Photo by Marvin Meyer on Unsplash

Lab 9

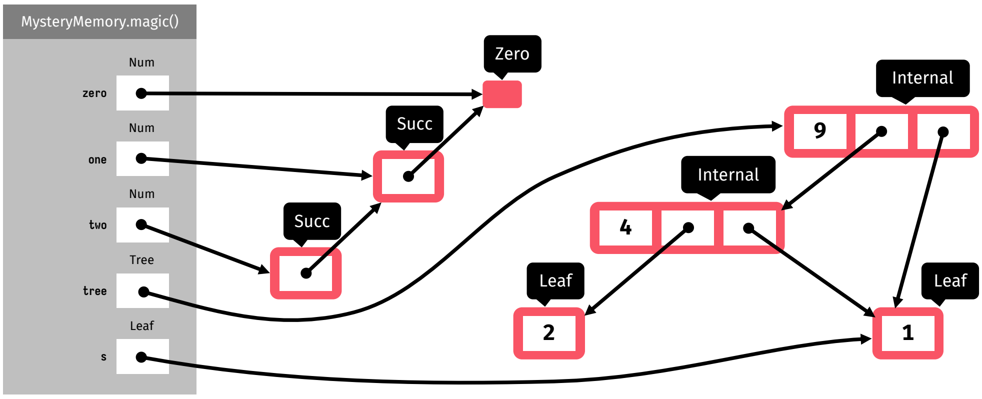
Memory Diagram to Code · CFG to Code · Random Access · Reductions with Loops · Mappings with Loops · Filterings with Loops · Pacman Game (Finish)

A. Memory Diagram to Code

We usually draw memory diagrams to explain code.

Let’s flip things around, and write the code that, when executed, will lead to a given memory state (shown in the given memory diagram).

Task A1

Class:	`lab.memory.MysteryMemory` and others you need to create
Task:	Here is a memory diagram involving the stack and the heap:  Implement method `int magic()` so it leads to the given memory diagram when called, just before it returns.

B. CFG to Code

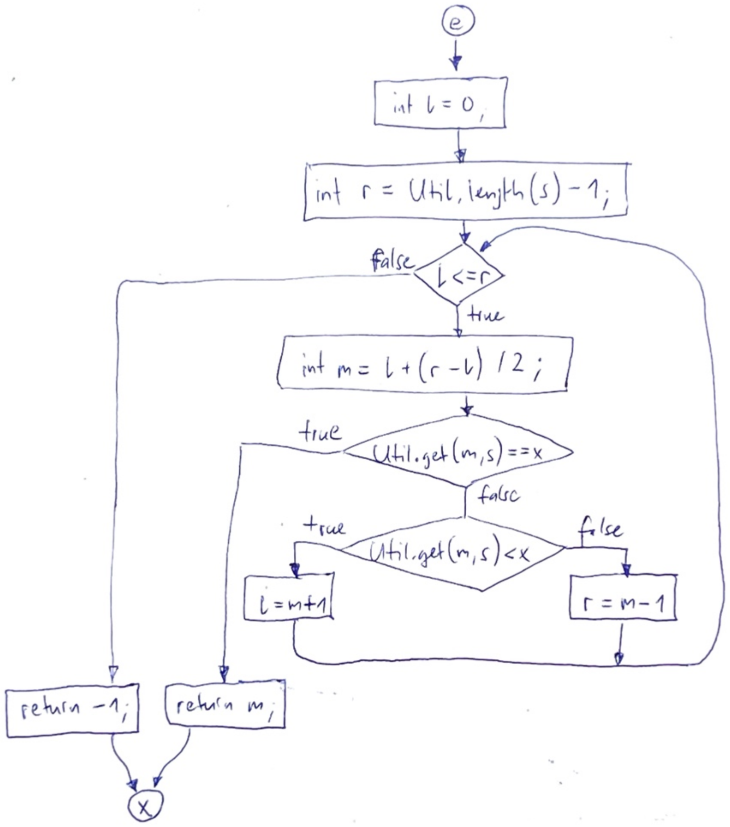
This week we learned about control-flow graphs (CFGs).

A method that is available as Java Bytecode (or in assembly and machine code in general) can easily be translated to a control-flow graph: branch instructions turn into diamonds with multiple outgoing edges, other instructions turn into rectangles with one outgoing edge to the next instruction.

However, given a control-flow graph, coming up with a corresponding structured program (using sequences of statements, conditional statements, and loop statements) can be more challenging. This is what a “decompiler” does: it takes compiled code (e.g., Java Bytecode) and produces source code, by producing the CFG and then turning the CFG into meaningful source code.

Let’s play “decompiler”!

Task B1

Class:	`lab.cfg.MysteryCFG`
Task:	 **Implement** the following method that corresponds to the above CFG. Use conditional statements and loops. **`public static`** `int magic(Sequence<Integer> s, int x)`
Run in JShell:	`MysteryCFG.magic(of(2), 2)`
Result:	`==> 0` -- for this to succeed, first implement `Util.get` in Task C1

Repetitive Computations with Loops

So far you implemented reductions, mappings, and filterings using recursion, using higher-order functions with method references, and using higher-order functions with lambdas. Now you will do so using **`for`-each-loops**, **`while`-loops**, and **`for`-loops**.

C. Random Access

Before implementing our reductions, mappings, and filterings, let's write a method that will come in handy when working with loops that use indices. Specifically, write a `get` method for sequences, so we can access an element of a sequence given an integer index.

Task C1

Class:	`lab.loops.Util`
Task:	Implement the following method in the `Util` class: `public static <E> E get(int index, Sequence<E> sequence)` **Note**: Use recursion. **Assert** that the index is in range (that it indeed refers to an existing element in the given sequence).
Run in JShell:	`Util.get(0, of(1.0, 2.0, 3.0))`
Result:	`==> 1.0`
Run in JShell:	`Util.get(0, empty())`
Result:	`AssertionError: index out of bounds`

Task C2

Class:	`lab.loops.Util`
Task:	Add the following method to the `Util` class: `public static <E> Sequence<E> reverse(Sequence<E> sequence)` **Note**: Use a `while`-loop, with `while (!isEmpty(sequence))` …. You may need this method in some of the subsequent tasks.
Run in JShell:	`println( Util.reverse(of(1, 2, 3)) )`

Output:	`3`
	`2`
	`1`

D. Reductions with Loops

Task D1

Class:	`lab.loops.Reductions`

Task:	As a baseline, **implement** the following `sum` method using recursion, like we did in the past:
	`public static double sum(Sequence<Double> values)`
	Implement the following `min` method using a `while`-loop, without using an index.
	`public static double min(Sequence<Double> values)`
	Implement the following `max` method using a `while`-loop, using an index, `Toolbelt.length`, and the `Util.get` method you implemented before:
	`public static double max(Sequence<Double> values)`
	Implement the following `and` method using a `for`-each-loop, without using an index:
	`public static boolean and(Sequence<Boolean> values)`
	Implement the following `or` method using a `for`-loop, using an index, `Toolbelt.length`, and the `Util.get` method you implemented before:
	`public static boolean or(Sequence<Boolean> values)`
	For comparison, **implement** the following `join` method using a `reduce` and a lambda:
	`public static String join(Sequence<String> values)`
	Note: use `Double.POSITIVE_INFINITY` and `Double.NEGATIVE_INFINITY` as neutral elements for `min` and `max`.

Run in JShell:	`Reductions.sum(of(1.0, 2.0, 3.0))`
Result:	`==> 6.0`
Run in JShell:	`Reductions.sum(empty())`
Result:	`==> 0.0`
Run in JShell:	`Reductions.min(of(1.0, 2.0, 3.0))`
Result:	`==> 1.0`
Run in JShell:	`Reductions.min(empty())`

Result:	`==> Infinity`
Run in JShell:	`Reductions.max(of(1.0, 2.0, 3.0))`
Result:	`==> 3.0`
Run in JShell:	`Reductions.max(empty())`
Result:	`==> -Infinity`
Run in JShell:	`Reductions.and(of(true, false, true))`
Result:	`==> false`
Run in JShell:	`Reductions.and(empty())`
Result:	`==> true`
Run in JShell:	`Reductions.or(of(true, false, true))`
Result:	`==> true`
Run in JShell:	`Reductions.or(empty())`
Result:	`==> false`
Run in JShell:	`Reductions.join(of("He", "ll", "o"))`
Result:	`==> "Hello"`
Run in JShell:	`Reductions.join(empty())`
Result:	`==> ""`

E. Mappings with Loops

Task E1

Class:	`lab.loops.Mappings`

Task:

The following mappings simply convert all elements from one type to another type.

Hint: If you get the result in the reverse order, as a first step in your method call `Util.reverse` and then operate on the reversed sequence.

As a baseline, **implement** the following `ds2is` method using recursion, like we did in the past:

```
public static Sequence<Integer> ds2is(Sequence<Double> vals)
```

Implement the following `ss2is` method using a `while`-loop, without using an index.

```
public static Sequence<Integer> ss2is(Sequence<String> vals)
```

Implement the following `is2ds` method using a `while`-loop, using an index, `Toolbelt.length`, and the `Util.get` method you implemented before:

```
public static Sequence<Double> is2ds(Sequence<Integer> vals)
```

Implement the following `ss2ds` method using a `for`-each-loop, without using an index:

```
public static Sequence<Double> ss2ds(Sequence<String> vals)
```

Implement the following `is2ss` method using a `for`-loop, using an index, `Toolbelt.length`, and the `Util.get` method you implemented before:

```
public static Sequence<String> is2ss(Sequence<Integer> vals)
```

For comparison, **implement** the following `ds2ss` method using a `map` and a lambda:

```
public static Sequence<String> ds2ss(Sequence<Double> vals)
```

Run in JShell:

```
print(
  Mappings.ds2is(of(0.1, 3.14, 0.2))
)
```

Output:

```
030
```

Run in JShell:

```
print(
  Mappings.ss2is(of("1", "3", "2"))
)
```

Output:

```
132
```

Run in JShell:

```
print(
   Mappings.is2ds(of(1, 3, 2))
)
```

Output:

```
1.03.02.0
```

Run in JShell:

```
print(
   Mappings.ss2ds(of("0.1", "3.14", "0.2"))
)
```

Output:

```
0.13.140.2
```

Run in JShell:

```
print(
   Mappings.is2ss(of(1, 3, 2))
)
```

Output:

```
132
```

Run in JShell:

```
print(
   Mappings.ds2ss(of(0.1, 3.14, 0.2))
)
```

Output:

```
0.13.140.2
```

F. Filterings with Loops

Task F1

Class:	`lab.loops.Filterings`
Task:	The following methods will work with the given the `Person` class.

As a baseline, **implement** the following `filterOlderThan` method using recursion, like we did in the past:

```
public static Sequence<Person> filterOlderThan(
  int age, Sequence<Person> ps)
```

Implement the following `filterByAge` method using a `while`-loop, without using an index.

```
public static Sequence<Person> filterByAge(
  int age, Sequence<Person> ps)
```

Implement the following `filterYoungerThan` method using a `while`-loop, using an index, `Toolbelt.length`, and the `Util.get` method you implemented before:

```
public static Sequence<Person> filterYoungerThan(
  int age, Sequence<Person> ps)
```

Implement the following `filterByLastName` method using a `for`-each-loop, without using an index:

```
public static Sequence<Person> filterByLastName(
  String lastName, Sequence<Person> ps)
```

Implement the following `filterByFirstName` method using a `for`-loop, using an index, `Toolbelt.length`, and the `Util.get` method you implemented before:

```
public static Sequence<Person> filterByFirstName(
  String firstName, Sequence<Person> ps)
```

For comparison, **implement** the following `filterByAgeRange` method using a `filter` and a lambda:

```
public static Sequence<Person> filterByAgeRange(
  int youngestInclusive, int oldestInclusive, Sequence<Person> ps)
```

Run in JShell:

```
println(
  Filterings.filterOlderThan(40,
    of(
      new Person("Jim", "Halpert", 31),
      new Person("James Morgan", "McGill", 48),
      new Person("Marquis", "Warren", 74)
    )
  )
)
```

Output:

```
Person[firstName=James Morgan, lastName=McGill, age=48]
Person[firstName=Marquis, lastName=Warren, age=74]
```

Run in JShell:

```
println(
  Filterings.filterByAge(48,
    of(
      new Person("Jim", "Halpert", 31),
      new Person("James Morgan", "McGill", 48),
      new Person("Jimmy", "McGill", 48),
      new Person("Marquis", "Warren", 74)
    )
  )
)
```

Output:

```
Person[firstName=James Morgan, lastName=McGill, age=48]
Person[firstName=Jimmy, lastName=McGill, age=48]
```

Run in JShell:

```
println(
  Filterings.filterYoungerThan(74,
    of(
      new Person("Jim", "Halpert", 31),
      new Person("James Morgan", "McGill", 48),
      new Person("Marquis", "Warren", 74)
    )
  )
)
```

Output:

```
Person[firstName=Jim, lastName=Halpert, age=31]
Person[firstName=James Morgan, lastName=McGill, age=48]
```

Run in JShell:

```
println(
  Filterings.filterByLastName("McGill",
    of(
      new Person("Jim", "Halpert", 31),
      new Person("James Morgan", "McGill", 48),
      new Person("Jimmy", "McGill", 48),
      new Person("Marquis", "Warren", 74)
    )
  )
)
```

Output:

```
Person[firstName=James Morgan, lastName=McGill, age=48]
Person[firstName=Jimmy, lastName=McGill, age=48]
```

Run in JShell:

```
println(
 Filterings.filterByFirstName("Jim",
  of(
   new Person("Jim", "Halpert", 31),
   new Person("James Morgan", "McGill", 48),
   new Person("Jimmy", "McGill", 48),
   new Person("Marquis", "Warren", 74)
  )
 )
)
```

Output:

```
Person[firstName=Jim, lastName=Halpert, age=31]
```

Run in JShell:

```
println(
 Filterings.filterByAgeRange(31, 48,
  of(
   new Person("Jim", "Halpert", 31),
   new Person("James Morgan", "McGill", 48),
   new Person("Jimmy", "McGill", 48),
   new Person("Marquis", "Warren", 74)
  )
 )
)
```

Output:

```
Person[firstName=Jim, lastName=Halpert, age=31]
Person[firstName=James Morgan, lastName=McGill, age=48]
Person[firstName=Jimmy, lastName=McGill, age=48]
```

G. Pacman Game (pt.2)

We will now continue the development of the Pacman game from the previous lab. **This second part builds on top of the first part. It's essential you do it before attempting these tasks. If you still have not done it, now it is the right moment.**

Copy the `src/main/java/lab/pacman` folder from your `lab-08` to `lab-09` at the same path.

Before writing code, make sure to read through the whole task description.

Task G1: Tiles - rendering

Task:	The most notable issue with the current implementation of our Pacman game is that there is no collision detection: our Pacman can move over the walls! To add the logic to determine whether we can step over or collide, tiles can no longer be a simple Graphic, but instead they need to become *entities* (objects). To do this, we need to perform some refactoring of the code we wrote last time: each tile has some shared behavior, and that's exactly where subtyping is useful. This means that you have to **define an interface and a number of record classes** that implement such interface. In order to preserve the functionalities that were already implemented, we begin by having our new type hierarchy be able to produce the `Graphic` of a tile. Later we will use the hierarchy to build the collision detection feature. Right now, in the `Maze` class, there is a static `charToTileGraphic` method, which produces a `Graphic`. That method takes a character and uses chained conditional operators to choose which method should be invoked to render the individual tile. With *dynamic dispatch*, it will become possible to simply invoke the same method on each different tile rather than having a chain of conditional expressions to decide which graphic to produce. We will instead use a similar chain of conditionals later in this task to construct the different tile instances. This time, it's up to you to decide how to name the interface for this type hierarchy, the name of its method (that produces the `Graphic` of the tile), the number of record classes you need to implement, their components and names. Implement an interface for a tile in the maze, and record classes that allow you to represent the different tiles and use them to simplify the `Maze.render` method by using dynamic dispatch.

	Hint: when implementing a hierarchy of classes and interface, it's a good practice to put them in their own package (e.g., `lab.pacman.game.tile`) so that the code is more isolated and easier to re–use. Once you have declared the interface and implemented all the classes, you need to instantiate them. To do that, you need a "factory". A factory class is used to create instances of a given interface depending on some input value. This avoids exposing the concrete classes to the clients, only the interface. This is very useful when developing bigger projects (especially when working in teams) as it allows you to have a clearer separation between your public API, which other people will rely on, and the implementation details of such API. Similarly to the `lab.pacman.game.heading.HeadingsFactory` class, create a factory class for your tiles with a method that given a char, instantiates the appropriate class. You will then use the newly created factory class in the `lab.pacman.Main.readMazeDescriptor` method to obtain, rather than an instance of `Sequence<Sequence<Character>>` an instance of `Sequence<Sequence<`*`NameOfYourNewTileInterface`*`>>`. Finally, now that you have concrete tile objects, it's a good idea to store them as components in the Maze class, so we will be able to do more things with them in the next tasks. This also implies converting the `Maze.render` method from a static method to an instance method now that it will no longer receive an argument of type `Sequence<Sequence<Character>>`, but rather will use the new tiles component of your maze.
Result:	Once you have completed this task, you should be able to run the game again and notice no difference when playing the game. This is what a refactoring is: improving the quality of your code while preserving all the functionalities that are exposed to the user. And we paved the way for implementing the collision detection.

Task G2

Task:	It is now time to implement a really important feature for our pacman game: maze collision detection: this way your pacman will be constrained to move within the maze. A simple way to implement this is by checking whether pacman can step on a given tile or not. Pacman can step on the "floor" tiles; it should not be able to step over walls. Given that we implemented an interface that represent the different tiles in the maze, we can put to good use the refactoring we did in the previous task to define which tiles pacman is allowed to step on. Then,

	with dynamic dispatch we could simply ask the tile of the position pacman wants to move towards and see if it allows such movement. Add a new method to the interface you created in the previous task that defines whether it's possible or not to step on such tile. Then, modify the computation of the next position of the pacman so that it not only checks whether the next step is within the boundaries of the maze, but also whether it is *landing* on a tile that can be stepped on. Finally, you might want to modify the value of the `INITIAL_PACMAN` constant in the `Configuration` class so that it defaults to a position that is not a wall (e.g., `(26, 13)`).
Result:	Once you have completed this task, you should be able to run the game and the movements of pacman should be also constrained by the inner walls of the maze, not only the outer bounds.

In the upcoming tasks, we'll continue refactoring the code to introduce new features and make your code better. Because of this, it is useful to write tests to ensure that the changes you make don't introduce *regressions* (a functionality becoming broken due to some change that may or may not be directly related to it).

Task G3

Task:

Another important missing component of the game is the `Ghost`.

`Ghost` is similar to pacman, but it has different movement logic. Instead of having its heading controlled by the user through keyboard input, the ghost follows some algorithmic rule.
To keep the task simple, the *evolution* of the ghost should be as follows:

- It should always proceed in the direction it's currently heading.
- When it hits a wall, the heading is changed randomly.

Declare and implement a `Ghost` class, which has the same functionality as the `Pacman` class except for the following methods having different implementations:

- `Ghost.render` should render a ghost and not a pacman. This also means that Ghost does not need a mouth angle component. Can you think of another render-related component that could be useful to have instead?
- Ghost should not have a turn method, instead it should compute the next heading depending on the next position when *evolving*.

To select a new `Heading` *randomly*, you may add (and then use) a new **static** method inside the `HeadingsFactory` class, which takes an instance of `Random` as its only parameter (this `Random` instance could be stored as a component in the `Ghost` class). Using the **instance** method `Random.nextInt` with arguments `0,4`, you can obtain a random integer in range `[0, 4)`. Depending on the randomly generated number, return a different `Heading` instance between `North`, `South`, `East` and `West`.

Once you have implemented the `Ghost` class, create an `INITIAL_GHOST` constant value. The ghost should be positioned at `Position(11, 11)` by default.

Now add a `Ghost` component to the `Game` record class and update all the calls to the constructor of this class and invoke the `evolve` method on the `Ghost` where appropriate, just like `Pacman`.

Finally, adapt the `Game.render` method so that the `Ghost` is rendered as well. To do this, add a `placeGhost` method which takes a `Ghost` instead of a `Pacman`. Use the result of the evaluation of the invocation of `place` as the argument of type `CartesianWorld` for the `placeGhost` function.

Result:

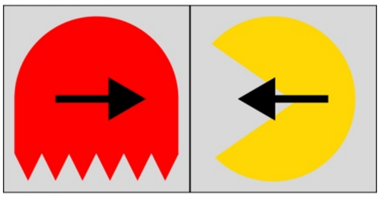
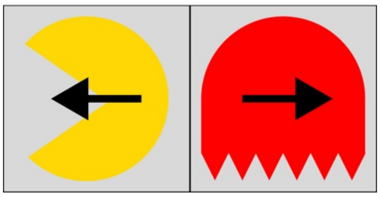
Once you have completed this task, you should be able to run the game. The ghost should be rendered, and it should move on its own around the maze. The Pacman's behavior is unchanged.

You could now write tests for the newly created Ghost, to ensure that whenever you'll change its implementation details in future tasks the behavior remains consistent.

Task G4

Task:	You may have noticed how Pacman and Ghost have some similarities: they both have several public methods that have the exact same signature and return type. Can you identify such methods? **Note**: also consider the nullary getter methods for the record components. The fact that there's a considerable amount of shared behavior between conceptually related entities provides a good opportunity to introduce some more subtyping. Now declare an interface that defines those shared methods and make `Pacman` and `Ghost` implement it, similarly to what was done during the lectures with the `Circle` and `Square` record classes and the `Shape` interface (Workbook 6). Now look at the code in your `Game` class: identify and remove some code duplication by using a shared interface rather than the specific `Pacman` and `Ghost` classes. Finally, now that you have yet another type hierarchy, it would be a good idea to move all the classes that belong to it to a sub–package just like for the tiles and heading type hierarchies. Move the interface and the two classes and make sure the imports are correctly updated (this should be done automatically by the IDE for you).
Result:	Once you have completed this task, you should be able to run the game again and play it as before. This refactoring reduced code duplication. And it would be essential if we were to have, say, multiple ghosts.

Task G5

Task:	This last task asks you to implement the "losing condition" of the game: that is, whenever the `Ghost` *eats* the `Pacman`, the player loses, and the game is reset.
	Beware: there are two possible situations in which a game may be considered lost:
	1. `Pacman` and `Ghost` are on the same tile (same position);
	2. `Pacman` and `Ghost` have opposing headings but are beside each other: in this situation the *next* `Position` of the `Ghost` is equal to the *current* `Position` of the `Pacman`. The Figure below illustrates the situation that we want to avoid by implementing this check:
	`Tick: n`
	
	`Tick: n+1`
	
	The checks for whether the game is lost should happen during the *evolution* of the `Game`. In case of a loss, the next game should revert to the **initial state** of the `Game` so it may start anew.

Result:	Once you have completed this task, you should be able to run the game and when the ghost eats the pacman, the game should be reset allowing you to play again as if you just opened the game the first time.