит так:



Теперь, нарисовав морковь, давайте снова изменим определение функции «`picture`», чтобы посмотреть, как выглядит всё изображение. Вот код:

```
import Graphics.Gloss

picture = dinner

dinner = pictures [table,
                   plate,
                   food,
                   translate ( 200) 0 fork,
                   translate (-200) 0 knife]

table = color brown (rectangleSolid 500 500)

plate = pictures [color gray (circleSolid 175),
                  color lightGray (circleSolid 150)]

food = pictures [translate (-50) ( 50) (rotate ( 45) carrot),
                 translate (-20) (-40) (rotate ( 20) broccoli),
                 translate ( 60) (-30) (rotate (-10) broccoli)]

carrot   = carrot = color orange (polygon [( -5, -40),
                                           (-20,  40),
                                           ( 20,  40),
                                           (  5, -40)])

broccoli = rectangleWire 80 80   -- stub
fork     = rectangleWire 50 300  -- stub
knife    = rectangleWire 50 300  -- stub

brown     = dark orange
lightGray = dark white
gray      = dark lightGray
```

К счастью, теперь вы сами можете видеть, как закончить нашу программу. Для всех оставшихся частей нашей программы, для одной за одной по очереди мы будем делать следующие шаги:

1. Изменять определение функции «`picture`», чтобы сфокусировать своё внимание на выбранной части.
2. Писать программу для точного рисований этой части.

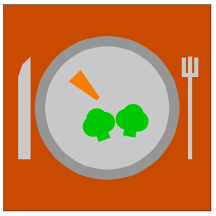
3. Опять изменять определение функции «`picture`», чтобы рисовалось всё изображение.

В интересах экономии места я пропускаю все оставшиеся шаги, однако вот окончательный вариант нашей программы:

```
{-
    Изображение тарелки с ужином.

    Эта программа рисует натюрморт: тарелку с ужином,
    состоящим из нескольких овощей,
    вокруг которой лежат серебряные приборы.
-}

import Graphics.Gloss

picture = dinner

dinner = pictures [table,
                   plate,
                   food,
                   translate ( 200) 0 fork,
                   translate (-200) 0 knife]

table = color brown (rectangleSolid 500 500)

plate = pictures [color gray      (circleSolid 175),
                  color lightGray (circleSolid 150)]

food = pictures [translate (-50) ( 50) (rotate   45  carrot),
                 translate (-20) (-40) (rotate   20  broccoli),
                 translate ( 60) (-30) (rotate (-10) broccoli)]

carrot = color orange (polygon [( -5, -40),
                                (-20,  40),
                                ( 20,  40),
                                (  5, -40)])

broccoli
  = color (dark green)
          (pictures [translate (  0) (-15) (rectangleSolid 30 50),
                     translate (-15) (  0) (circleSolid 25),
                     translate ( 15) (  0) (circleSolid 25),
                     translate (  0) ( 15) (circleSolid 25)])

fork
  = color lightGray
          (pictures [rectangleSolid 10 250,
                     translate (  0) ( 80) (rectangleSolid 40 10),
                     translate (-15) (100) (rectangleSolid 10 45),
```

```
                    translate ( 15) (100) (rectangleSolid 10 45)])

knife
  = color lightGray
          (pictures [translate 0 (-25) (rectangleSolid 30 200),
                     polygon [(-15,  75),
                              ( -5, 105),
                              ( 15, 125),
                              ( 15,  75)]])

brown     = dark orange
lightGray = dark white
gray      = dark lightGray
```

И вот окончательный вариант изображения:



Вуа-ля! Мы закончили. Важнейшая вещь, которую надо понять, заключается в том способе, которым мы нарисовали всё изображение — сначала сделали набросок из заготовок, а потом прорисовывали детали бит за битом. В этом и заключается идея *разработки сверху вниз*, которая является прекрасным методом написания программ в принципе.

## О, нет! Экзамен!!!

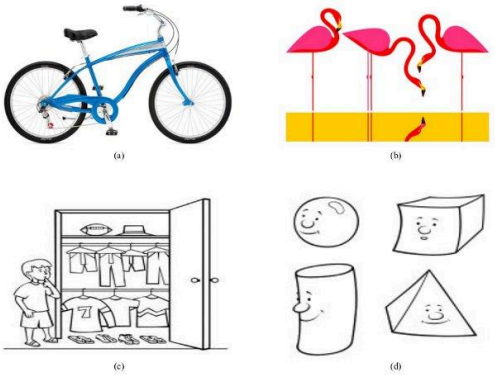
Мы затратили много времени на разговоры об изображениях, и некоторые из учеников собрались двигаться дальше. Однако перед этим нам надо удостовериться, что все всё поняли относительно тех идей, с которыми мы познакомились во время написания наших программ, рисующих изображения. Так что пришло время показать свои навыки в написании таких программ. Пришло время сдавать экзамен.

Вот описание того «экзамена», который я устроил своим ученикам в классе. Мы займёмся им до начала следующей недели. Не беспокойтесь об этом. Это мероприятие необходимо нам для того, чтобы всё снова вспомнить перед тем, как мы двинемся дальше. Я не собираюсь никого заваливать.

### Инструкции

Вот то, над чем нам надо поработать, прежде чем на следующей неделе на наших занятиях мы перейдём от рисования изображений к анимации.

1. Выберите какое-нибудь одно из следующих изображений:



2. Напишите программу, которая рисует упрощённую версию выбранного вами изображения. Пишите свою программу, используя метод **разработки сверху вниз**, как мы его обсудили ранее.
3. В качестве специального правила используйте такое: разбейте выбранное изображение на три составных части, каждую из которых, в свою очередь, разбейте от двух до четырёх таких составных частей, детализация которых важна, и над которыми вы можете внимательно поработать. **Не пытайтесь** рисовать всё.
4. В своей программе используйте все нижеперечисленные примитивные изображения хотя бы один раз:
   - Окружность.
   - Прямоугольник.
   - Отрезок или многоугольник (можно использовать что-то одно, а можно и оба).
   - Сдвиг, поворот и растяжение.
   - Цвета (выбирайте такие, которые кажутся наиболее подходящими).
   - Вставляйте комментарии там, где это необходимо.

***Помните***: целью является не просто заставить компьютер нарисовать изображение. Целью также является написание программы, которую просто читать и понимать, которая выглядит чистенько и упорядочено, которая сама собой объясняет изображение. Будьте хорошим писателем.

## Об этих заметках

Да, теперь становится совершенно ясно, что иногда мне приходится просиживать все выходные, чтобы описать всё то, что мы делали в классе за прошедшую неделю. Так что с этого момента я перестаю использовать слово «позже» и определяю, что заметки в блоге будут доступны в конце недели. Я просто изменяю определение слова «позже», чтобы больше не опаздывать. Умно, не правда ли?

До следующей недели веселитесь и задавайте свои вопросы в комментариях.

# Неделя 5

## Функции

Добро пожаловать в краткие заметки о моём обучении детей программированию на языке Haskell. Если вы хотите, вы можете прочитать все предыдущие обзоры по следующим

ссылкам:

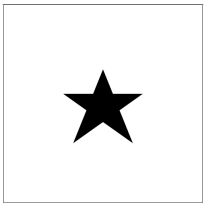
1. Первая неделя: Основы
2. Вторая неделя: Порядок
3. Третья неделя: Представляем себя компьютером
4. Четвёртая неделя: Разработка сверху вниз

На этой неделе мы более внимательно посмотрим на то, что представляют собой функции.

## Раздел 1. Как нарисовать удивительного слона

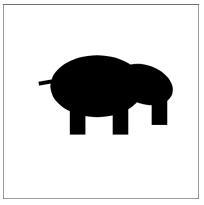
Начнём с простого примера. Мы уже знаем, как нарисовать звезду:

```
star = polygon [( -75, -100),
                (   0,   87),
                (  75, -100),
                (-100,   25),
                ( 100,   25),
                ( -75, -100)]
```



А вот София придумала, как нарисовать слона (здесь представлена одна из её самых первых версий, однако я выбрал её потому, что она короче; вам не надо вникать в суть):

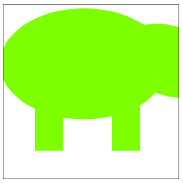
```
elephant = pictures [
      rotate (-20) (scale 3 2 (translate 30 40 (circleSolid 25))),
      translate 150 (-20) (rectangleSolid 40 80),
      translate (-10) 40 (scale 1.5 1 (circleSolid 80)),
      translate 50 (-50)(rectangleSolid 40 70),
      translate (-60) (-50) (rectangleSolid 40 70),
      translate (-140) 50 (rotate 100 (rectangleSolid 10 40))]
```



Но теперь мы хотели бы нарисовать действительно удивительную звезду, что, без всяких сомнений, обозначает, что она должна быть больше и зеленовато-жёлтого цвета.

```
awesomeStar     = scale 2 2 (color chartreuse star)
awesomeElephant = scale 2 2 (color chartreuse elephant)
```

и



И теперь вы вероятно догадались, что я могу нарисовать удивительную версию *любого* изображения. Можно нарисовать удивительные космические корабли, удивительные ботинки, удивительные мотоциклы... Дайте мне произвольную картинку, и я скажу вам, как нарисовать её удивительную версию: вы просто рисуете её в два раза крупнее и разукрашиваете в зеленовато-жёлтый цвет. Но вскоре станет очень скучно повторять это дело раз за разом, однако язык Haskell предоставляет нам возможность определить что-то вроде этого:

```
awesome x = scale 2 2 (color chartreuse x)
```

Теперь нам не надо определять многое множество константных функций типа «`awesomeStar`» или «`awesomeElephant`», вместо этого мы просто пишем `awesome star` с пробелом между словами.

Мы уже видели нечто похожее и ранее: слово «`awesome`» называется **функцией**, а то, что стоит после этого слова, называется её **аргументом**. Но до сегодняшнего момента мы использовали функции, которые другие люди определяли для нас — такие как «`rotate`» или «`translate`» из библиотеки Gloss. А теперь мы знаем, как определять свои собственные функции.

Подводя итог сказанному, подчеркнём — мы можем определить функцию, потому что:

1. Для любого заданного изображения мы знаем, как нарисовать его удивительным.
2. Удивительные версии рисуются при помощи следующей формулы: `scale 2 2 (color chartreuse __________)`, где вы подставляете на пустое место своё изображение.

Если оба утверждения истины, то вы можете определить функцию. Определение функции выглядит ровно так же, как и определение константной функции, которые мы использовали ранее, только в нём есть **параметры**. Параметром называются все слова, которые находятся перед символом равенства (`=`), **за исключением** первого слова. В функции, которую мы рассмотрели несколько минут назад, в качестве параметра используется слово «`x`». Параметры являются просто символами-заполнителями. Когда вы используете функцию, вы передаёте ей аргумент, который встаёт на все места, где в определении используется параметр.

Мы также разговаривали о том, что значит быть прелестным («`cute`»), и пришли к такому пониманию:

```
cute x = scale 0.5 0.5 (color purple x)
```

Давайте подумаем... Для любой фигуры, которую вы можете нарисовать, мы можем нарисовать её прелестную версию. Такое рисование следует указанной формуле — уменьшает изображение в два раза и разукрашивает его пурпурным цветом. И мы написали определение функции «`cute`», в котором параметр имеет название «`x`». Если вам захочется нарисовать прелестного слона, то вы можете определить своё изображение как `picture = cute elephant`. Здесь функция «`cute`» принимает аргумент «`elephant`», так что для того, чтобы понять, что происходит, замените в определении функции «`cute`» все вхождения параметра «`x`» на аргумент «`elephant`».

## Раздел 2. Слоны могут вилять хвостами...

Давайте посмотрим на другую функцию. Мы знаем, как нарисовать слона... Но есть одна проблема. Не важно, каким мы нарисовали слова, его хвост всегда торчит под одним и тем же углом. Представьте себе, что мы хотим нарисовать слона, чей хвост торчал бы под тем углом, под которым бы нам хотелось.

Похоже, что вся функция для рисования слона останется без изменений, за исключением одного вызова функции «`rotate`» там, где нам надо повернуть хвост на заданный угол. Так что у нас на входе имеется:

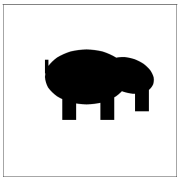
1. Для любого угла мы можем нарисовать слона, чей хвост торчит под этим углом.
2. То, как мы рисуем слона, каждый раз повторяет одну и ту же формулу, за исключением одного числа в одном месте.

Звучит так, как будто бы нам надо определить функцию!

```
elephant angle
  = pictures [
      rotate (-20) (scale 3 2 (translate 30 40 (circleSolid 25))),
      translate 150 (-20) (rectangleSolid 40 80),
      translate (-10) 40 (scale 1.5 1 (circleSolid 80)),
      translate 50 (-50)(rectangleSolid 40 70),
      translate (-60) (-50) (rectangleSolid 40 70),
      translate (-125) 50
        (rotate angle (translate 0 20 (rectangleSolid 10 40)))]
```

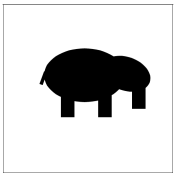
Видите, как это работает? Большая часть слона осталась той же самой, кроме вызова функции «`rotate`» в последней строке, которой передаётся угол «`angle`». Так что какой угол мы зададим, таким хвост и будет. Например, торчащим ровно вверх.

```
picture = elephant 0
```



Или практически опущенным вниз...

```
picture = elephant 160
```



Так что мы снова можем использовать функцию: на этот раз для рисования слона, но оставляя на потом решение вопроса о направлении его хвоста.

## Раздел 3. Списки, списки вокруг нас

Интересным моментом относительно функций является тот факт, что у функций есть параметры, которые представляют собой специального рода переменные, которые могут принимать различные значения в зависимости от способа использования функции. Есть друой способ того, чтобы делать что-то похожее, — это использование списков. Мы посмотрим на некоторые виды списков.

Списки, которые мы использовали до этих пор, выглядели примерно следующим образом:

```
[1, 2, 3, 4, 5]
```

Другими словами, список всегда начинается открывающей квадратной скобкой, за которой идут элементы списка (то, из чего состоит список), разделённые запятой, после которых следует закрывающая квадратная скобка. Элементы списка не обязательно должны быть числами... Мы же уже работали со списками изображений и списками точек. Вы можете создать список из чего угодно, главное, соблюсти форму — квадратные скобки и запятые. Мы можем назвать эту форму «простыми списками», поскольку их очень просто использовать.

Второй формой задания списка, которой мы ещё не встречали, является определение интервала. Это выглядит примерно так:

```
[1 .. 10]
```

Такая форма состоит из открывающей квадратной скобки, начального значения интервала, двух точек, конечного значения интервала и закрывающей квадратной скобки. Этот вид списков считает элементы за вас. Элементами таких списков (по крайней мере, сейчас, на текущий момент) может быть только числа, которые можно пересчитывать... И компьютер заполнит недостающие элементы за вас. В рассмотренном примере список, который мы записали, одинаков с таким: `[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]`. Задание интервала — это просто способ сократить запись.

Есть ещё один трюк, который мы можем здесь применить. Можно задать два числа из начала интервала: первый и следующий за ним элемент. И в этом случае компьютер продолжит последовательность:

```
[2, 4 .. 10]
```

Эта форма выглядит практически так же, как и интервал, за исключением того, что мы

задали *два* числа в начале. Компьютер это поймёт и продолжит последовательность так, чтобы окончательный результат стал таким: `[2, 4, 6, 8, 10]`.

Наконец, есть наиболее интересная форма списка, называемая *генератором списков*. Вот пример:

```
[x + 7 | x <- [1 .. 5]]
```

Здесь немного частей, так что давайте пройдёмся по каждой из них:

- Генератор списков начинается с открывающей квадратной скобки, как и обычный список.
- Далее стоит выражение. Надо отметить, что в выражении используется переменная «`x`», которая ещё нигде не определена. Это нормально, позже она найдётся.
- В середине мы видим вертикальную черту. Она отделяет выражение от следующей части.
- Теперь у нас стоит что-то вроде определения переменной, за исключением того, что мы говорим, что переменная «`x`» последовательно принимает *каждое* значение из другого списка. Мы указываем это при помощи обратной стрелки, которая печатается как символ «меньше» с дефисом после него.
- Наконец, генератор закрывается квадратной скобкой.

Это обозначает следующее: верни мне список всех значений (*x* + 7), где *x* принимает значения из заданного списка. Эта запись является эквивалентом такого списка: `[1 + 7, 2 + 7, 3 + 7, 4 + 7, 5 + 7]`, и, конечно, представляет собой список `[8, 9, 10, 11, 12]`.

Эта последняя форма списка очень, очень интересна. Но пока она, быть может, выглядит слишком сложной, и вам она может быть непонятна. Для понимания того, почему она интересна, давайте начнём определять различные списка изображений, вместо список чисел.

## Раздел 4. Генераторы списков и изображения

Последняя вещь, которой мы занимались на прошедшей неделе, заключалась в комбинировании генераторов списков и изображений для получения ещё более интересных изображений. Я сейчас предоставлю кучу примеров, но не покажу изображений, которые они рисуют. Вашей задачей является *сначала* постараться понять, что же там изображено, а только *потом* использовать наш web-сайт Gloss для проверки ваших догадок.

Я категорически настаиваю на том, чтобы вы восприняли этот совет со всей серьёзностью. Используйте миллиметровую бумагу, проработайте определения, сделайте всё от себя зависящее, чтобы понять, что будет изображено перед тем, как попытаться запустить программу и проверить.

Первый пример:

```
picture = pictures [circle x | x <- [10, 20 .. 100]]
```

Второй пример:

```
picture = pictures [translate x 0 (circle 20) | x <- [0, 10 .. 100]]
```

Третий пример:

```
picture = pictures [translate x 0 (circle x) | x <- [10, 20 .. 100]]
```

Четвёртый пример:

```
picture = pictures [rotate x (rectangleWire 300 300) |
                    x <- [0, 10 .. 90]]
```

Пятый пример:

```
picture = pictures [translate x 0 (rotate x (rectangleWire 300 300)) |
                    x <- [-45, -35 .. 45]]
```

Мы также можем использовать несколько переменных, которые определяются справа от вертикальной черты, отделяясь друг от друга запятыми. В этом случае мы получим список, составленный из всех возможных *комбинаций* значений этих переменных. Это мы тоже можем использовать.

Шестой пример:

```
picture = pictures [translate x y (circle 10) |
                    x <- [-200, -100 .. 200],
                    y <- [-200, -100 .. 200]]
```

Седьмой пример:

```
picture = pictures [rotate angle (translate x 0 (circle 10)) |
                    x     <- [50, 100 .. 200],
                    angle <- [ 0,  45 .. 360]]
```

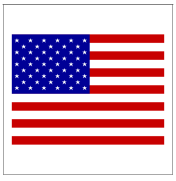
Восьмой пример:

```
picture = pictures [rotate angle (translate (5*x) 0 (circle x)) |
                    x     <- [10, 20 ..  40],
                    angle <- [ 0, 45 .. 360]]
```

Как вы можете видеть, генераторы списков можно использовать для рисоваия прикольных картинок там, где вам пришлось бы заниматься ерундой, вручную набивая координаты для сотен отдельных изображений, как мы это делали до сих пор.

## Домашнее задание

Вашим домашним заданием будет выбрать изображение со множеством повторяющихся элементов и нарисовать его. Если вы живёте в США, и у вас нет никаких идей насчёт изображения, то можете попытаться нарисовать американский флаг.



Подсказки:

- У нас уже есть функция, которая рисует звезду (её определение приведено в начале описания этой недели). Если хотите, можете использовать ту функцию.
- Вам, вероятно, проще будет нарисовать все звёзды в два приёма: сначала большую сетку 5 на 6, а потом малую сетку 4 на 5 звёзд внутри первой.

Вам не обязательно надо рисовать флаг. Особенно, если живёте вы не в США, ваш флаг, скорее всего, не содержит столько повторяющихся элементов, как наш. Вы вполне можете выбрать что-то иное. Просто придумайте, что бы вы могли нарисовать при помощи генераторов списков, которые можно использовать для создания списка изобаржений.

Удачи!

### Ссылки

Ещё одна вещь... Я собрал вместе всю информацию о том, что мы уже научились: отрезки, многоугольники, виды списков. Это может вам пригодиться. Вот ссылка: многоугольники, отрезки и списки. Берите, если вам нужно.

## Неделя 6

(Эта заметка описывает занятия на прошлой неделе. Да, я действительно опоздал с публикацией этого рассказа... Извините! Описание этой недели будет опубликовано позже).

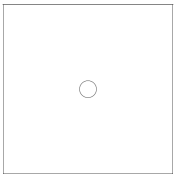
### Время практиковаться

Добро пожаловать на шестую неделю моих занятий «Язык Haskell для детей», которые я провожу в средней школе (для 12 — 13 летних учеников) и на которых мы изучаем язык Haskell и библиотеку Gloss в Little School в Vermijo. Вы можете возвратиться и просмотреть описания недель 1, 2, 3, 4 и 5, если вам этого хочется.

На этой неделе мы практиковались. Вместо представления новых идей я просто опубликую те практические занятия, которые мы проводили во вторник. Мы практиковались очень простым способом: я рисовал изображения, после чего мы пытались сделать их описания. И теперь, вместо того, чтобы тратить время, давайте просто перейдём к этим занятиям. Вам может помочь тот PDF-файл (ссылка находится в конце описания пятой недели, можете даже распечатать его), в котором написаны подсказки о генераторах списков.

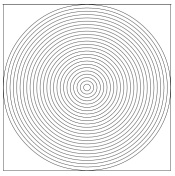
Я не включаю в описание ответы, поскольку слишком легко прокрутить текст вниз и посмотреть их. Если вы затрудняетесь с решением, можете задавать вопросы в комментариях.

#### Упражнение № 1



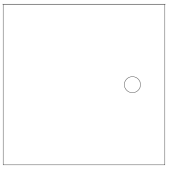
**Подсказка**: Нет! К счастью, это слишком просто.

**Упражнение № 2**



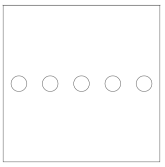
**Подсказка**: Параметром функции «`circle`» является число. Используйте переменную из генератора списка.

**Упражнение № 3**



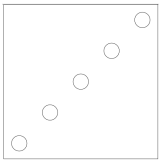
**Подсказка**: Это тоже просто. Это просто подготовка к следующим упражнениям.

**Упражнение № 4**



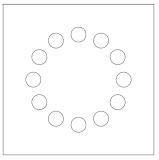
**Подсказка**: Переменная в генераторе списка указывает, насколько производить сдвиг.

**Упражнение № 5**



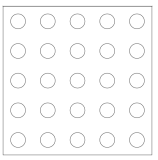
**Подсказка**: Не используйте для рисования этого изображения функцию «`rotate`». Вам надо поменять всего лишь одну вещь в решении упражнения № 4, чтобы получить такое изображение. Помните, что вы можете использовать переменные из генераторов списков неоднократно.

**Упражнение № 6**



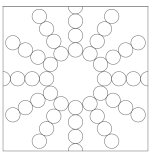
**Подсказка**: Помните, что вы сначала можете сдвинуть изображение, а потом вращать его, и оно будет вращаться вокруг центра экрана.

### Упражнение № 7



**Подсказка**: Здесь вам надо бы использовать две переменные в генераторе списка. Один из примеров на листе с подсказками показывает, как нарисовать такое изображение.

### Упражнение № 8



**Подсказка**: Опять же вам потребуются две переменные, но теперь одну надо использовать в функции «`translate`», а другую — в функции «`rotate`».

## Проекты

В дополнение к этому мы работали над нашими проектами, посвящёнными генераторам списков. Если вы помните, на прошлой неделе домашним заданием было выбрать изображение с повторяющимися элементами и нарисовать его. Я предложил нарисовать американский флаг, если вы проживаете в США, и у вас нет никаких собственных идей. Другими предложениями, которые прозвучали от учеников, являются: iPhone с сеткой из пиктограмм, клавиатура музыкального синтезатора и спиральная галактика на фоне звёзд. Так что подходите к вопросу творчески!

Ох, и мне надо бы быть гордым за изображение Софии, которая нарисовала клавиатуру, и она очень, очень трудолюбиво над ней корпела. Я очень впечатлён. И использовать генераторы списков — это круто. Вот она:


# Неделя 7

## Где мы находимся

Это описание моих занятий по программированию на языке Haskell с детьми от 11 до 13 лет или около того. Вы можете просмотреть предыдущие записи по следующим ссылкам:

- Неделя 1
- Неделя 2
- Неделя 3
- Неделя 4
- Неделя 5
- Неделя 6

Мы уже шесть недель занимаемся изучением языка Haskell и рисованием различных изображений: от простых картинок до достаточно прикольных сложных рисунков. Теперь же пришло время для следующего большого шага — шага к анимации!

## Использование библиотеки Gloss для анимирования

Мы производим все наши эксперименты с программированием на web-сайте по адресу http://dac4.designacourse.com:8000, на котором можно непосредственно программировать и видеть результат исполнения программы на экране. Всё, что мы делали до этого, надо было исполнять в разделе «Picture» («Изображение») этого web-сайта, перейдя в него посредством нажатия на кнопку:


Теперь же мы готовы перейти к следующему типу программ. Так что начнём, нажав для этого кнопку «Animate» («Анимировать»):


Появившаяся страница будет выглядеть абсолютно так же, как и раньше, но не дайте себя обмануть! Теперь web-сайт ожидает, что вы предоставите ему программу, которая описывает изображение, изменяющееся во времени.

Вы должны помнить, что предыдущий режим работы web-сайта требовал, чтобы в вашей программе была определена функция «`picture`». Всё остальное в программе было просто вспомогательными определениями, которые позволяли определить главную функцию, и после того, как вы закончили работу над программой, web-сайт искал определение функции «`picture`», чтобы нарисовать соответствующее изображение. Ваши другие определения были полезны постольку, поскольку они позволяли более легко определять основную функцию.

То же самое происходит и здесь, в новом режиме работы, за исключением пары вещей:

1. Вместо функции «`picture`» необходимо определять функцию «`animation`».

2. Эта функция принимает один параметр, который определяет, сколько времени прошло с момента запуска.

*Предупреждение*: я с самого начала говорил, чтобы вы не использовали для работы с web-сайтом программу для скачивания браузеров Internet Explorer. Если вы меня не слушали и пользовались ею, то, вполне возможно, она даже работала. До текущего момента. Но теперь она точно не будет работать, так что вам придётся скачать какой-нибудь браузер, типа Firefox, Chrome, Safari, Opera, и использовать его.

Давайте начнём, рассмотрев несколько примеров.

## Пример 1. Вращение квадрата

Прежде всего в качестве примера рассмотрим программу, которая выдаётся вам web-сайтом, когда вы на него заходите:

```
import Graphics.Gloss

animation t = rotate (60 * t) (rectangleSolid 100 100)
```

Вот описание того, как понять, что эта программа делает. Думайте о параметре `t` как о количестве секунд, которое прошло с того момента, как вы нажали на кнопку «**Run**» («Пуск») на web-сайте.

В самом начале работы вашей анимации этот параметр будет иметь значение 0. Так что эта программа говорит то же самое, что и выражение `rotate (60 * 0) (rectangleSolid 100 100)`. Помните, что `*` обозначает умножение. Конечно, 60 * 0 = 0, так что выражение примет вид `rotate 0 (rectangleSolid 100 100)`. Оно рисует прямоугольник (квадрат), который вообще не повёрнут.

Но после этого проходит некоторое время. После половины секунды параметр `t` имеет значение 0.5. Программа рисует изображение, определяемое выражением `rotate (60 * 0.5) (rectangleSolid 100 100)`. Сколько будет 60 * 0.5? Конечно, 30. Так что теперь рисуемое изображение является в точности `rotate 30 (rectangleSolid 100 100)`, и оно нарисуется в виде прямоугольника, повёрнутого на 30 градусов. Но течение времени продолжается, и после истечение одной секунды параметр `t` примет значение 1, и 60 * 1 = 60, так что прямоугольник будет повёрнут на 60 градусов. После двух секунд прямоугольник будет повёрнут дважды на 60 градусов, то есть на 120 градусов. С ростом значения параметра `t` прямоугольник вращается всё круче и круче.

Хорошо, вот небольшая проверка: через сколько секунд прямоугольник полностью обернётся вокруг на 360 градусов? Понять это не так уж и сложно — достаточно найти число, которое при умножении на 60 даёт в результате 360. И это число покажет, сколько *секунд* займёт такое вращение.

А что же после этого? Вы можете повернуть что-нибудь более, чем на 360 градусов? Да, вполне. Но вы не сможете показать, что вы сделали это. Поворот на 360 градусов выглядит так, как будто бы фигуру не вращали вообще. Так что, например, если что-нибудь попытаться повернуть на 390 градусов, то это будет то же самое, что и поворот на 30 градусов. (Если вы тщательно размышляли над этим вопросом, то можете сказать, что относительно квадрата справедливо, что его невозможно различить уже при повороте на 90

градусов, поскольку он симметричен). И безотносительно, симметрична фигура или нет, поворот на 360 градусов не изменяет её вообще.

Попробуйте этот пример на web-сайте и убедитесь, что программа ведёт себя так, как мы и ожидаем.

## Пример 2. Изменение скорости

Давайте внесём совершенно незначительное изменение в нашу предыдущую программу и заставим наш квадрат вращаться быстрее:

```
import Graphics.Gloss

animation t = rotate (100 * t) (rectangleWire 100 100)
```

Как и ранее, параметр `t` принимает в качестве значения количество секунд, прошедших с момента запуска программы на исполнение. В самом начале этот параметр принимает значение 0, так что 100 * 0 = 0. Программа начинает с того же самого места, что и предыдущая. Но смотрите, что происходит: после половины секунды квадрат вращается на 50 градусов. После секунды он уже повёрнут на 100 градусов. Теперь для того чтобы сделать полный оборот квадрату потребуется примерно три с половиной секунды.

## Пример 3. Старт вращения с другого угла

Теперь давайте попробуем изменить стартовый угол для вращения нашего квадрата так, чтобы он начинал вращаться с позиции, напоминающей бриллиант:

```
import Graphics.Gloss

animation t = rotate (100 * t + 45) (rectangleWire 100 100)
```

Хорошо, что будет при `t = 0`? Легко увидеть, что это будет выражение `rotate 45 (rectangleWire 100 100)`. Так что квадрат начнёт вращаться, будучи повёрнутым на 45 градусов. Тем не менее, он будет вращаться с той же самой скоростью, что и в предыдущем примере: 100 градусов в секунду. Попробуйте и посмотрите, как это выглядит.

## Пример 4. Движение

Теперь давайте подвигаем что-нибудь вместо того, чтобы вращать его.

```
import Graphics.Gloss

animation t = translate (50 * t) 0 (circle 25)
```

Задержитесь и подумайте, что будет происходить.

Для понимания этого, давайте посмотрим на изображения, которые будут получаться с течением времени. Когда анимация только-только запущена, прошло 0 секунд, так что параметр `t` примет значение 0. Так что будет нарисовано изображение, определяемое выражением `translate 0 0 (circle 25)`. Оно не сдвинет круг ни на йоту, так что он будет находиться в центре экрана. Но после того как параметр `t` примет значение 1 (то есть

с момента запуска пройдёт одна секунда), выражение примет вид `translate 50 0 (circle 25)`, так что круг будет сдвинут немного вправо. Фактически, круг будет двигаться до тех пор, пока не заедет за границу экрана (и даже после этого он продолжит движение, просто вы его больше не будете видеть).

Попробуйте!

## Пример 5. Ускоряем движение

Вы можете придумать, как заставить круг двигаться быстрее? Если вы подумали, что для этого надо умножить параметр `t` на большее число, то вы правы.

```
import Graphics.Gloss

animation t = translate (100 * t) 0 (circle 25)
```

Круг всё так же будет начинать движение из центра (почему?), но теперь через секунду он окажется дальше в два раза, чем предыдущий. Этот круг двигается быстрее.

Мы не будем выносить эту программу в качестве отдельного примера, но как бы вы поступили, если бы вам надо было бы запустить круг не из центра, а с левого края экрана? Подсказка: такую программу надо определить практически так же, как и в примере № 3.

## Пример 6. Изменение размера

А что если вам надо, чтобы что-нибудь росло с изменением времени? Это тоже можно устроить. Вот пример:

```
import Graphics.Gloss

animation t = circle (20 * t)
```

И опять, параметр `t` принимает в качестве значения количество секунд с момента запуска программы. В самом начале ваша программа будет описываться выражением `circle (20 * 0)`, то есть то же самое, что и `circle 0`. Круг с нулевым радиусом слишком мал, чтобы его можно было увидеть, так что вы ничего и не увидите. Но через полсекунды выражение принимает вид `circle (20 * 0.5)`, то есть `circle 10`, и вы увидите круг радиусом 10. Круг будет расти: после десяти секунд од будет иметь радиус 200. Через некоторое время, если вы проявите выдержку, круг вырастет настолько, что не будет помещаться на экране.

Попробуйте эту программу и посмотрите, работает ли она так, как вы ожидали.

(Вы также могли записать определение в виде: `animation t = scale t t (circle 20)`. Вы можете объяснить, почем эти два определения работают одинаково?)

## Пример 7. Вращение вокруг другой точки

Иногда нам потребуется, что изображение (или его часть) вращались вокруг центра экрана, а не вокруг своего собственного центра. Трюк заключается в том, чтобы сначала

сдвинуть изображение, а только потом повернуть его в зависимости от времени.

```
import Graphics.Gloss

animation t = rotate (60 * t) (translate 100 0 (circleSolid 25))
```

Попробуйте определить, как будет выглядеть результат в разные моменты времени, а потом проверьте себя, запустив программу на web-сайте.

## Анимация, генераторы списков и функции

Вы, должно быть, уже заметили, что многие вещи, которые мы уже сделали с анимацией, очень похожи на то, что ранее мы делали с генераторами списков. Вы были правы!

В обоих случаях у нас были *переменные*, которые принимали различные значения, а от того получались и различные изображения. Различие заключается в том, что в случае генераторов списков мы рисовали все изображения одновременно. Теперь же мы рисуем различные изображения в разное время. Но идея остаётся той же самой: вы используете переменные (ранее при описании генераторов списков мы использовали слово «параметр») для того, чтобы представлять числа, которые мы ещё не знаем, после чего строить различные версии своего изображения, когда эти переменные начинали значить что-либо различное.

Фактически, мы уже использовали такие переменные трижды:

1. Когда мы определяли функции (такие, как функция «`awesome`»), параметры получали свои имена, которые потом использовались в определении функции.
2. В генераторах списков, когда мы использовали смешные стрелки (`<-`) для выбора имени переменной, которая будет получать различные значения.
3. И теперь при определении анимации, когда мы снова создаём функции с параметром, который теперь представляет время.

Это нечто, что постоянно будет нам встречаться. Переменные очень важны, и вам надо научиться делать операцию под названием «подстановка», когда вы видите переменную в выражении. Это то же самое, как и ответить на вопрос типа «Если `t` равно 5, то как это работает?».

## Разработка анимации сверху вниз

Когда мы работали с изображениями, мы использовали подход к разработке сверху вниз, начиная записывать определение функции «`picture`» и иногда используя новые функции, которые компьютер ещё не знает. Потом мы возвращались к таким функциям и определяли их. Это происходило до тех пор, пока программа не становилась законченной. Прекрасным в разработке сверху вниз является то, что вам не нужно думать обо ***всей*** вашей программе. Каждый раз вам надо думать только об одном небольшой кусочке.

Для анимирования мы также продолжим использование этого подхода. Новой вещью, о которой нам надо подумать, является применение параметра `t`. Вот, что мы сделаем:

- Если мы определяем часть изображения, то нам надо смотреть, будет ли эта часть

двигаться. Если да, то эту часть надо определять в виде функции с параметром `t`.
- Когда мы потом будем использовать эту функцию, мы будем передавать ей `t` в качестве параметра.

Это необходимо чётко соблюдать — если вы определяете функцию с параметром, то при *вызове* функции ей этот параметр надо обязательно передавать. Если вы определяете функцию без параметров, то при её вызове ей нельзя передавать какие-либо параметры.

Давайте посмотрим на пример. Вот программа, которая рисует часы:

```
import Graphics.Gloss

animation t = clock t

clock t = pictures [backPlate,
                    minuteHand t,
                    secondHand t]

backPlate = color (dark white) (circleSolid 250)

minuteHand t = rotate (-0.1 * t) (line [(0,0), (0,250)])

secondHand t = rotate (-6 * t) (line [(0,0), (0,250)])
```

Рассмотрим её шаг за шагом:

1. Для начала мы определили функцию «`animation`». Когда вы работаете в режиме анимации, вы всегда должны определить эту функцию, и она должна принимать один параметр `t`, который определяет, сколько секунд прошло с момента запуска программы.
2. Поскольку ранее мы передали `t` в функцию «`clock`» в качестве параметра, определять эту функцию нам также теперь необходимо с параметром. Часы состоят из трёх частей: задника, минутной стрелки и секундной стрелки. Задник не двигается, так что мы не передаём определяющей его функции `t` в качестве параметра. Однако минутная и секундная стрелки двигаются, так что им этот параметр передаём.
3. Далее мы определяем функцию для рисования задника «`backPlate`». Помните, что это просто картинка, и она не двигается, так что мы не передаём этой функции параметр `t`.
4. Теперь определяем функцию для рисования минутной стрелки «`minuteHand`». Ранее мы передавали в неё параметр, и теперь нам надо указать, что функция принимает параметр. И как в ранее показанном примере мы используем параметр `t` для определения вращения этого изображения. Мы хотим, чтобы минутная стрелка делала полный оборот в 360 градусов за один час, час состоит из 3600 секунд, так что каждую секунду нам надо поворачивать стрелку на одну десятую долю градуса. Так же мы хотим, чтобы стрелка вращалась, как это ни странно, по часовой стрелке, так что мы указываем отрицательный коэффициент при параметре `t` (помните, что положительные числа вращают изображения против часовой стрелки).
5. Наконец, мы определяем функцию «`secondHand`», которая рисует секундную стрелку. Она также ожидает параметр, так что мы его задаём. Определение выглядит точно так же, но теперь мы хотим, чтобы стрелка вращалась на 360 градусов за минуту (60 секунд). Это значит, что каждую секунду нам надо поворачивать стрелку на 6 градусов, и коэффициент

всё так же должен быть отрицательным, чтобы вращение было по часовой стрелке.

К счастью, всё это выглядит не так уж и плохо. Это всё тот же подход к разработку сверху вниз, за исключением того, что теперь мы передаём в качестве параметра время для того, чтобы изображение двигалось. При разработке вы должны себя спрашивать относительно каждой части, двигается ли она или нет, и знать точный ответ на этот вопрос.

## Домашнее задание этой недели

Заданием, над которым мы работали на текущей неделе, является разработка работающей модели Солнечной системы. Это значит, что Солнце должно стоять в центре, а вокруг него должны вращаться планеты. Некоторые замечания:

- Модель может не соблюдать масштаба. Фактически, планеты находятся так далеко, и являются такими маленькими по сравнению с расстояниями между ними, что если вы хотите точно смасштабировать реальные размеры, вы не увидели бы на экране самих планет. Так что просто выберите то, что выглядит красиво.
- Если вы амбициозны, то добавьте к модели по крайней мере один планетарный спутник, а также нарисуйте у Сатурна кольца. Подсказкой здесь является то, что ни спутники, ни кольца у Сатурна не являются такой уж сложной вещью, если вы используете разработку сверху вниз. Например, у вас должна быть функция «`saturn`», которую надо определить не просто в виде круга, а в виде двух кругов, один из которых немного сплющен.
  Теперь представьте, что вы хотите нарисовать Луну около Земли, тогда у вас должна быть функция типа «`earthSystem`», в которой рисуется Земля и Луна. Поскольку Луна движется вокруг Земли, эта функция должна принимать параметр `t`. В самой функции «`earthSystem`» наверняка будут вызываться функции «`earth`» и «`moon`». А эти функции, скорее всего, будут просто определять цветные кружки.
- Все ученики нашего класса решили, что каждый будет рисовать свою собственную Солнечную систему с различными именами планет. Это тоже прекрасно. Фактически же компьютеру всё равно, как вы называете свои функции.

Удачи! И не смущайтесь задавать вопросы в комментариях.

## Просто для веселья...

Если вам скучно, то попробуйте следующую анимацию. Это предварительный набросок того, чем мы будем заниматься на нескольких следующих неделях.

```
import Graphics.Gloss

animation t = pictures [jumprope t,
                        translate (-210) 0 stickFigure,
                        translate ( 210) 0 stickFigure,
                        jumper t]

jumprope t = scale 1.75 (1.1 * sin t) rope

rope = line [(x, 100 * cos (0.015 * x)) | x <- [-100, -75 .. 100]]

stickFigure = pictures [line [(-35, -100), (0, -25)],
```

```
                    line [( 35, -100), (0, -25)],
                    line [(  0,  -25), (0,  50)],
                    line [(-35,    0), (0,  40)],
                    line [( 35,    0), (0,  40)],
                    translate 0 75 (circle 25) ]

jumper t = translate 0 (max 0 (-50 * sin t)) stickFigure
```

И это всё на этот раз.

# Неделя 8

Добро пожаловать в описание занятий восьмой недели по обучению программированию на языке Haskell детей. Если вы только что присоединились к нам, просмотрите описания занятий на неделях 1, 2, 3, 4, 5, 6 и 7.

## Краткий обзор

Кстати, вот, чем мы занимались. Несколько недель мы изучали то, как рисовать изображения. И буквально на прошлой неделе перешли к анимации. Вот, что мы изучили:

- Функции для рисования изображений не принимают параметров, а для воспроизведения анимации — принимают.
- Эти функции для анимации принимают единственный параметр, в котором записывается количество секунд, прошедших с момента запуска программы.
- Результатом функции должно быть изображение.

Для определения функции с параметром мы просто даём ему имя и помещаем слева от знака равенства сразу после наименования функции. И после этого мы можем использовать это имя справа от знака равенства так, как будто бы это было просто число. Вот, например:

```
animation t = rotate (60 * t) (rectangleSolid 40 200)
```

Функция называется «`animation`». Параметр называется «`t`». И после знака равенства мы можем использовать имя `t` ровно так же, как и любое число. Например, мы можем умножить его на 60, после чего повернуть изображение на полученное число градусов.

Результат функции получается при помощи *подстановки*. Так, к примеру, для понимания того, что произойдёт через 5 секунд, компьютер смотрит на выражение после знака равенства и подставляет вместо всех вхождений имени `t` число 5. Это выполняется посредством следующих шагов:

1. `animation 5` — это то, с чего компьютер начинает, пытаясь понять изображение.
2. `rotate (60 * 5) (rectangleSolid 40 200)` — компьютер подставил 5 вместо `t` в выражении после знака равенства.
3. `rotate 300 (rectangleSolid 40 200)` — а это просто математика: компьютер вычисляет выражение 60 * 5 и получает значение 300.

Так что через 5 секунд компьютер нарисует прямоугольник, повёрнутый на 300 градусов. Если вы попробуете этот пример с различными числами, то вы заметите, что прямоугольник вращается на всё большее и большее количество градусов.

### Так в чём же проблема?

Мы игрались с анимацией и делали много прикольных вещей: при помощи использования параметра `t` в разных местах вы можете заставить изображения вращаться (использование в функции «`rotate`»), двигаться (использование функции «`translate`»), становиться больше или меньше (при помощи функции «`scale`» или, используя параметр `t` непосредственно при вызове таких функций библиотеки Gloss, как «`circle`»). Но вы наверняка заметили нечто обескураживающее во всех типах использования, кроме вращения, — в конце концов числа становятся слишком большими, чтобы видеть изображения на экране.

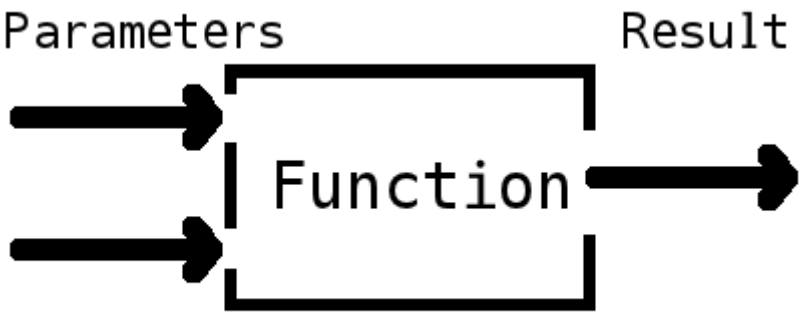
Из-за этого наши программы до сих пор не делали ничего интересного, кроме вращения изображений. Для того чтобы использовать анимацию и не сталкиваться с такими неприятными явлениями, как разрастание до огромных размеров или перемещение за границы экрана, нам необходимо познакомиться с различными видами математических функций...

## Функции, возвращающие числа

Для создания анимации необходимо определить функцию, которая получает на вход число, а возвращает изображение. Но это не единственный вид функций. Мы уже встречались с некоторыми видами, которые определены для нас в библиотеке Gloss:

- Функция «`circle`» превращает число в изображение (прямо как функция для анимации, но число обозначает что-то другое).
- Функция «`light`» превращает цвет в другой цвет.
- Функция «`rotate`» превращает число и изображение в другое изображение (которое повёрнуто).
- Функция «`polygon`» превращает список точек в изображение.

В общем, вы можете представлять функцию в виде машины, которая преобразует одни объекты в другие объекты.




Вы передаёте параметры на вход функции и получаете результат на её выходе. Различные функции требуют различное число и различные типы параметров, равно как и

возвращают вам результаты различных типов. Так что если вы хотите использовать функцию, вы должны задать себе следующие вопросы: (1) Сколько параметров функция ожидает на вход? (2) Объекты каких типов функция ожидает в качестве параметров? (3) Объект какого типа будет возвращён функцией, когда она завершит свою работу?

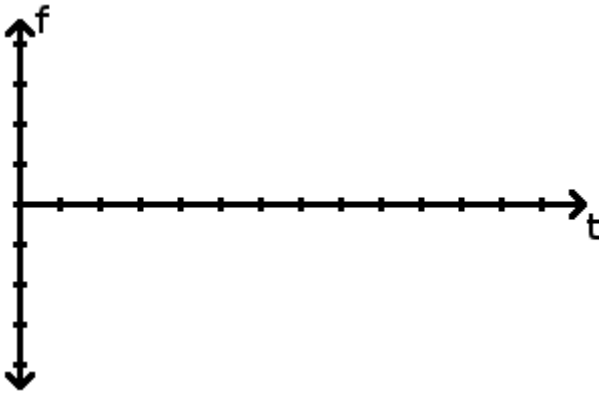
В конце концов, вы хотите создавать изображения. Так что в первую очередь вы будете задумываться о функциях, которые возвращают изображения. Но это слишком близоруко. Помните о функциях «`light`» и «`dark`»? Они ожидают на вход цвет, а возвращают тоже цвет, а не изображение. Но они всё также очень полезны при рисовании изображений, не так ли?

То же самое справедливо и относительно чисел. Даже если вы хотите закончить рисование своих изображений, вам всё равно требуется понять функции, работающие с числами, поскольку мы постоянно используем числа при рисовании. Так что на этой неделе мы потратим много времени на рассмотрение функций, работающих с числами.

## Как нарисовать функцию, работающую с числами

Когда у нас есть функция, которая принимает на вход число и возвращает тоже число, у нас есть очень удобный способ её отображения. Сперва этот способ может показаться жутким, но поверьте, что он намного облегчает жизнь. Так что вот описание того, как нарисовать функцию. Он подходит только для функций, принимающую на вход одно число и возвращающую число.

Мы начинаем с изображения двух линий — горизонтальной и вертикальной. Вот так:




Я назвал вертикальную линию *f*, а горизонтальную — *t*. Это нормально, поскольку горизонтальная линия будет представлять собой время, которое прошло с момента запуска программы, так что мы и назвали её *t*. А наименование *f* просто является сокращением слова «function» («функция»).

Далее надо приписать числа отметкам на линиях: для линии *t* это будут просто числа 1, 2, 3 и т. д. Ноль находится на пересечении двух линий, так что первая отметка после пересечения будет иметь пометку 1. Для вертикальной линии метки около отметок зависят от того, с какими числами мы работаем. Заметьте, что вертикальная линия продолжается в две стороны, то есть охватывает положительные и отрицательные числа.

Следующий шаг состоит в рисовании нескольких точек. Спросите себя, если вы передадите функции число 0, какое число она возвратит вам? Теперь нарисуйте точку прямо

над или под отметкой на горизонтальной линии, обозначающей 0 (помните, что это точка, где линии пересекаются). Как далеко сверху или снизу от горизонтальной линии надо рисовать точку зависит от числа, возвращённого функцией. После этого это же действие надо провести для чисел 1, 2 и т. д.

В конце концов, когда у вас будет набор точек, соедините их последовательно гладкой линией. Эта линия и является изображением функции.

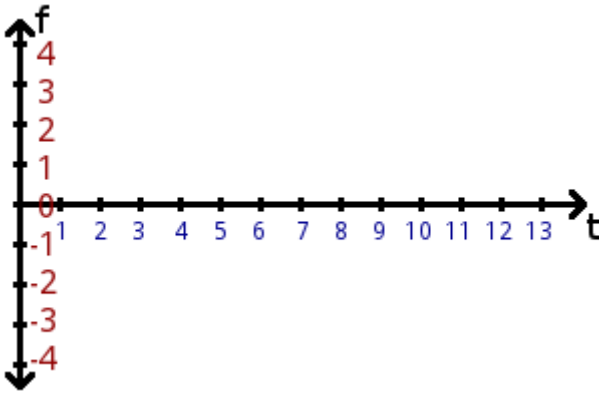
### Пример

Давайте попробуем рассмотреть пример. Рисуем функцию:

```
f t = t/2 - 2
```

Эта запись обозначает, что функция называется `f`, её единственный числовой параметр называется `t`, а возвращает она число, равное разности половины `t` и 2.

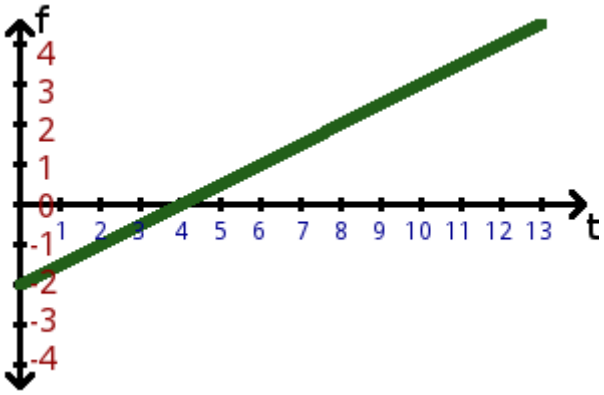
Теперь нарисуем линии, как мы говорили ранее, и пометим все отметки на них:




Следующим шагом будет вычисление некоторых значений функции и рисование их на графике в виде точек.

```
f t = t/2 - 2

f  0 =  0/2 - 2 = 0.0 - 2 = -2.0
f  1 =  1/2 - 2 = 0.5 - 2 = -1.5
f  2 =  2/2 - 2 = 1.0 - 2 = -1.0
f  3 =  3/2 - 2 = 1.5 - 2 = -0.5
f  4 =  4/2 - 2 = 2.0 - 2 =  0.0
f  5 =  5/2 - 2 = 2.5 - 2 =  0.5
f  6 =  6/2 - 2 = 3.0 - 2 =  1.0
f  7 =  7/2 - 2 = 3.5 - 2 =  1.5
f  8 =  8/2 - 2 = 4.0 - 2 =  2.0
f  9 =  9/2 - 2 = 4.5 - 2 =  2.5
f 10 = 10/2 - 2 = 5.0 - 2 =  3.0
f 11 = 11/2 - 2 = 5.5 - 2 =  3.5
f 12 = 12/2 - 2 = 6.0 - 2 =  4.0
f 13 = 13/2 - 2 = 6.5 - 2 =  4.5
```

А вот график с линией, соединяющей все эти точки:




### Как читать график

Глядя на график, вы можете получить хорошее представление о том, что делает функция. Представьте себе, что вы движетесь ко всё большим и большим числам с момента старта программы (и ведь это действительно так). И теперь видно, что эта функция будет возрастать и возрастать до бесконечности. И как вы можете видеть, она начинает свою работу со значения -2, после четырёх секунд достигает значения 0, а потом растёт и растёт.

Что же это обозначает?

- Если вы скажете «`rotate (t/2 - 2)`», то это будет обозначать, что вращение начнётся с небольшого поворота по часовой стрелке и будет вращаться против часовой стрелке до бесконечности.
- Если вы скажете «`translate (t/2 - 2) 0`», то это будет обозначать, что перемещение начнётся немного слева, а потом будет происходить всё дальше и дальше направо до бесконечности.
- И т. д.

## Практикуемся рисовать функции над числами

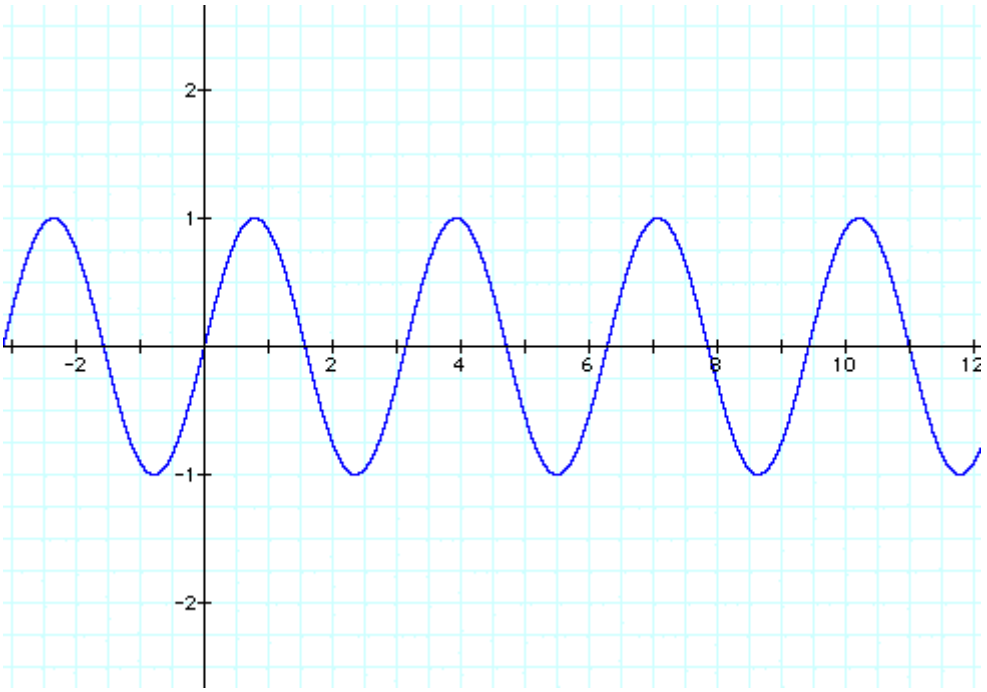
Все вместе мы практиковались рисовать функции в прошедший вторник прямо в классе. Для этого мы использовали этот файл.

## Некоторые полезные функции над числами

Вы, должно быть, заметили, что в конце представленного PDF-файла есть несколько примеров, использующих функции, с которыми мы ещё не встречались:

- Функции `min` и `max`. Эти функции принимают в качестве параметра два числа и возвращают одно из них. Так функция `min` возвращает меньшее из этих чисел, а функция `max` — большее соответственно.
- Функция `sin`. Эта функция используется для определения движения вперёд и назад. Конечно, она имеет более интересные применения, чем просто определение движения, но их изучение мы оставим до уроков тригонометрии в старших классах школы.

Хотите увидеть, как выглядит функция `sin`? Вот:




Видите, как графики функций помогают понять, что они делают?

## Домашнее задание

Поскольку школа закрывается на неделю на каникулы, до следующего описания занятий у вас будет две недели. Но не беспокойтесь — у меня есть для вас немного работы.

Вашим заданием будет разработка анимации парка развлечений. Попробуйте использовать несколько различных способов движения: вращение по кругу, движение вперёд и назад, что-то ещё, что вы можете сделать. Ученики из моего класса собираются сделать следующее:

- Скайлер: американские горки.
- Грант: прыжки на тарзанке.
- Марчелло: чёртово колесо.
- София: tilt-a-whirl.
- Сью: раскачивающееся пиратское судно.
- Я: дети, прыгающие через прыгалку (пример был в конце предыдущей недели, но я над ним ещё поработаю).

Мы планируем всё это дело объединить в одну гигантскую анимацию парка развлечений, когда мы закончим. Если у вас есть товарищи, которые тоже работают над этой темой, вы также можете объединиться. (Вы знаете, как? Всё та же разработка сверху вниз — пусть все пришлют свои разработки, а вы назовите их как-нибудь иначе, чем «`animation`». Затем отмасштабируйте их, сдвиньте в нужное положение и объедините при помощи функции «`pictures`»).

Удачи! И как обычно — пользуйтесь комментариями, чтобы задать интересующие вас вопросы.