HIGHER SCHOOL OF ECONOMICS
NATIONAL RESEARCH UNIVERSITY

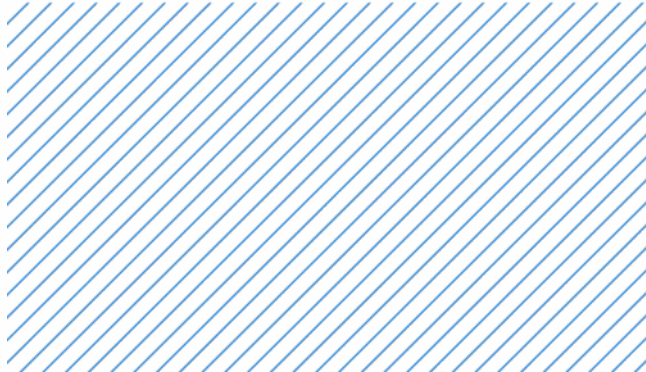
Лекция 2

Основы процедурного программирования (часть 2)

Программирование на языке Java

Роман Гуров

ВШЭ БИ 2021



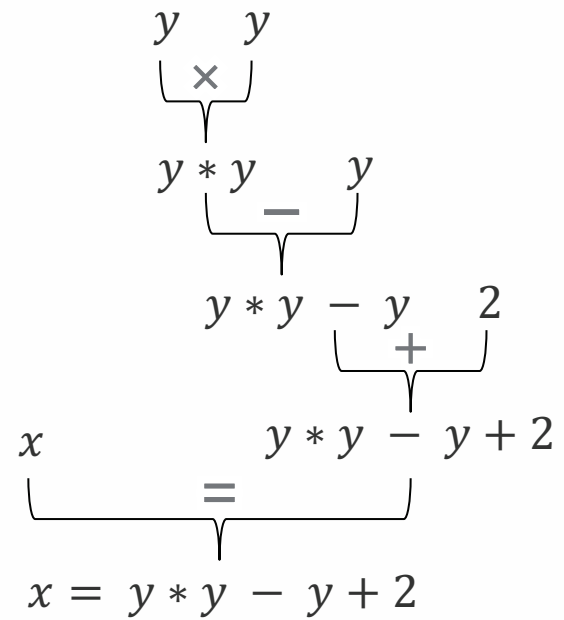
Об операторе присваивания

Оператор присваивания законно является оператором, наравне с математическими операторами

```
x = y*y - y + 2
```

Какой же у него тогда приоритет?

Больший приоритет	***, /**
Меньший приоритет	+, -



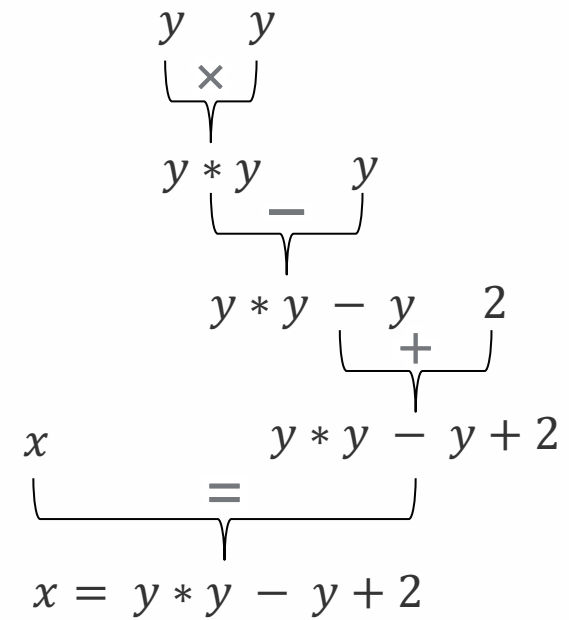
Об операторе присваивания

Оператор присваивания законно является оператором, наравне с математическими операторами

```
x = y*y - y + 2
```

Какой же у него тогда приоритет?

Больший приоритет	***, /**
Меньший приоритет	+, -
Совсем маленький приоритет	**=**



Циклы

Задача:
Перейти дорогу.

Что же всё-таки делать, если на дороге нашлись машины?

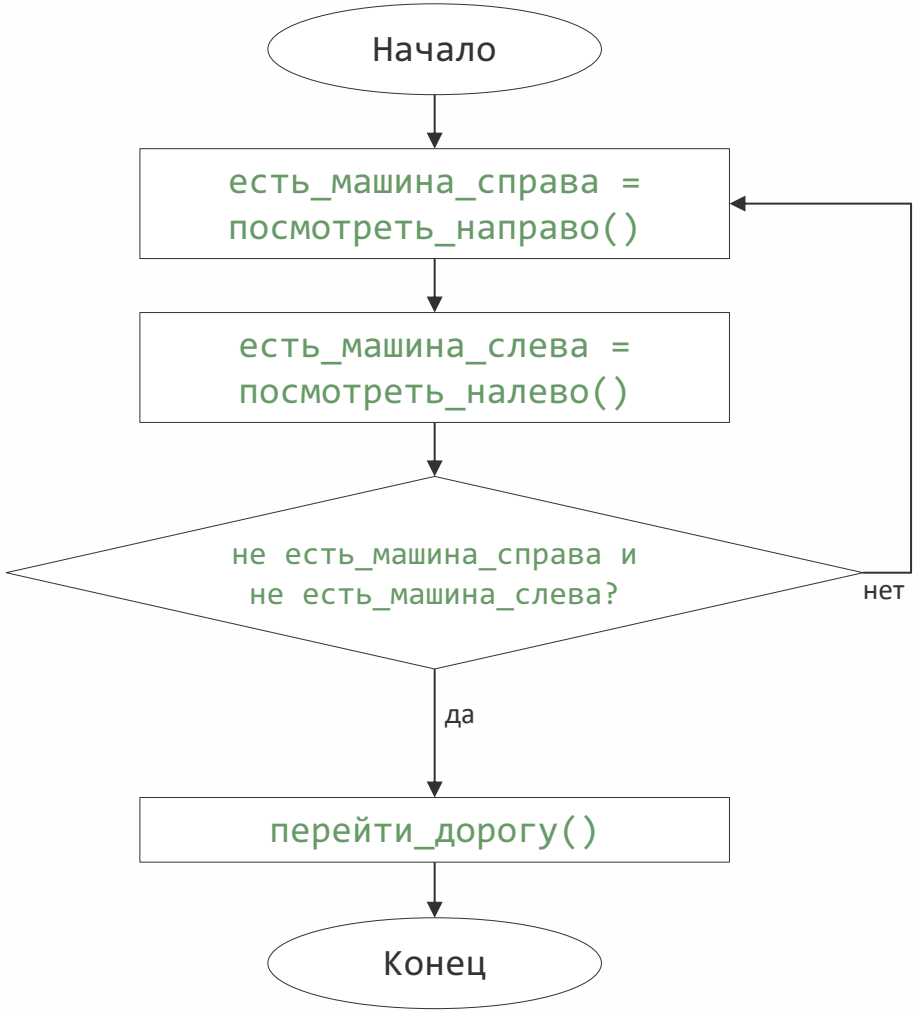
```
есть_машина_справа = посмотреть_направо()
есть_машина_слева = посмотреть_налево()

если не есть_машина_справа и не есть_машина_слева то:
        перейти_дорогу()

иначе:
        попробовать_еще_раз
```

Нужно пробовать до тех пор, пока машины не пропадут

Раньше весь код выполнялся только сверху вниз, но теперь мы захотели научиться возвращаться назад

Как языки программирования позволяют это сделать?

Циклы



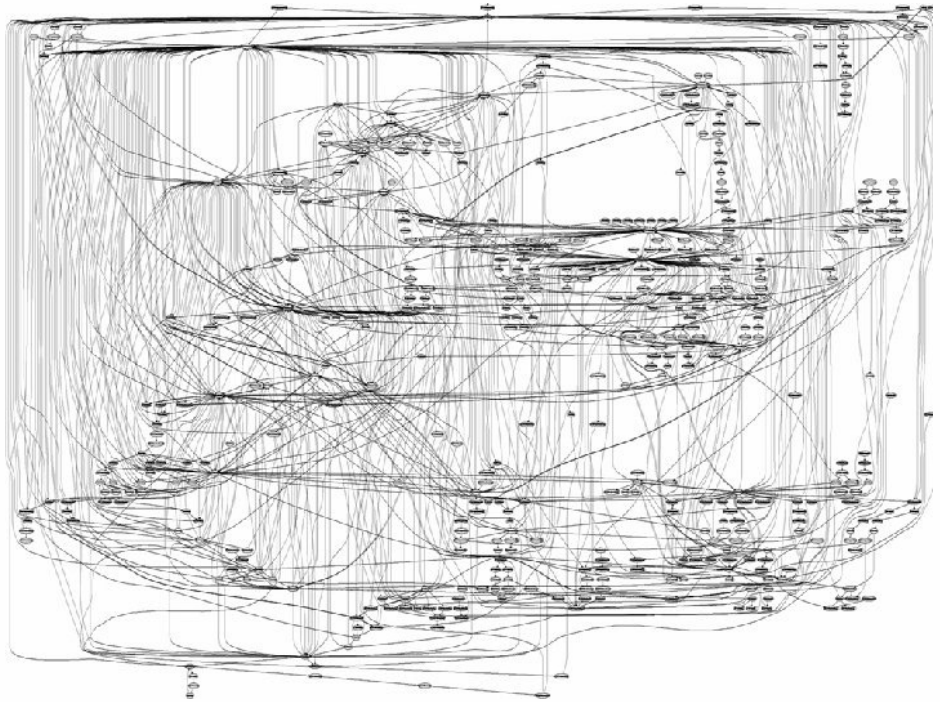
Рассмотрим блок-схему нашего решения задачи

Всё, что мы хотим – добавить в программу такой переход

Циклы

Блок-схемы дают полную свободу в переходах, стрелочку можно нарисовать куда угодно

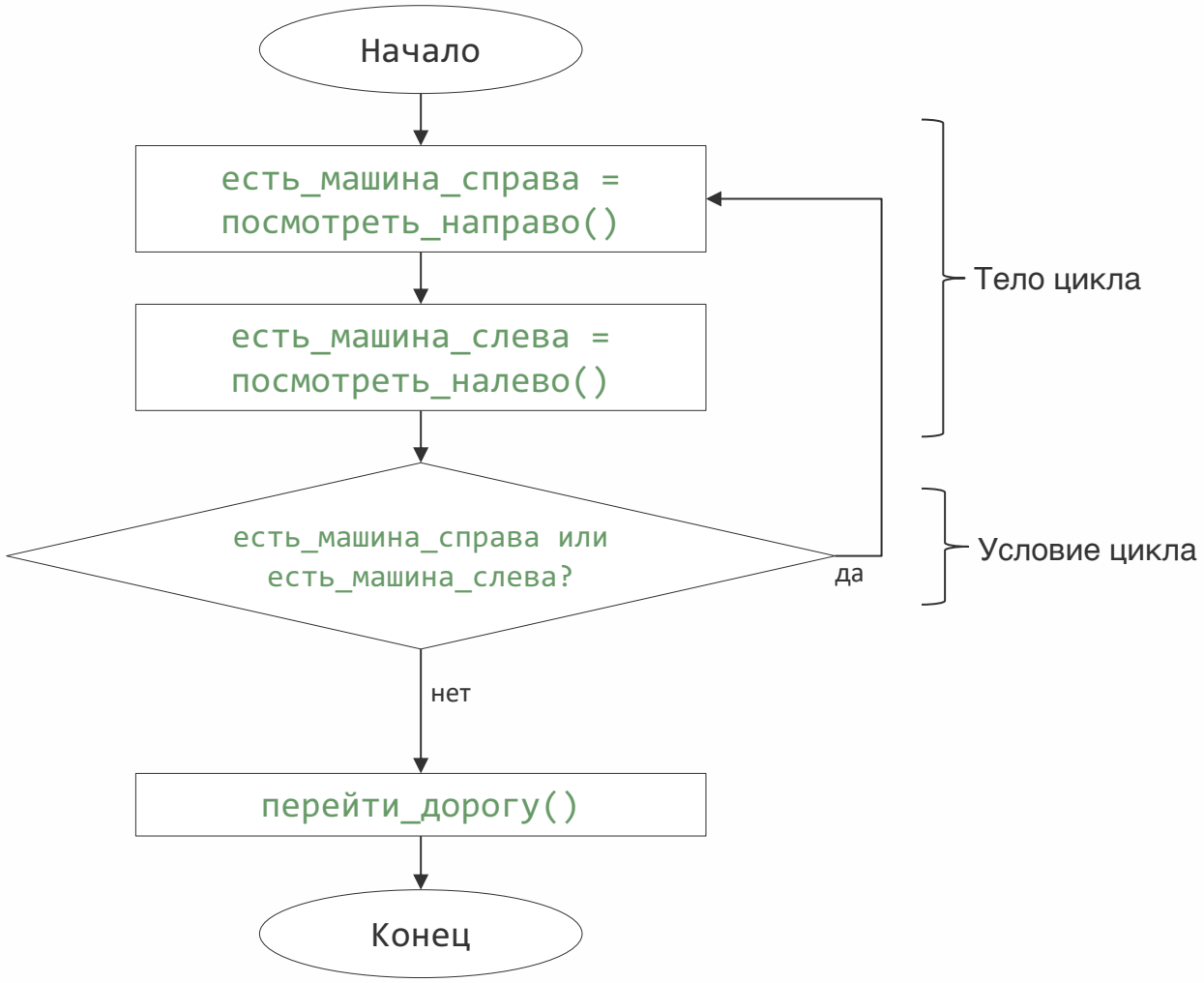
Но какой ценой?



Языки программирования ограничивают эту свободу, предоставляя только специальные конструкции – циклы

Благодаря этому прямолинейность кода сохраняется

Цикл while с постусловием



На самом деле, одну из этих конструкций мы уже использовали

У любого цикла есть тело – набор инструкций, исполнение которых хотим повторять

Также, у цикла while есть условие – выражение, по результату которого определяется, нужно ли вернуться обратно в начало тела цикла.

Цикл с постусловием работает так:

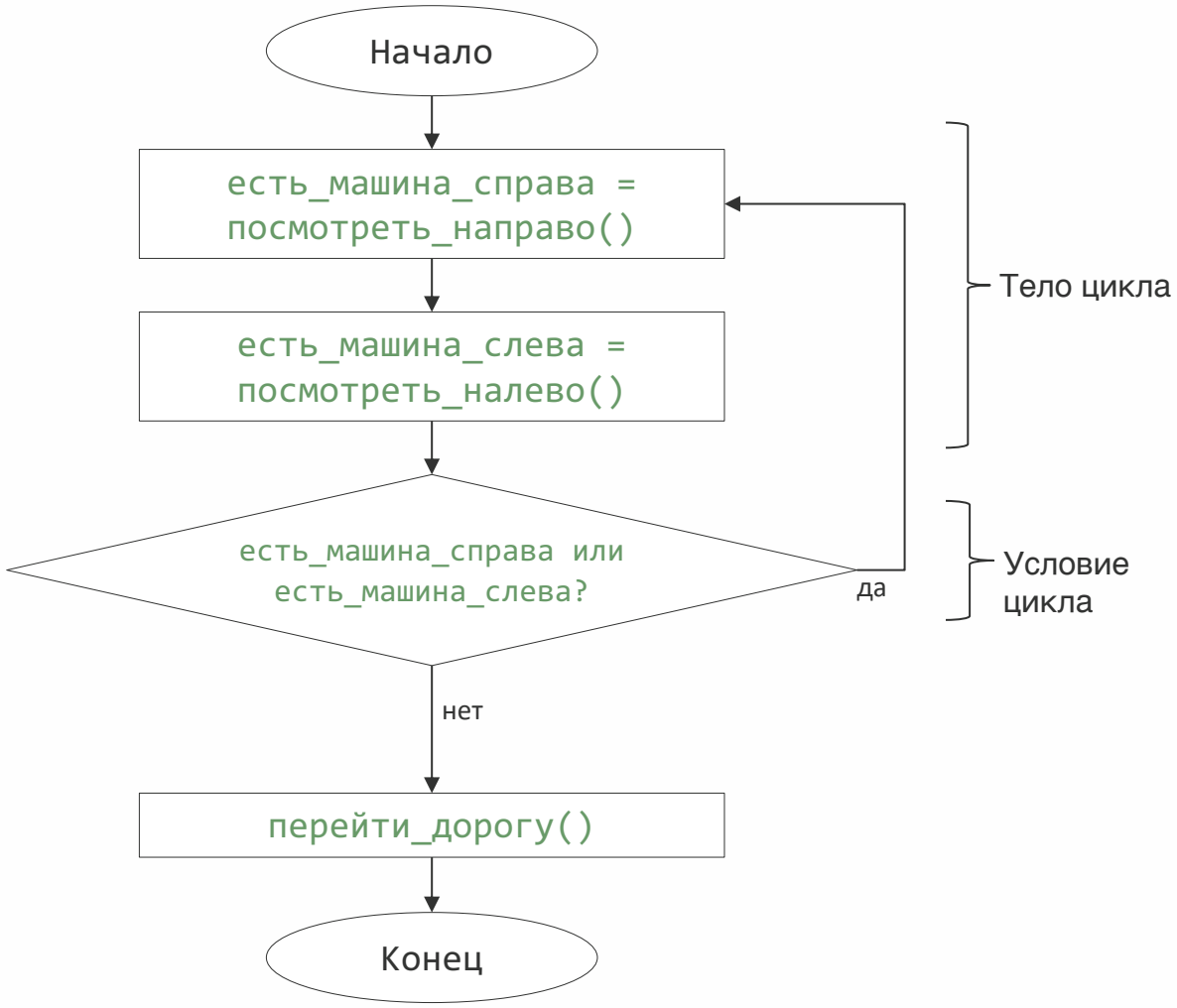
1. Выполняется тело цикла
2. Вычисляется выражение в условии
3. Если выражение дало положительный ответ, то возврат в начало цикла

В итоге, тело цикла выполняется до тех пор, пока условие положительно

```
делай:
    <тело цикла>
пока <условие цикла>
```

```
do:
    <тело цикла>
while <условие цикла>
```

Цикл while с постусловием



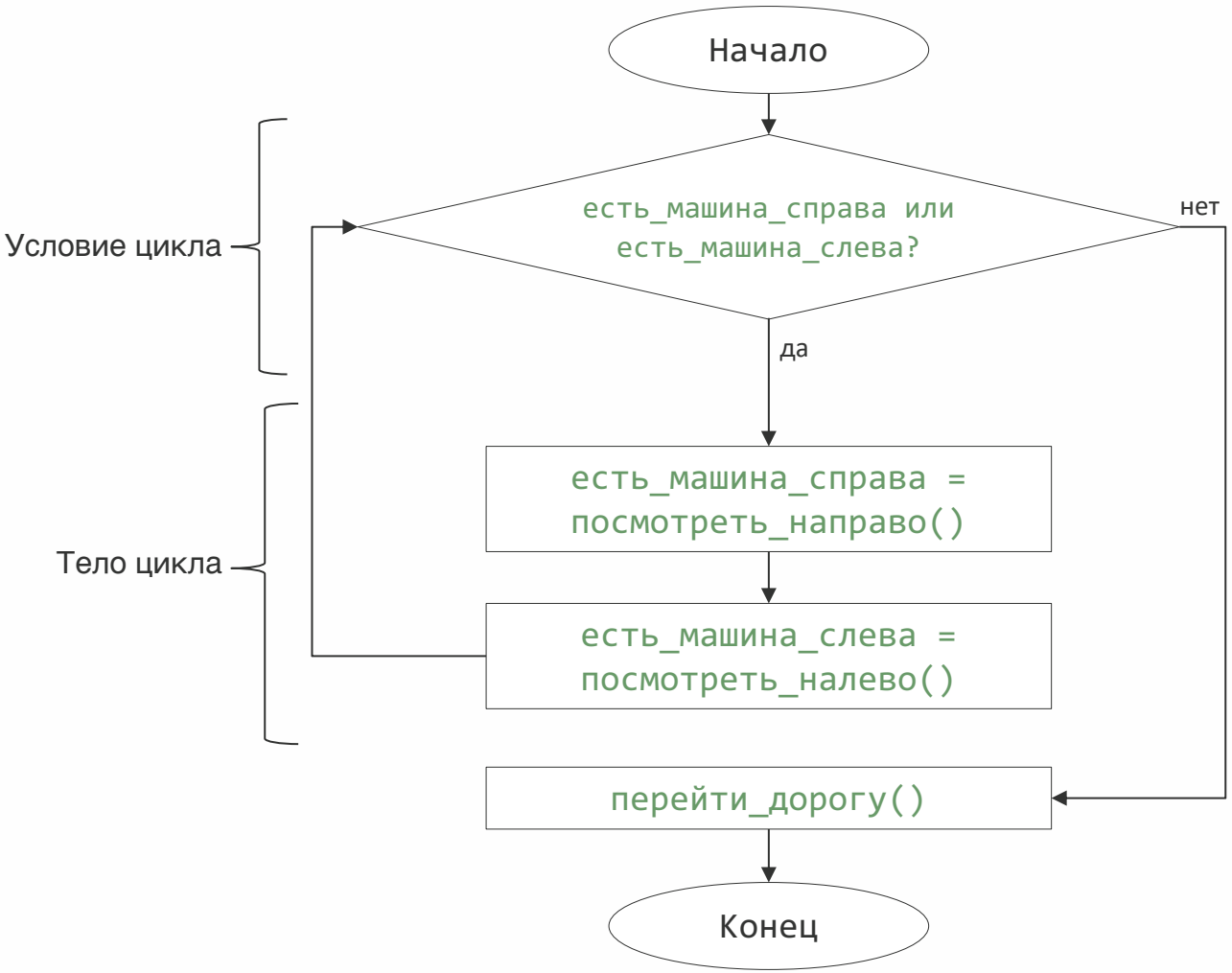
Теперь можем записать решение задачи кодом:

```
делай:
    есть_машина_справа = посмотреть_направо()
    есть_машина_слева = посмотреть_налево()
пока есть_машина_справа или есть_машина_слева

перейти_дорогу()
```

тело

условие

Цикл while с предусловием



Теперь рассмотрим цикл while с предусловием

Отличие в том, что условие проверяется до выполнения тела цикла

1. Вычисляется выражение в условии
2. Если выражение дало негативный ответ, то выход из цикла
3. Выполняется тело цикла
4. Снова пункт 1

Единственное фактическое отличие в том, что с постусловием тело гарантированно выполняется хотя бы один раз

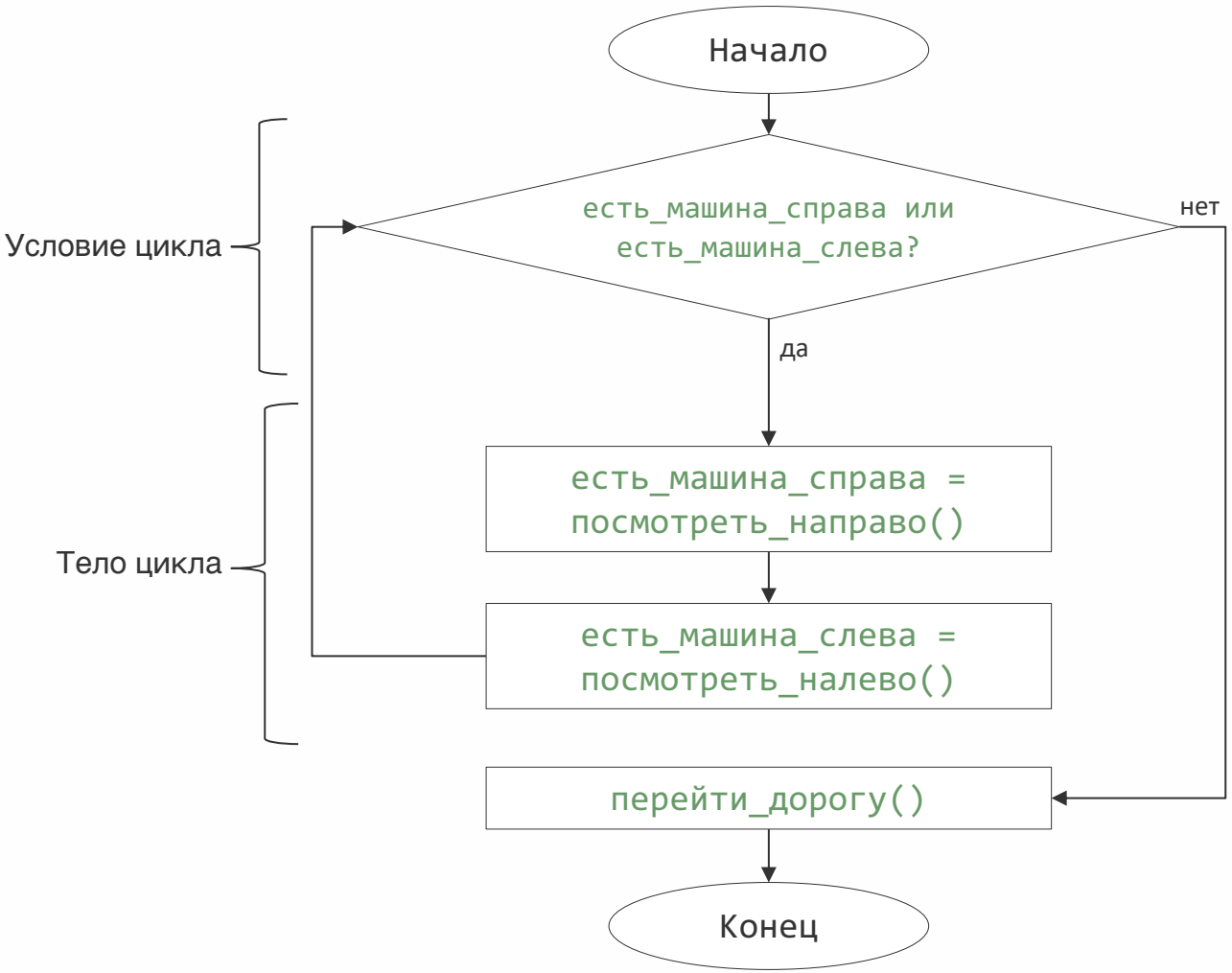
Если условие заведомо ложное, то в случае предусловия тело цикла не выполнится ни разу

```
пока <условие цикла>:
    <тело цикла>
```

```
while <условие цикла>:
    <тело цикла>
```

На практике гораздо чаще применяется именно цикл с предусловием

Цикл while с предусловием



Как это выглядит в коде:

условие
тело

```
пока есть_машина_справа или есть_машина_слева:
    есть_машина_справа = посмотреть_направо()
    есть_машина_слева = посмотреть_налево()

перейти_дорогу()
```

Один проход тела цикла называется *итерацией*

Какая проблема появилась в цикле с предусловием?

До первой итерации цикла мы вообще не знаем значения переменных `есть_машина_справа` и `есть_машина_слева`!
К этому моменту они не заданы, а значит их использование не несёт смысла

Цикл while с предусловием

Для решения проблемы придется задать какие-то начальные значения этих переменных:

```
есть_машина_справа =
есть_машина_слева =
пока есть_машина_справа или есть_машина_слева:
    есть_машина_справа = посмотреть_направо()
    есть_машина_слева = посмотреть_налево()

перейти_дорогу()
```

Цикл while с предусловием

Для решения проблемы придется задать какие-то начальные значения этих переменных:

```
есть_машина_справа = да
есть_машина_слева = да
пока есть_машина_справа или есть_машина_слева:
    есть_машина_справа = посмотреть_направо()
    есть_машина_слева = посмотреть_налево()

перейти_дорогу()
```

Или использовать цикл с постусловием.

Цикл for

Задача:
Вывести на экран числа от 1 до 10

Плохое решение:

```
напечатать 1
напечатать 2
напечатать 3
напечатать 4
напечатать 5
напечатать 6
напечатать 7
напечатать 8
напечатать 9
напечатать 10
```

- А если захотим поменять 10 на другое число?
- А если потребуется от 1 до 10000?

Хорошее решение:

```
n = 10
i = 1
пока i <= n:
    напечатать i
    i = i + 1
```

Необходимость проитерироваться по некоторой области возникает в программировании очень часто

Цикл for

Для подобных случаев существует ещё один вид цикла – цикл for

```
для (<начальное действие>; <условие цикла>; <действие в конце тела>):
    <тело цикла>
```

Эта конструкция полностью выражается через цикл while

```
<начальное действие>
пока <условие цикла>:
    <тело цикла>
    <действие в конце тела>
```

Ничего не напоминает?

```
n = 10
i = 1
пока i <= n:
    напечатать i
    i = i + 1
```

Итого:

```
n = 10
для (i = 1; i <= n; i += 1):
    напечатать i
```

Переменную `i` называют итератором цикла

Цикл for

Цикл for может принимать разные виды в разных языках программирования

Иногда он выглядит так:

```
n = 10
для i от 1 до n:
    напечатать i
```

Такой вид проще для понимания, но работает только с численными интервалами

В Java мы встретим именно сложный вариант:

```
n = 10
для (i = 1; i <= n; i += 1):
    напечатать i
```

Функции / процедуры

Вспомним математику: в математических выражениях встречаются не только обычные операторы:

$$y = \mathrm{y}_0 + \sin(2\pi t)$$

Также, математика позволяет самостоятельно определять новые функции:

$$f(x) = 2x - 1$$

Разделяем большую формулу на какие-то маленькие подформулы,
чтобы потом удобно их переиспользовать…

Что-то напоминает…

Точно! Очень похожая логика привела нас к процедурному программированию!

Функции / процедуры

Пока что, процедуры воспринимаются нами как новая инструкция, которая просто что-то делает, но не принимает новой информации и не выдает результат тому, кто её вызвал

Так почему бы нам, вдохновившись математикой, не расширить понятие процедуры?

Хотим добавить возможность:
1. Принимать аргументы
2. Возвращать ответ

В математике:

$f(x) = 2x - 1$

В коде:

```
f(x):
    result = 2*x - 1
    вернуть result
```

Использование в математике:

$k = y + f(y)$

Использование в коде:

```
ответ = y + f(y)
напечатать ответ
```

Функции / процедуры

Можно не возвращать ничего – тогда наша функция будет только выполнять действия

```
напечатать_дважды(x):
    напечатать(x)
    напечатать(x)

напечатать_дважды(3)
напечатать_дважды(7)
```

Вывод программы:

```
3
3
7
7
```

Полученную нами структуру в программировании называют именно функцией, а слово процедура применяют иногда как синоним, в случае, если функция ничего не возвращает

Но в любом случае, хоть программирование и процедурное, подпрограммы называют всё-таки функциями, потому что понятие функции шире

Функции

В общем виде функция выглядит так:

```
<имя функции>(<список аргументов>):
    <тело функции>
```

Аргументы – входные значения функции – перечисляются через запятую.
Аргументы являются переменными, значения которым передаются в момент вызова функции.

Тело функции может содержать специальные инструкции `return`, заканчивающие выполнение функции и возвращающие из неё значение

```
f(x):
    return 2*x - 1
```

Типы значений переменных

Вспомним аналогию переменных с ячейками в камере хранения

Нельзя положить в небольшую ячейку автомобиль.
И даже маленькой собачке не место в обычной ячейке, собаке нужны миски с едой и водой

Во многих языках программирования нельзя записать в переменную что попало
У каждой переменной должен быть конкретный тип значения, которая она может хранить

Примеры типов переменных:

1. Целочисленный тип – хранит целые числа – $\{\dots, -2, -1, 0, 1, 2, \dots\}$ (англ. integer, сокр. int)
   ```
   int y = 15
   ```
2. Вещественный тип – хранит дробные числа, например, 3.74 или число пи (англ. float)
   ```
   float x = 1.5 * cos(3.1415)
   ```
3. Тип текстовых строк – хранит последовательность текстовых символов (англ. string)
   ```
   string message = "Good afternoon!"
   ```

Типы значений переменных

Перепишем знакомый нам код, указав типы переменных:

```
y = 5
x = y*y
x = x - y
x = x + 2
answer = (10*x*x + 7*x - 3) / 42
print answer
```

```
int y = 5
int x = y*y
x = x - y
x = x + 2
int answer = (10*x*x + 7*x - 3) / 42
print answer
```

Важно заметить, что тип указывается только при первом использовании переменной, потому что именно в этот момент переменная создаётся

По сути, создание и присваивание – разные операции, просто мы объединили их в одну строку

```
int y
int x
y = 5
x = y*y
```

Функции с типами

Рассмотрим простой пример функции с указанными типами:

```
float f(float x, float y):
    float result = y*cos(x)
    return result
```

В общем виде функция с указанием типов выглядит так:

```
<тип возвращаемого значения> <имя функции>(<список аргументов с типами>):
    <тело функции>
```

Общий вид программы

При рассмотрении процедурного программирования упоминалось понятие *главной процедуры (точки входа)*, с которой начинается выполнение программы

Обычно, главная процедура определяется по своему стандартному названию – main()

Примерно так может выглядеть полноценная программа с точкой входа:

```
float f(float x, float y):
    float result = y*cos(x)
    return result

void main():
    float answer = 1 + f(2, 3)
    print answer
```

`void` (рус. пустота) – особый тип возвращаемого значения функции, обозначающий, что функция ничего не возвращает

Массивы

Переменная похожа на ячейку в камере хранения

Что если нам хочется взять себе в пользование целый ряд таких ячеек?

a =

124	5326	37345	34	15	-12

Массив – это переменная, только в ней сразу несколько переменных одинакового типа

При этом элементы массива упорядочены – у каждого есть свой номер.

Имя есть только у всего массива целиком, у элементов есть лишь порядковый номер в этом массиве

Массивы

Нумерация элементов начинается с нуля. Номер элемента в массиве называется его *индексом*

a =

0	1	2	3	4	5
124	5326	37345	34	15	-12

Размер массива – 6!

Можно считать, что индекс – это расстояние от самого левого элемента.
На линейке тоже нумерация чёрточек идёт с нуля

Обращение к элементу массива происходит через квадратные скобки: `a[i]`

```
print a[1]
a[2] = 0
a[3] = a[2] + a[1]
print a[3]
```

Создание массива

При создании массива нужно указать его размер:

```
int a[6]
a[5] = 1
a[4] = 2
a[0] = 3
```

Примечание: в Java создание массива выглядит немного иначе, считайте это упрощением

Массив и цикл for – лучшие друзья

Попробуем *проинициализировать* массив числами от 1 до 10:

```
int a[10]
for (int i = 0; i < 10; i += 1):
    a[i] = i + 1
```

Итерация по массиву

Задача:
- На клавиатуре вводится последовательность из n чисел. Вывести числа в обратном порядке.

Ввод:
- Сначала само число n, после этого последовательность из n чисел

Вывод:
- Те же n чисел в обратном порядке

```
int n = считать_число()
int a[n]

for (int i = 0; i < n; i += 1):
    a[i] = считать_число()

for (int i = 0; i < n; i += 1):
    напечатать a[n-i-1]
```