Principles of Programming Languages
Course Notes

Thomas Wies

September 12, 2023

Preface

This document contains the lecture notes for the NYU undergraduate course CSCI-UA.0480-055, "Principles of Programming Languages", in fall 2023. The document will be extended throughout the semester. So please stay tuned!

Course Summary

Computing professionals have to learn new programming languages all the time. This course teaches the fundamental principles of programming languages that enable you to learn new languages quickly and help you decide which one is best suited for a given task.

We will explore new ways of viewing computation and programs, and new ways of approaching algorithmic problems, making you better programmers overall. The topics covered in this course include:

- recursion and induction
- algebraic data types and pattern matching
- higher-order functions
- continuations and tail recursion
- programming language syntax and semantics
- type systems
- monads
- objects and classes

We will explore this material by building interpreters for programming languages of increasing complexity. The course will thus be accompanied by extensive programming assignments. We will use the programming language Scala for these assignments, which you will also learn in this course.

Contents

List of Figures

Chapter 1

Scala Basics

1.1 Getting Started

In the following, we assume that you have installed sbt and Intellij Idea with the Scala plugin. If you have not yet done so, please do it now. You can find a link to the installation instructions on the course web site on Brightscape.

1.1.1 Compiling and Running Scala Applications

Compiling and running Scala applications works similar to Java. For example, you can open a text editor and type in the following Scala code:

```
package greeter

object Hello extends App:
  def main(args: Array[String])
    println("Hello␣World!")
```

This code creates an object `Hello` in the package `greeter`. The object `Hello` contains a method called `main`, which means that `Hello` can serve as the entry point of a Scala application that calls `main` upon start, providing the command line arguments via the parameter `args`. When this application is started, the object `main` prints the message `"Hello␣World!"` on standard output.

The above code is roughly equivalent to the following Java code:

```
package greeter;

public class Hello {
  public static void main(String[] args) {
    System.out.println("Hello␣World!");
  }
}
```

You can safe the Scala code, say, in a file called `Hello.scala`, and then compile it with the Scala compiler. To do this, open a command prompt, go to the directory where you saved the file, and type `scalac Hello.scala`. This will create a file `Hello.class`, which contains the compiled byte code of the object `Hello`.

To execute the program, type `scala greeter.Hello` in your terminal. This will start the Scala runtime environment, which will execute the byte code in `Hello.class` using the Java virtual machine. You should see the message `"Hello␣World!"` printed in your terminal.

If you are using the Intelij Idea IDE, you can import the `in-class-code` project following the instructions provided in the `README.md` file of the repository. The repository contains a file `ScalaGreeter.scala`, which you can find in the package `popl.class03`. To compile and run the application, right-click on the file and select “Run 'ScalaGreeter' ”. This should print `"Hello Scala"` in the output view at the bottom of the window.

1.1.2 The Scala REPL and Worksheets in the IDE

If you want to experiment with the Scala language, it is quite cumbersome to write an extra application for every small code snippet that you would like to execute. To make life easier, Scala provides a useful tool called a read-eval-print loop, or *REPL* for short. The Scala REPL is essentially a command line calculator on steroids. It allows you to type arbitrary Scala code in a terminal. The code is then evaluated and the result of the evaluation is printed in the terminal.

To start the Scala REPL, open a terminal and execute `scala`. This will start the REPL and a message similar to the following should appear:

```
Welcome to Scala 3.3.0 (18.0.2-ea, Java OpenJDK 64-Bit Server VM).
Type in expressions for evaluation. Or try :help.

scala>
```

Now, you can type a Scala expression. For example typing 3 + 4 yields

```
scala> 3 + 4
val res0: Int = 7
```

You can exit the REPL by typing `:quit` or by pressing `Ctrl-d` (respectively, `Cmd-d` on OS X).

If you only installed sbt and Intelij Idea, you can start the REPL by typing `sbt console` in a terminal. Intelij Idea provides a feature similar to the REPL, called *Worksheets*. You can use this feature as follows:

1. Start Intelij Idea. (In the following, I assume you have previously imported the `in-class-code` project with the `popl` package.)
2. Right-click the `popl` package in the package explorer and choose New/Scala Worksheet. Name the worksheet “MyWorksheet” and press Finish.

3. An empty Scala source file called `MyWorksheet.sc` will open in the editor window.

4. You can now type Scala expressions in the editor window. Each time you click the "Evaluate Worksheet" button at the top of the editor view (the button with the green arrow icon), the expressions in the file are evaluated and the result of the evaluation appear in a separate view next to the editor.

5. If you click the 'Show worksheet settings' button (the button with the wrench icon), you can set the worksheet to Interactive Mode, which will cause all expressions to be evaluated automatically as you type.

1.2 Scala Crash Course

In the following, we assume that you have started the Scala REPL. Though, (almost) all of these steps can also be done in a worksheet.

1.2.1 Expressions, Values, and Types

After you type an expression in the REPL, such as `3 + 4`, and hit enter:

```
scala> 3 + 4
```

The interpreter will print:

```
val res0: Int = 7
```

This line includes:

- the keyword **`val`**, indicating that you have defined a new value resulting from evaluating the expression.
- an automatically generated name `res0` that is *bound* to that new value,
- a colon `:`, followed by the type `Int` of the expression,
- an equals sign `=`,
- the value `7` resulting from evaluating the expression.

The type `Int` names the class `Int` in the package `scala`. Packages in Scala partition the global name space and provide mechanisms for information hiding, similar to Java packages. Values of class `Int` correspond to values of Java's primitive type `int` (Scala makes no difference between primitive and object types). More generally, all of Java's primitive types have corresponding classes in the `scala` package.

We can reuse the automatically generated name `res0` to refer to the computed value in subsequent expressions (this only works in the REPL but not in a worksheet):

```
scala> res0 * res0
val res1: Int = 9
```

Java's ternary conditional operator `? :` has an equivalent in Scala, which looks as follows:

```
scala> if res1 > 10 then res0 - 5 else res0 + 5
val res2: Int = 2
```

In addition to the `? :` operator, Java also has if-then-else statements. Scala, on the other hand, is a functional language and makes no difference between expressions and statements: every programming construct is an expression that evaluates to some value. In particular, we can use if-then-else expressions where we would normally use if-then-else statements in Java.

```
scala> if res1 > 2 then println("Large!")
       else println("Not␣so␣large!")
Large!
```

Note that the result value is not automatically bound to a name in this case. The if-then-else expression still evaluates to the value `()`, which is of type `Unit`. This type indicates that the sole purpose of evaluating the expression is the side-effect of the evaluation (here, printing a message). In other words, in Scala, statements are expressions of type `Unit`. Thus, the type `Unit` is similar to the type void in Java (which however, has no values). The value `()` is the only value of type `Unit`.

1.2.2 Names

We can use the **`val`** keyword to give a user-defined name to a value, so that we can subsequently refer to it in other expressions:

```
scala> val x = 3
val x: Int = 3
scala> x * x
val res0: Int = 9
```

Note that Scala automatically infers that `x` has type `Int`. Sometimes, automated type inference fails, in which case you have to provide the type yourself. This can be done by annotating the declared name with its type:

```
scala> val x: Int = 3
val x: Int = 3
```

A **`val`** is similar to a `final` variable in Java. That is, you cannot reassign it another value:

```
-- [E052] Type Error: -----------------------------------------------------------
1 |x = 5
  |^^^^^
  |Reassignment to val x
```

Scala also has an equivalent to standard Java variables, which can be reassigned. These are declared with the `var` keyword

```
scala> var y = 5
var y: Int = 5
scala> y = 3
y: Int = 5
```

The type of a variable is the type inferred from its initialization expression. It is fixed throughout the lifetime of the variable. Attempting to reassign a value of incompatible type results in a type error:

```
-- [E007] Type Mismatch Error: ------------------------------------------------
1 |y = "Hello"
  |    ^^^^^^^
  |    Found:    ("Hello" : String)
  |    Required: Int
```

For the time being, we will pretend that variables do not exist. Repeat after me: `val`s are gooood! `var`s are baaaad!

1.2.3 Functions

Here is how you write functions in Scala:

```
scala> def max(x: Int, y: Int): Int =
         if x > y then x else y
def max(x: Int, y: Int): Int
```

Function definitions start with `def`, followed by the function's name, in this case `max`. After the name comes a comma separated list of parameters enclosed by parenthesis, here `x` and `y`. Note that the types of parameters must be provided explicitly since the Scala compiler does not infer parameter types. The type annotation after the parameter list gives the result type of the function. The result type is followed by the equality symbol, indicating that the function returns a value, and the body of the function which computes that value. The expression in the body that defines the result value is enclosed in curly braces.

If the defined function is not recursive, as is the case for `max`, the result type can be omitted because it is automatically inferred by the compiler. However, it is often helpful to provide the result type anyway to document the signature of the function.

Once you have defined a function, you can call it using its name:

```
scala> max(6, 3)
val res3: Int = 3
```

Naturally, you can use values and functions that are defined outside of a function's body in the function's body:

```
scala> val pi = 3.14159
val pi: Double = 3.14159

scala> def circ(r: Double) = 2 * pi * r
def circ(x: Double): Double
```

You can also nest value and function definitions:

```
scala> def area(r: Double) =
         val pi = 3.14159
         def square(x: Double) = x * x
         pi * square(r)

def area(Double): Double
```

Note that the scope of the body of a function definition is determined automatically by the indentation level. For instance, the following code does not compile because the last line is no longer interpreted to be part of the body of the function `area`:

```
scala> def area(r: Double) =
         val pi = 3.14159
         def square(x: Double) = x * x
       pi * square(r)
-- [E006] Not Found Error: ---------------------------------------------------
4 |pi * square(r)
  |^^
  |Not found: pi
```

If you have a longer function definition, it can be helpful to mark the end of the function explicitly using an **end** marker, followed by the name of the function:

```
def area(r: Double) =
  val pi = 3.14159
  def square(x: Double) = x * x
  pi * square(r)
end area
```

Recursive functions can be written as expected. For example, the following function `fac` computes the factorial numbers:

```
scala> def fac(n: Int): Int = if n <= 0 then 1 else n*fac(n-1)
def fac(n: Int): Int

scala> fac(5)
val res4: Int = 120
```

1.2.4 Scopes

You can use curly braces `{ ... }` to create block scopes. Scala's scoping rules are almost identical to Java's:

```
val a = 5
// only a in scope
{
  val b = 4
  // b and a in scope

  def f(x: Int) =
    // f, x, b, and a in scope
    a * x + b

  // f, b, and a in scope
}
// only a in scope
```

There are two difference to Java, though. First, the scope of a name extends the entire block in which it is defined. Using a name before its definition leads to an error:

```
val a = 3
{
  val b = a // Refers to 'a' defined on the next line.
  val a = 4 // Does not compile.
}
```

However, unlike Java, Scala allows you to redefine names in nested scopes, thereby shadowing definitions in outer scopes.

```
val a = 3
{
  val a = 4 // Shadows outer definition of a.
  a + a     // Yields 8
}
```

As in Java, you cannot redefine a name in the same scope:

```
val a = 3
val a = 4 // Does not compile.
```

1.2.5 Tuples

Scala provides ways to create new compound data types without requiring you to define simplistic data-heavy classes. One of the most useful of these constructs are *tuples*. A tuple combines a fixed number of items together so that they can be passed around as a whole. The individual items can have different types. For example, here is a tuple holding an `Int` and a `String`:

```
scala> val p = (1, "banana")
val p: (Int, String) = (1, "banana")
```

and here is a tuple holding three items: two `String`s and a `Double` value:

```
scala> val q = ("apple", "pear", 1.0)
val q: (String, String, Double) = (apple, pear, 1.0)
```

To access the items of a tuple, you can use method `_1` to access the first item, method `_2` to access the second, and so on:

```
scala> p._1
val res5: Int = 1

scala> p._2
val res6: String = banana
```

Additionally, you can assign each element of the tuple to its own **`val`**:

```
scala> val (fst, snd) = p
val fst: Int = 1
val snd: String = banana
```

Be aware that tuples are not automatically decomposed when you pass them to functions:

```
def f(x: Int, s: String) = x

f(p._1, p._2) // Works.
f(p) // Does not compile.

def g(p: (Int, String)) = p._1

g(p) // Works.
g((1, "banana")) // Works.
g(1, "banana") // Works.
```

1.3 Recursion

Recursion will be our main device for expressing unbounded computations. In the following, we study how recursive functions are evaluated. We will further see that there is a close connection between certain recursive functions and loops in imperative programs.

1.3.1 Evaluating Recursive Functions

Consider the following function which computes the sum of the integer values in the interval given by the parameters `a` and `b`.

```
def sum(a: Int, b: Int): Int =
  if a < b then a + sum(a + 1, b) else 0
```

How are calls to such functions evaluated? Conceptually, we can think of the evaluation of a Scala expression as a process that rewrites expressions into simpler expressions. This rewriting process terminates when we obtain an expression that cannot be further simplified, e.g., an integer number. Expressions that cannot be simplified further are called *values*. Concretely, if we have a function call such as `sum(1 + 1, 0 + 2)`, we proceed as follows to compute a value using rewriting:

- First, we rewrite the call expression by rewriting the arguments of the call until they are reduced to values. In our example, this step yields the simplified call expression `sum(2, 2)`.
- Next, we replace the entire call expression by the body of the function. At the same time, we replace the formal parameters occurring in the function body (i.e., the occurrences of `a` and `b` in the example) by the actual arguments of the call. In our example, this step yields the expression

  ```
  if 2 < 2 then 2 + sum(2 + 1, 2) else 0
  ```

- Finally, we continue rewriting the function body recursively in the same manner until we obtain a value that cannot be simplified further. In our example, this process eventually terminates, producing the result value `0`.

Here is how we compute the value of `sum(1, 4)` using rewriting:

```
sum(1, 4)
-> if 1 < 4 then 1 + sum(1 + 1, 4) else 0
-> if true then 1 + sum(1 + 1, 4) else 0
-> 1 + sum(1 + 1, 4)
-> 1 + sum(2, 4)
-> 1 + (if 2 < 4 then 2 + sum(2 + 1, 4) else 0)
-> 1 + (if true then 2 + sum(2 + 1, 4) else 0)
-> 1 + (2 + sum(2 + 1, 4))
-> 1 + (2 + sum(3, 4))
-> 1 + (2 + (if 3 < 4 then 3 + sum(3 + 1, 4) else 0))
-> 1 + (2 + (if true then 3 + sum(3 + 1, 4) else 0))
-> 1 + (2 + (3 + sum(3 + 1, 4)))
-> 1 + (2 + (3 + sum(4, 4)))
-> 1 + (2 + (3 + (if 4 < 4 then 4 + sum(4 + 1, 4) else 0)))
-> 1 + (2 + (3 + (if false then 4 + sum(4 + 1, 4) else 0)))
-> 1 + (2 + (3 + 0))
```

```
-> 1 + (2 + 3)
-> 1 + 5
-> 6
```

We refer to this sequence of rewriting steps as an *execution trace*.

Termination. Does the rewriting process always terminate and produce a finite execution trace? Consider the following recursive function:

```
def loop(x: Int): Int = loop(x)
```

If we evaluate, e.g., the call `loop(0)`, we obtain an infinite rewriting sequence:

```
loop(0) -> loop(0) -> loop(0) -> ...
```

In order to guarantee termination of a recursive function, we have to make sure that each recursive call makes progress according to some progress measure. For example, in the recursive call to the function `sum` in our example above, the difference `b - a` between the arguments decreases with every recursive call. This means that `b - a` will eventually reach 0 or become negative. At this point, we take the `else` branch in the body of `sum` and the recursion stops. For our non-terminating function `loop`, it is impossible to find such a progress measure.

1.3.2 Tail Recursion

If we apply the function `sum` to larger intervals we observe the following:

```
scala> sum(1, 1000000)
java.lang.StackOverflowError
...
```

The problem is that a call to a function requires the Scala runtime environment to allocate stack space that stores the arguments of the call and any intermediate results obtained during the evaluation of the function body in memory. For the function `sum`, the intermediate results of the evaluation must be kept on the stack until the final recursive call returns. We can see this nicely in the execution trace for the call `sum(1, 4)`. The length of the expression that we still need to simplify grows with each recursive call:

```
sum(1, 4)
-> ...
-> 1 + sum(2, 4)
-> ...
-> 1 + (2 + sum(2 + 1, 4))
-> ...
-> 1 + (2 + (3 + sum(2 + 1, 4)))
-> ...
-> 6
```

Only when the final call to `sum` has returned, can we simplify the entire expression to a value.

During execution of a Scala expression, the arguments of functions that have been called, but have not yet returned, are maintained on the *call stack*. The stack space that is needed for evaluating a call `sum(a, b)` grows linearly with the recursion depth, which is given by the size of the interval `b - a`. Since the Scala runtime environment only reserves a relatively small amount of memory for the call stack, a call to `sum` for large interval sizes runs out of stack space. This is signaled by a `StackOverflowError` exception.

Can we implement the function `sum` so that it only requires constant space? To this end, consider the following *imperative* implementation of `sum`, which uses a **`while`** loop and mutable variables to perform the summation:

```
def sumImp(a: Int, b: Int): Int =
  var acc = 0
  var i = a
  while i < b do
    acc = i + acc
    i = i + 1
  acc
```

This implementation requires only constant space, since it involves only a single function call. Moreover, the execution of a single loop iteration for the summation does not allocate memory that persists across iterations. The intermediate results are stored in the variables `i` and `acc`, which are reused in each iteration. Unfortunately, this implementation uses mutable variables. Mutable variables make it more difficult to reason about the correctness of the code. However, we can turn the imperative **`while`** loop into a recursive function by hoisting the loop counter `i` and accumulator `acc` to function parameters:

```
def loop(acc: Int, i: Int, b: Int): Int =
  if i < b then loop(i + acc, i + 1, b) else acc

def sumTail(a: Int, b: Int): Int =
  loop(0, a, b)
```

Note how the function `loop` closely mimics the **`while`** loop in the imperative implementation without relying on mutable variables. We simply pass the new values that we obtain for the loop counter `i` and the accumulator `acc` to the recursive call of `loop`.

The function `loop` has an important property: the recursive call to `loop` in the *then* branch of the conditional expression is the final computation that is performed before the function returns. That is, in the recursive case, the function directly returns the result of the recursive call. We refer to functions in which all recursive calls are of this form as *tail-recursive* functions. Contrast the new implementation of `sum` with our original implementation, which added `a` to the result of the recursive call and was therefore not tail-recursive. The tail recursive implementation has an interesting effect on the execution trace:

```
sumTail(1, 4)
   -> loop(0, 1, 4)
   -> if 1 < 4 then loop(1 + 0, 1 + 1, 4) else 0
   -> if true then loop(1 + 0, 1 + 1, 4) else 0
   -> loop(1, 2, 4)
   -> if 2 < 4 then loop(2 + 1, 2 + 1, 4) else 1
   -> if true then loop(2 + 1, 2 + 1, 4) else 1
   -> loop(3, 3, 4)
   -> if 3 < 4 then loop(3 + 3, 3 + 1, 4) else 3
   -> if true then loop(3 + 3, 3 + 1, 4) else 3
   -> loop(6, 3, 4)
   -> if 4 < 4 then loop(4 + 6, 4 + 1, 4) else 6
   -> if false then loop(4 + 6, 4 + 1, 4) else 6
   -> 6
```

Observe that the size of the expressions that we obtain throughout the trace does not grow with the recursion depth. This is because the tail-recursive call to `loop` is the final computation that is performed in the body of `loop`, before the function returns.

To simplify our implementation, we can move the declaration of the function `loop` inside the body of the function `sumTail`:

```
def sumTail(a: Int, b: Int): Int =
  def loop(acc: Int, i: Int): Int =
    if i < b then loop(i + acc, i + 1) else acc
  loop(0, a)
```

Note that in this version, we have dropped the third parameter `b` of the first version of the function `loop` since it is just passed to the recursive call without change. The occurrence of `b` in the body of the new nested version of `loop` now always refers to the parameter `b` of the outer function `sumTail`.

For tail-recursive functions, the stack space that is allocated for the current call can be reused by the recursive call. In particular, the memory that is needed to store the arguments of the current call can be reused to store the arguments of the recursive call. By reusing the current stack space, we effectively turn the recursive function back into an imperative loop. This optimization is referred to as *tail call elimination*. Many modern compilers, including the Scala compiler, automatically eliminate tail calls. Thus, tail-recursive functions are guaranteed to execute in constant stack space. We can test this feature by rerunning the tail-recursive version of `sum` for large interval sizes:

```
scala> sumTail(1, 1000000)
val res0: Int = 704982704
```

This time the function terminates normally without throwing an exception.

With tail call elimination we get the best of both worlds: we obtain the efficiency of an imperative implementation and the simplicity of a functional implementation. If you are unsure about how to write a tail-recursive function, it is often helpful to first write the function using a **`while`** loop and then transform the loop into a tail-recursive function, as we have done above. Once you get more used to functional programming, you will find writing tail-recursive functions as natural as writing loops.

If you are in doubt whether a recursive function that you wrote is tail-recursive, you can add the `@tailrec` annotation to the declaration of the function:

```
import scala.annotation.tailrec
...
def sumTail(a: Int, b: Int): Int =
  @tailrec def loop(acc: Int, i: Int): Int =
    if i < b then loop(i + acc, i + 1) else acc
  loop(0, a)
```

If the compiler fails to apply tail call elimination to a `@tailrec` annotated function, then it will issue a warning:

```
-- Error: ----------------------------------------------------
2 |  if a < b then a + sum(a + 1, b) else 0
  |                    ^^^^^^^^^^^^^
  |   Cannot rewrite recursive call: it is not in tail position
```

You may wonder whether non-tail-recursive functions should be avoided at all costs. This depends on the function. Often, tail-recursive functions are harder to understand than a recursive function that performs the same computation, but that is not tail-recursive. [1]. If you know that the recursion depth of the calls to your function will be small in practice, you may want to write the function without tail-recursion. In general, if you are in doubt, you should always value the clarity of your code higher than its efficiency. When you observe that your code is inefficient, you can still optimize it later.

1.4 Classes and Objects

In the previous sections, we have learned about the basic language features of Scala. In this section, we will learn how Scala programs are organized. Scala is an object-oriented language, so Scala programs are organized using *classes* and *objects*.

1.4.1 Classes, Fields, and Methods

Similar to Java, Scala allows you to define classes with *fields* and *methods*, which you can extend using inheritance, override, etc. Fortunately, Scala's syntax for classes is much more lightweight than Java's. For example, consider the following Java class which we can use to wrap pairs of integer values in a single

[1] Similarly, recursive functions are easier to understand than computations that use imperative loops.

object:

```
public class Pair {
  private int first;
  private int second;

  public Pair(int fst, int snd) {
    first = fst;
    second = snd;
  }

  public int getFirst() {
    return first;
  }

  public int getSecond() {
    return second;
  }
}
```

The class consists of:

- two fields called `first` and `second` of type `int` to store the two values;
- a constructor, which takes values to initialize the two fields;
- two "getter" methods to retrieve the two values (we follow good practice and declare all non-final fields as private so that their values cannot be modified without explicit method calls.).

Let us ignore for the moment that we can represent pairs directly in Scala using a tuple type. Here is how we can define the corresponding class in Scala:

```
class Pair(val first: Int, val second: Int)
```

There are some important differences between the Java and Scala version of the class `Pair`:

- In Scala, the class name is followed by a list of class parameters. These parameters serve two purposes:
  1. Parameters that are prefixed by a **`val`** or **`var`** keyword automatically create a field with the given name and type.
  2. The parameter list implicitly defines a constructor with a corresponding list of arguments. The values that are provided for arguments prefixed with **`val`** or **`var`** will be used to initialize the associated fields.
- The default visibility of classes, fields, and methods in Scala is `public`. Hence, we can access the values of the fields `first` and `second` directly and we do not need to define extra getter methods. Note that this makes

> sense because Scala discourages mutable state. In particular, we defined the two fields as `val`s, so their values cannot be changed, once an instance of class `Pair` has been created. In Scala, you can leave out the braces around an empty class body, so `class C` is the same as `class C {}`.

We can create instances of class `Pair` and access their fields as usual:

```
scala> val p = new Pair(1,2)
val p: Pair = Pair@1458e1cc
scala> p.first
val res0: Int = 1
```

What if we do want to modify the values stored in a `Pair` object? In Java, we would do this by adding appropriate "setter" methods to the class:

```
public class Pair {
  private int first;
  private int second;

  ...

  public void setFirst(int fst) {
    first = fst;
  }
  public void setSecond(int snd) {
    second = snd;
  }
}
```

In Scala, we could follow the same route: change all `val`s into `var`s, make them private, and add getter and setter methods. However, we want to avoid using `var` declarations as much as possible. The idiomatic solution in Scala is to make a copy of the entire object and change the appropriate value:

```
class Pair(val first: Int, val second: Int):
  def setFirst(fst: Int): Pair = new Pair(fst, second)
  def setSecond(snd: Int): Pair = new Pair(first, snd)
```

1.4.2 Overriding Methods

Java allows us to override methods that are declared in super classes. Since method calls are dynamically dispatched at run-time, this feature allows us to modify the behavior of an object of the subclass when it is used in a context where an object of the super class is expected.

All Java classes extend the class `Object`. The class `Object` provides, among others, a method `toString`, which computes a textual representation of the object. In particular, the `toString` method can be used to pretty-print objects. By default, the textual representation of objects consists of the name of the object's class, followed by a unique object ID. We can modify the way objects of a specific class are printed, by overriding the `toString` method. In Java, this can be done as follows:

```
public class Pair {
  private int first;
  private int second;

  ...

  public String toString() {
    return "Pair(" + first + ",␣" + second + ")";
  }
}
```

In Scala, all classes extend the class `scala.Any` which also provides a method called `toString`. Scala's class hierarchy is further subdivided into the classes `scala.AnyVal` and `scala.AnyRef`, which are directly derived from `scala.Any`. All instances of `scala.AnyVal` are immutable, whereas instances of `scala.AnyRef` may have mutable state. That is, `scala.AnyRef` corresponds to Java's `Object` class.

If we want to override a method in a Scala class, we have to explicitly say so by using the **`override`** qualifier:

```
class Pair(val first: Int, val second: Int):
  ...
  override def toString = "Pair(" + first + ",␣" + second + ")"
```

The pretty printer in the REPL will now use the new `toString` method to print `Pair` objects:

```
scala> val p = new Pair(1,2)
val p: Pair = Pair(1, 2)
```

1.4.3 Singleton and Companion Objects

If the construction of an object involves complex initialization code, it is often useful to declare dedicate *factory* methods that perform this initialization. In Java, we would declare such methods as *static* members of the corresponding class:

```
public class Pair {
  ...
  public static Pair make(int fst, int, snd) {
    return new Pair(fst, snd);
  }
}
```

We can now call `Pair.make` to create new `Pair` instances.

Scala does not support static methods as they violate the philosophy of object-oriented programming that "everything is an object". Instead of static methods, it provides *singleton objects*. Singleton objects are declared just like classes, but using the keyword **`object`** instead of **`class`**. There exists exactly one instance of each **`object`**, which is automatically created from the **`object`** declaration when the program is started. Since no further instances of the object can be created, object declarations do not have parameter lists.

For every class `C` in a Scala program, one can declare a singleton object that is also called `C`. This object is referred to as the *companion object* of `C`. Companion objects have access to all private members of instances of `C`. Consequently, a method or field that is defined in the companion object is equivalent to a static method/field of `C` in Java:

```
class Pair(val first: Int, val second: Int):
  ...
object Pair:
  def make(fst: Int, snd: Int) = new Pair(fst, snd)
```

We can access members of companion objects just like static class members in Java:

```
scala> def p = Pair.make(3,4)
val p: Pair = Pair(3, 4)
```

1.4.4 The `apply` Method

Methods with the name `apply` are treated specially by the Scala compiler. For example, if we rename the factory method `make` in our companion object for the `Pair` class to `apply`

```
object Pair:
  def apply(fst: Int, snd: Int) = new Pair(fst, snd)
```

then we can call this method simply by referring to the `Pair` companion object, followed by the argument list of the call:

```
scala> def p = Pair(3,4)
p: Pair = Pair(3, 4)
```

This is equivalent to the following explicit call to the `apply` method:

```
scala> def p = Pair.apply(3,4)
p: Pair = Pair(3, 4)
```

The compiler automatically expands `Pair(3,4)` to `Pair.apply(3, 4)`. That is, objects with an `apply` method can be used as if they were functions[2]. This feature is particularly useful to enable concise calls to factory methods. In fact, factory methods for the data structures in the Scala standard library are typically implemented using `apply` methods provided by companion objects.

[2]If you are familiar with C++, then you will notice that this feature is similar to overloading the function call operator `()` in C++.

1.5 Algebraic Data Types

Algebraic data types (ADTs) and pattern matching are constructs that are commonly found in functional programming languages. They allow you to implement regular, non-encapsulated data structures (such as lists and trees) in a convenient fashion. We will make heavy use of this feature throughout this course.

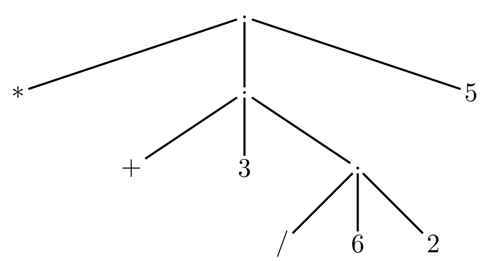
1.5.1 Enumerations

Suppose we want to implement a simple calculator program that takes arithmetic expressions such as

$$(3 + 6/2) * 5$$

as input and evaluates these expressions. This problem is quite similar to writing an interpreter for a programming language, except that the language that we are interpreting here (i.e., arithmetic expressions) is much simpler than a full-blown programming language.

One of the first questions that we have to answer is: how do we represent expressions in our program? Our representation should allow us to easily implement common tasks such as pretty printing, evaluation, and simplification of expressions. In particular, the representation should make the precedence of operators in expressions explicit. E.g., consider the expression $3 + 6/2$, then when we evaluate the expression, our representation should immediately tell us that we first have to divide 6 by 2 before we add 3. To achieve this, expressions are represented as *abstract syntax trees*, or ASTs for short. For example, the abstract syntax tree of the expression $(3 + 6/2) * 5$ can be visualized as follows:



For the AST node representing an expression $e_1 \, op \, e_2$, we follow the convention of representing the operator op as the left-most child of the node, rather than putting it between the children representing the operands e_1 and e_2.

Note that the AST tells us exactly how to evaluate the expression. We start at the root. At each node that we visit, we first recurse into the second subtree to evaluate the left operand of the operation. Then we do the same for the third subtree, which represents the right operand of the operation. Finally, we combine the results obtained from the two operands according to the operator labeling the first child. We will learn more about ASTs later. For now it suffices if you have an intuitive understanding what ASTs are.

Algebraic data types allow us to represent tree-like data structures such as ASTs. In Scala, algebraic data types are constructed using *enumerations* or "enums" for short. The following enum defines the ASTs of our arithmetic expressions:

```
enum Expr:
  /* Numbers such as 1, 2, etc. */
  case Num(num: Int)
  /* Expressions composed using binary operators */
  case BinOp(op: Bop, left: Expr, right: Expr)

/* Binary operators */
enum Bop:
  case Add /* + */
  case Sub /* - */
  case Mul /* * */
  case Div /* / */
```

This code declares an enum `Expr` whose instances represent the ASTs of our expression language. The code distinguishes two types of `Expr` objects based on the root node of the represented AST. A `Num(num)` object represents an AST consisting of a single node storing the integer constant `num`. Similarly, an object `BinOp(op, left, right)` represents an AST whose root node combines two subexpressions represented by ASTs `left` and `right` using the *binary operator* `op`. Binary operators are represented by the abstract class `Bop` which is implemented using four cases representing the different kinds of arithmetic operations: `Add`, `Sub`, `Mul`, and `Div`.

We refer to the cases of an algebraic data type as its *variants*. That is, the ADT `Expr` has variants `Num` and `BinOp`.

The Scala compiler adds some convenient functionality to enums. First, it automatically generates companion objects with appropriate factory methods. These methods are particularly useful when you nest them to construct complex expressions:

```
scala> import Expr._, Bop._
scala> val e = BinOp(Add, Num(3), BinOp(Mul, Num(4), Num(5)))
val e: Expr = BinOp(Add, Num(3), BinOp(Mul, Num(4), Num(5)))
```

Note that the enums `Expr` and `Bop` itself have companion objects. In turn, these companion objects have nested companion objects for the variants of the defined ADT within them. For example, the companion object for `Expr` contains companion objects for `Num` and `BinOp`. The nested companion objects `Num` and `BinOp` then have `apply` methods for constructing objects of the corresponding variant. The **`import`** instruction in the above code snippet makes the nested companion objects directly accessible in the code. For instance, in the code `Num(4)`, `Num` refers to the companion object of variant `Num` in `Expr`. Thus, this code expands to `Expr.Num.apply(4)`.

Second, the compiler adds natural implementations of the methods `toString`, `hashCode`, and `equals` for each variant. These will print, hash, and compare a whole tree consisting of the top-level enum instance and (recursively) all its arguments. In Scala, an expression of the form `x == y` always translates into a call of the form `x.equals(y)` (just like in Java). The overriden `equals` method therefore ensures that enum instances are always compared structurally. For example, we have:

```
scala> val e1: BinOp = BinOp(Add, Num(3), Num(4))
val e1: Expr.BinOp = BinOp(Add, Num(3), Num(4))
scala> val e2 = BinOp(Add, Num(3), Num(4))
val e2: Expr.BinOp = BinOp(Add, Num(3), Num(4))
scala> e1 == e2
val res2: Boolean = true
```

In the example above, `e1` and `e2` point to two different objects in memory. However, the two ASTs represented by these objects have exactly the same structure. Hence, `e1 == e2` evaluates to **`true`**.

Next, the compiler implicitly adds a **`val`** prefix to all arguments in the parameter list of an enum object, so that they are maintained as fields:

```
scala> val n = e1.left
val n: Expr = Num(3)
```

Note that the above code only works because we explicitly declared `e1` to be of type `BinOp` earlier. Without this type annotation, the inferred type of `e1` would be `Expr`. However, `Expr` objects are not guaranteed to have a field called `left` since `Expr` also includes `Num` objects.

Finally, the compiler adds a `copy` method to your enum cases for making modified copies. This method is useful if you need to create a copy of an existing enum object `o` that is identical to `o` except for some of `o`'s attributes:

```
scala> e1.copy(op = Sub)
val res4: Expr = BinOp(Sub, Num(3), Num(4))
```

1.5.2 Pattern Matching

Suppose we want to implement an algorithm that simplifies expressions by recursively applying the following simplifications rules:

- $e + 0 \Rightarrow e$
- $e * 1 \Rightarrow e$
- $e * 0 \Rightarrow 0$

To identify whether a given expression matches one of the left-hand sides of the rules, we have to look at some of its subexpressions. E.g., to check whether an expression of the form $e_1 + e_2$ matches the left-hand side of the first rule, we have to look at the left subexpression e_1 to check whether $e_1 = 0$. Implementing this kind of pattern matching is quite tedious in many languages (including Java). Fortunately, the Scala language has inbuilt support for pattern matching that works hand-in-hand with enumerations.

Let us first reformulate the three simplification rules in terms of our enum representation of expressions:

```
BinOp(Add, e1, Num(0)) => e1
BinOp(Mul, e1, Num(1)) => e1
BinOp(Mul, e1, Num(0)) => Num(0)
```

Using pattern matching, these rules almost directly give us the implementation of the following function `simplifyTop`, which applies the rules at the top-level of the given expression `e`:

```
def simplifyTop(e: Expr) =
  e match
    case BinOp(Add, e1, Num(0)) => e1
    case BinOp(Mul, e1, Num(1)) => e1
    case BinOp(Mul, _, Num(0)) => Num(0)
    case _ => e
```

The body of `simplifyTop` is a *match expression*. A match expression consists of a *selector*, in this case `e`, followed by the keyword **`match`**, followed by a sequence of match alternatives.

Each match alternative starts with the keyword **`case`**, followed by a pattern, followed by an expression that is evaluated if the pattern matches the selector. The pattern and expression are separated by an arrow symbol **`=>`**.

A match expression is evaluated by checking whether the selector matches one of the patterns in the alternatives. The patterns are tried in the order in which they appear in the program. The first pattern that matches is selected and the expression following the arrow is evaluated. The result of the entire match expression is the result of the expression in the selected alternative.

Here is an example of a recursive function that uses pattern matching to pretty print arithmetic expressions:

```
def pretty(e: Expr): String =
  e match
    case BinOp(bop, e1, e2) =>
      val bop_str = bop match
        case Add => " + "
        case Sub => " - "
        case Mul => " * "
        case Div => " / "
      "(" + pretty(e1) + bop_str + pretty(e2) + ")"
    case Num(n) => n.toString()
```

```
scala> val e = BinOp(Add, BinOp(Mult, Num(3), Num(4)), Num(1))
val e: BinOp = BinOp(Add, BinOp(Mult, Num(3), Num(4)), Num(1))
scala> pretty(e)
val res0: String = ((3 * 4) + 1)
```

There are different types of patterns. The most important types are:

- *Constant patterns*: A constant pattern such as `0` matches values that are equal to the constant (with respect to `==`).
- *Variable patterns*: A variable pattern such as `e1` matches every value. Here, `e1` is a variable that is bound in the pattern. The variable refers to the matched value in the right-hand side of the match alternative.
- *Wildcard patterns*: A wildcard pattern `_` also matches every value, but it does not introduce a variable that refers to the matched value.
- *Constructor patterns*: A constructor pattern such as `BinOp(Add, e, Num(0))` matches all values of type `BinOp` whose first argument matches `Add`, whose second argument matches `e`, and whose third argument matches `Num(0)`. Note that the arguments to the constructor `BinOp` are themselves patterns. This allows you to write deep patterns that match complex enum values using a concise notation.

1.5.3 Binding Names in Patterns

Sometimes we want to match a subexpression against a specific pattern and also bind the matched expression to a name. This is useful when we want to reuse a matched subexpression in the right-hand side of the match alternative. For example, in the third simplification rule of `simplifyTop` we are returning `Num(0)`, which is also the second subexpression of the matched expression `e`. Instead of creating a new expression, `Num(0)` on the right-hand side of the rule, we can also directly return the second subexpression of `e`. We can do this by binding a name to that subexpression in the pattern using the operator `@` as follows:

```
def simplifyTop(e: Expr) =
  e match
    case BinOp(Add, e1, Num(0)) => e1
    case BinOp(Mul, e1, Num(1)) => e1
    case BinOp(Mul, _, e2 @ Num(0)) => e2
    case _ => e
```

Note that the pattern in the third match alternative now binds the name `e2` to the value matched by the pattern `Num(0)`. This value is then returned on the righ-hand side of the rule by referring to it using the name `e2`.

1.5.4 Pattern Guards

Suppose we want to extend our expression simplifier so that it additionally implements the following simplification rule: $e + e \Rightarrow 2 \times e$

If we directly translate the rule to a corresponding match alternative, we obtain the following implementation of `simplilfyTop`:

```
def simplifyTop(e: Expr) =
  e match
```

```
    ...
    case BinOp(Add, e1, e1) => BinOp(Mul, Num(2), e1)
    case _ => e
```

Unfortunately, the compiler will reject this function because we use the name `e1` twice within the same pattern. In general, a variable name such as `e1` may only be used once in a pattern. We can solve this problem by using a different variable name for the second subexpression, say `e2`, and then use a *pattern guard* to enforce that the subexpressions matched by `e1` and `e2` are equal:

```
def simplifyTop(e: Expr) =
  e match
    ...
    case BinOp(Add, e1, e2) if e1 == e2 =>
      BinOp(Mul, Num(2), e2)
    case _ => e
```

In general, a pattern guard can be an arbitrary Boolean expression over the names that are in the scope of the match alternative. The pattern guard is appended to the pattern of a match alternative using the keyword **if**.

1.5.5 Exhaustiveness Checks

Consider the following function `simplifyAll` that applies our simplification rules recursively to the given expression:

```
def simplifyAll(e: Expr): Expr =
  e match
    case BinOp(Add, e1, Num(0)) => simplifyAll(e1)
    case BinOp(Mul, e1, Num(1)) => simplifyAll(e1)
    case BinOp(Mul, _, e2 @ Num(0)) => e2
    case BinOp(Add, e1, e2) if e1 == e2 =>
      BinOp(Mul, Num(2), simplifyAll(e2))
    case BinOp(bop, e1, e2) =>
      BinOp(bop, simplifyAll(e), simplifyAll(e2))
```

Observe that in this function, the pattern alternatives are no longer exhaustive. That is, there exist values `e` that are not matched by any of the match alternatives, e.g., the value `Num(0)`. If `simplifyAll` is called with `Num(0)`, it will throw a runtime exception.

We can fix this code by adding an explicit match alternative for the `Num` constructor:

```
def simplifyAll(e: Expr): Expr =
  e match
    case BinOp(Add, e1, Num(0)) => simplifyAll(e1)
    case BinOp(Mul, e1, Num(1)) => simplifyAll(e1)
    case BinOp(Mul, _, e2 @ Num(0)) => e2
    case BinOp(Add, e1, e2) if e1 == e2 =>
```

```
      BinOp(Mul, Num(2), simplifyAll(e2))
    case BinOp(bop, e1, e2) =>
      BinOp(bop, simplifyAll(e), simplifyAll(e2))
    case Num(_) => e
```

With complex patterns like this it can be tricky to keep track of all the possible cases. Fortunately, the compiler will automatically check whether the case analysis is exhaustive. For instance, consider the following faulty implementation of `simplifyAll` where we have omitted the last case for `BinOp` from the previous implementation:

```
def simplifyAll(e: Expr): Expr =
  e match
    case BinOp(Add, e1, Num(0)) => simplifyAll(e1)
    case BinOp(Mul, e1, Num(1)) => simplifyAll(e1)
    case BinOp(Mul, _, e2 @ Num(0)) => e2
    case BinOp(Add, e1, e2) if e1 == e2 =>
      BinOp(Mul, Num(2), simplifyAll(e2))
    case Num(_) => e
```

This version does not compile:

```
-- [E029] Pattern Match Exhaustivity Warning: ----------
2 |  e match
  |  ^
  |  match may not be exhaustive.
  |
  |  It would fail on pattern case: Expr.BinOp(_, _, _)
```

Unfortunately, these exhaustiveness checks can sometimes produce spurious warnings. For example, suppose we have a function that is meant to pretty-print number expressions, but not other expressions which have not yet been reduced:

```
def prettyNumber(e: Expr): String =
  e match
    case Num(num) => num.toString()
```

Further suppose that we know that our program ensures that `prettyNumber` is never called on a `BinOp` expression. Yet, the compiler still complains about the non-exhaustive pattern matching. We can suppress this warning by declaring `e` as *unchecked*:

```
def prettyNumber(e: Expr): String =
  (e: @unchecked) match
    case Num(num) => num.toString()
```

While `@unchecked` notations are sometimes necessary to suppress spurious warnings, you should be very careful about introducing them in your code. In most cases, the compiler generated warnings indicate actual problems in your code that need your attention.

1.5.6 Option Types

Suppose we want to write a function that evaluates arithmetic expressions to `Int` values. One question is: How should we deal with undefined operations such as division by zero:

```
BinOp(Div, e, Num(0)) => ???
```

In Java, we would typically go for one of the following two solutions:

- throw an exception such as `ArithmeticException`;
- return **`null`** to indicate that the intended operation does not yield a valid result.

Both approaches have advantages and drawbacks.

Exceptions are a good solution if the undefined operation is indeed exceptional behavior that should, e.g., abort the program. In this case, we ensure that a computation that returns normally always yields a valid result. However, if the undefined operation commonly occurs in computations, we will have to catch the exception and handle it appropriately. This has two disadvantages. First, the exception mechanism is relatively expensive and should only be used in truly exceptional situations. Second, the exception handlers will clutter the code and the non-structured control flow of thrown exceptions makes it more difficult to understand what the program is doing.

If we return **`null`**, we avoid the two disadvantages of exceptions: the computation always returns normally, and there is no computational overhead such as recording the stack-trace to the point where the exception was thrown. However, **`null`** values introduce their own problems. Since **`null`** can have an arbitrary type, the type checker of the compiler will give us much weaker static correctness guarantees for our code. In particular, it will be unable to statically detect unintended accesses to the return value in cases where the return value is invalid (hello `NullPointerException`!).

In languages that support pattern matching, there is a common idiom that avoids the problem of introducing **`null`** values: option types.

The option type is an algebraic data type with two variants: `Some(v)` to indicate that a computation returned a proper result value `v`, and `None` to indicate that the intended operation was undefined and has no proper result.

In Scala, we can define an option type for `Int` values using an enum as follows:

```
enum IntOption:
  case Some(value: Int)
  case None
```

We can now use the option type similarly to null values in Java:

```
def div(x: Int, y: Int): IntOption =
  if y == 0 then None else Some(x / y)
```

Unlike in Java, where the static type checker is unable to distinguish a **null** value from a genuine result of a computation, the Scala type checker will force us to explicitly unwrap the Int value embedded in an IntOption before we can access it. Using pattern matching, we can do this conveniently. For example, suppose we want to convert the result of div to a double precision floating point number. By using pattern matching on the return value of div, we can recover from some of the cases where integer division by 0 is undefined:

```
def divToDouble(x: Int, y: Int): Double =
  div(x,y) match
    case Some(x) => x
    case None =>
      if x < 0then Double.NegativeInfinity
      else if x > 0 then Double.PositiveInfinity
      else Double.NaN
```

Since option types are so useful, Scala already provides a generic option type, called Option, in its standard library. Using the predefined type Option we can write the function div like this:

```
def div(x: Int, y: Int): Option[Int] =
  if y == 0 then None else Some(x / y)
```

Chapter 2

Foundations

Recursion and induction will be our main tools for formalizing programming languages. In this chapter, we study the mathematical foundations of these two closely related concepts. We will then apply these concepts to understand a ubiquitous recursive data structure: lists. We will later see that this formal approach allows us to prove mathematical theorems about individual programs and, more generally, about a programming language as a whole.

2.1 Notation

Throughout the course notes we will assume a basic understanding of mathematical notation and some concepts and notations from set theory. We briefly recap these in this section.

Properties. We use standard logical notation to express *properties*, i.e. formal mathematical statements that are true or false. For instance, $1 < 2$ is a true property and $2 < 1$ is a false property. For given properties P and Q, we introduce

- the *negation* of P, written $\neg P$, which is true iff[1] P is false,
- the *conjunction* of P and Q, written $P \wedge Q$, which is true iff both P and Q are true,
- the *disjunction* of P and Q, written $P \vee Q$, which is true iff P or Q is true,
- the *implication* of P and Q, written $P \Rightarrow Q$, which is true iff P implies Q (i.e., if P is true then Q is true),
- and the *equivalence* of P and Q, written $P \Leftrightarrow Q$, which is true iff P and Q are logically equivalent (i.e. $P \Rightarrow Q$ and $Q \Rightarrow P$).

[1] "iff" abbreviates "if and only if"

Properties may refer to variables (or unknowns). For instance, the property $0 \leq x \wedge x < 10$ is true for all natural numbers x between 0 and 9. We write $P(x_1, \ldots, x_n)$ to indicate that P refers to the variables x_1 to x_n. For a property $P(x)$, *universal quantification* over x yields the property $\forall x : P(x)$, which is true iff $P(x)$ is true for all possible values of x. Similarly, *existential quantification* over x yields the property $\exists x : P(x)$, which is true iff there exists at least one value for x such that $P(x)$ is true.

Sets. A *set* is an unordered collection of objects, which we refer to as the *elements* or *members* of the set. We write $e \in S$ to indicate that e is an element of the set S and we write $e \notin S$ for $\neg(e \in S)$. We write $\forall x \in S : P(x)$ as a short-hand for the property $\forall x : x \in S \Rightarrow P(x)$ and likewise write $\exists x \in S : P(x)$ for $\exists x : x \in S \wedge P(x)$.

We sometimes denote a set by explicitly enumerating its elements using the notation $\{a, b, c, d, e\}$. Here, $\{a, b, c, d, e\}$ is the set consisting of the elements a, b, c, d, and e. We denote the empty set by $\emptyset$ and we write $\mathbb{N}$ for the set of all natural numbers $\mathbb{N} = \{0, 1, 2, \ldots\}$ and $\mathbb{Z}$ for the set of all integers $\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$.

At times, we will define a set in terms of a *set comprehension*, i.e., the set of all elements x that satisfy some given property $P(x)$, written $\{\, x \mid P(x) \,\}$. For example, the set comprehension $\{\, x \mid x \in \mathbb{Z} \wedge x < 0 \,\}$ defines the set of all negative integers.[2]

We define the usual operations on sets such as *union* $S \cup T$, *intersection* $S \cap T$, and *set difference* $S \setminus T$:

$$
\begin{aligned}
S \cup T &\stackrel{\text{def}}{=} \{\, x \mid x \in S \vee x \in T \,\} \\
S \cap T &\stackrel{\text{def}}{=} \{\, x \mid x \in S \wedge x \in T \,\} \\
S \setminus T &\stackrel{\text{def}}{=} \{\, x \mid x \in S \wedge x \notin T \,\}
\end{aligned}
$$

A set S is a *subset* of another set T, written $S \subseteq T$, if every element of S is also an element of T. For example, we have $\mathbb{N} \subseteq \mathbb{Z}$ and for all sets S we have $\emptyset \subseteq S$ and $S \subseteq S$. Two sets S and T are equal if they have the same elements, i.e. if $S \subseteq T$ and $T \subseteq S$.

Given a set S, we write 2^S for the set of all subsets of S:

$$2^S \stackrel{\text{def}}{=} \{\, x \mid x \subseteq S \,\}$$

We call 2^S the *powerset* of S. For instance, the powerset of $\{0, 1\}$ is $2^{\{0,1\}} = \{\emptyset, \{0\}, \{1\}, \{0, 1\}\}$.[3]

Given two elements a and b, we construct a new element $\langle a, b\rangle$ called the *(ordered) pair* consisting of a and b. Formally, we can define $\langle a, b\rangle$ as the set

[2] Not every property $P(x)$ defines a set. A notorious example is the property $x \notin x$, which is at the heart of Russel's Paradox.

[3] The notation 2^S is reminiscent of the fact that if S is a finite set consisting of n elements, then the powerset of S has 2^n elements.

$\{\{a\},\{a,b\}\}$.[4] We denote by $S \times T$ the *Cartesian product* of the sets S and T, which is the set of all pairs consisting of elements from S and T:

$$S \times T \stackrel{\text{def}}{=} \{\, \langle x, y\rangle \mid x \in S \wedge y \in T \,\}$$

We generalize these definitions to *n-tuples* $\langle a_1, \ldots, a_n\rangle$ consisting of n elements $a_1,\ldots,a_n$ and Cartesian products over n sets $S_1 \times \cdots \times S_n$ for arbitrary $n \in \mathbb{N}$. In particular, for an element a, the 1-tuple $\langle a\rangle$ is equal to the set $\{a\}$ and the *empty tuple* $\langle\rangle$ is equal to the empty set.

Relations and Functions. Given two sets S and T, we call an element $R \in 2^{S\times T}$ a *binary relation* over S and T. That is, we have $R \subseteq S \times T$ by definition. If $\langle x, y\rangle \in R$ for some $x \in S$ and $y \in T$, we say that R *relates* x to y. We denote by $R^{-1} \subseteq T \times S$ the *inverse* of R:

$$R^{-1} \stackrel{\text{def}}{=} \{\, \langle y, x\rangle \mid \langle x, y\rangle \in R \,\}$$

R is called *total* if R relates every element of S to at least one element of T. R is called *functional* if R relates every element of S to at most one element of T. If R is functional, it is also called a *partial function* from S to T. If R is functional and total, it is called a *(total) function* from S to T. We write $R : S \rightharpoonup T$ to indicate that R is a partial function from S to T and $R : S \to T$ to indicate that it is a function from S to T. Every (total) function is also a partial function but the converse does not hold.

Let $f : S \rightharpoonup T$. We say that f is *defined* for $x \in S$ if there exists y such that $\langle x, y\rangle \in f$. In this case, we say that f *maps* x to y. We denote this unique y by $f(x)$. The *domain* of f, written $\mathsf{dom}(f)$, is the set of all $x \in S$ for which f is defined. Note that f is total iff $\mathsf{dom}(f) = S$.

We typically define a (partial) function from S to T using one or more equations that define $f(x)$ for all $x \in \mathsf{dom}(f)$. For example, the following definition defines the *successor* function on natural numbers:

$$\begin{aligned} \mathit{succ} &: \mathbb{N} \to \mathbb{N} \\ \mathit{succ}(x) &= x + 1 \end{aligned}$$

That is, $\mathit{succ} = \{\langle 0, 1\rangle, \langle 1, 2\rangle, \langle 2, 3\rangle, \ldots\}$.

For a partial function $f : S \rightharpoonup T$ and any elements a and b, we denote by $f[a \mapsto b]$ the partial function from $S \cup \{a\}$ to $T \cup \{b\}$ defined as follows:

$$f[a \mapsto b](x) \stackrel{\text{def}}{=} \begin{cases} f(x) & x \in \mathsf{dom}(f) \setminus \{a\} \\ b & x = a \\ \text{undefined} & \text{otherwise} \end{cases}$$

That is, $f[a \mapsto b]$ maps a to b and otherwise behaves like f. Note that we allow $a \in S$ and $b \in T$ but we do not require this.

[4] Convince yourself that with this definition we have $\langle a, b\rangle = \langle a', b'\rangle$ if and only if $a = a'$ and $b = b'$.

For a binary relation $R \subseteq S \times S$, we define its *reflexive closure* as the relation $R \cup \{ \langle x, x \rangle \mid x \in S \}$. We further define its *transitive closure*, denoted R^+, as the smallest relation that satisfies the following two conditions:

1. $R \subseteq R^+$, and
2. if $\langle x, y \rangle \in R^+$ and $\langle y, z \rangle \in R^+$ then $\langle x, z \rangle \in R^+$.

We denote by R^* the reflexive and transitive closure of R.

2.2 Structural Recursion and Induction

Recursion is a constructive technique for describing infinite sets (and infinite functions and relations on these sets). Induction is a technique for proving properties about recursively defined sets, functions and relations. There exist different variants of recursion and induction. We are interested in the simplest form of these concepts which we refer to as *structural recursion* and *structural induction*, respectively.

2.2.1 Structurally Recursive Definitions

We explain all these concept using a very simple example. To this end, we define a set N that behaves just like (or mathematically speaking, is isomorphic to) the natural numbers $\mathbb{N}$. We will represent the natural numbers as certain tuples. Once we have defined N, we will build structurally recursive functions on N that correspond to addition and multiplication, and then prove properties of these functions.

We represent the natural numbers as follows, using only nesting of tuples: we start by 0, which we represent as the empty tuple $\langle\rangle$. Then, given a natural number n, we construct its successor $n+1$ by wrapping n in another tuple:

$$\begin{array}{ll} 0 & \langle\rangle \\ 1 & \langle\langle\rangle\rangle \\ 2 & \langle\langle\langle\rangle\rangle\rangle \\ & \ldots \end{array}$$

That is, the number 1 is represented by the tuple that contains the empty tuple, the number 2 is the tuple that contains the tuple containing the empty tuple, etc. For the set N, we choose exactly those tuples that represent natural numbers following the above convention[5].

We now show how we can describe the set N using structural recursion, without referring to the natural numbers $\mathbb{N}$. We do this by providing recursive construction rules for the elements of N:

[5] This construction of the natural numbers is similar to the standard set-theoretic construction of natural numbers, except that we use a slightly simpler construction rule for successors here.

1. The empty tuple $\langle\rangle$ is an element of N.
2. If x is an element of N, then the tuple $\langle x\rangle$ is an element of N.
3. N only contains elements that can be constructed using rules 1 and 2.

We can present these construction rules more compactly using the following *inference rules*:

$$\textsc{Rule 1}\quad \langle\rangle \in N \qquad\qquad \textsc{Rule 2}\quad \frac{x \in N}{\langle x\rangle \in N}$$

Rule 3 is left implicit.

In general, an inference rule takes the form

$$\frac{P_1 \qquad \dots \qquad P_n}{C}$$

Such a rule states that if the properties $P_1, \dots, P_n$ hold, then also C holds, $P_1 \wedge \dots \wedge P_n \Rightarrow C$. The properties P_i are called the *premises* of the rule, and the property C the *conclusion* of the rule. If a rule has no premise, then the conclusion always holds. Such rules are also called *axioms* and we omit the line separating the premises from the conclusion in this case. For instance, Rule 1 above is an axiom.

Using Rule 1, we can construct the representation of the number 0. Using Rule 2, we can construct the representation of the number $n + 1$, given the representation of the number n. For the representation of the number 3 we need four construction steps:

1.	$\langle\rangle$	with rule 1
2.	$\langle\langle\rangle\rangle$	with rule 2
3.	$\langle\langle\langle\rangle\rangle\rangle$	with rule 2
4.	$\langle\langle\langle\langle\rangle\rangle\rangle\rangle$	with rule 2

The recursive definition of N is to be understood such that N contains exactly those objects that can be constructed by the inference rules in a *finite number* of steps. Whenever we give a recursive definition of a set in terms of inference rules, then this additional requirement (which corresponds to Rule 3) is always implicit.

Alternatively, we can describe the construction rules for the elements of N using a fixpoint equation:

$$N = \{\langle\rangle\} \cup \{\, \langle x\rangle \mid x \in N \,\}$$

That is, N is defined as the smallest set (with respect to subset inclusion) that satisfies this recursive equation. We also say that N is the *least fixpoint* of the equation. The left hand side of the equation captures the two construction rules for the elements of N. The fact that N is defined as the smallest set that satisfies the equation (i.e., the least fixpoint rather than any other fixpoint) captures Rule 3.

All of the above definitions of the set N are equivalent, i.e., they define the exact same set of mathematical objects. We will mostly work with definitions of recursive sets that are given in the form of inference rules as these are usually more intuitive than those given as solutions of fixpoint equations.

The recursive definition of N has two important properties that make it a *structurally recursive* definition:

1. Every object in N can be constructed only with exactly one rule. For example, $\langle\rangle$ can only be constructed with the first rule and $\langle\langle\rangle\rangle$ only with the second rule.

2. The recursive rule constructs from an object $x \in N$ a larger object $\langle x\rangle \in N$ that contains x as a proper subobject.

In general, we will work with structurally recursive definitions that have more than one base case rule and more than one recursive rule.

Algebraic data type definitions in Scala can be viewed as structurally recursive definitions of sets of tuples. For example the following declarations give a possible Scala encoding of our definition of the set N:

```
enum N:
  case Zero // rule 1
  case Succ(x: N) // rule 2
```

We have one **case** declaration per construction rule of N.

2.2.2 Recursive Definitions of Functions

The real power of structural recursion is that we can also use it to define functions that operate on the elements of a recursive set. For example, suppose we want to define a function $D : N \to \mathbb{N}$ that maps the elements of N to the natural numbers they represent. We can do this as follows:

$$
\begin{aligned}
D &: N \to \mathbb{N} \\
D(\langle\rangle) &= 0 \\
D(\langle x\rangle) &= 1 + D(x)
\end{aligned}
$$

This definition has two important properties that make it a *structurally recursive function definition*:

1. For each defining rule of N, there is a corresponding defining rule for D. This implies that for every element of N exactly one rule for D applies, which ensures that D is a partial function.

2. Recursive applications of D only apply to proper subobjects of its argument. (In the second rule for D, the recursive application of D for the argument $\langle x \rangle$ is on the proper subobject x.) This ensures that D is total.

Hence, together, these properties guarantee that the rules for D define a function, i.e., D is well-defined. We can also view the above definition as a blue print for the declaration of a Scala function on our algebraic data type representation of `N`:

```
def D(y: N): Int =
  y match
    case Zero => 0
    case Succ(x) => 1 + D(x)
```

From the fact that the definition of the function D is structurally recursive, it follows that `D` terminates normally for all input values `y`.

Next, we use structural recursion to define a two-valued function $\oplus : N \times N \to N$ that corresponds to addition on natural numbers:

$$\begin{aligned} \oplus &: N \times N \to N \\ \langle\rangle \oplus y &= y \\ \langle x \rangle \oplus y &= \langle x \oplus y \rangle \end{aligned}$$

We use the symbol $\oplus$ for this function instead of $+$ to distinguish it from the actual addition function $+ : \mathbb{N} \times \mathbb{N}$ on natural numbers. We will see below how these two functions are formally related.

Note that the second rule in the definition of D determines how the value of D for larger arguments is constructed from values of D for smaller arguments. In the case of $\oplus$, the recursion only goes over the first argument of the function.

2.2.3 Structural Induction

Often, we want to prove that a particular property holds for all elements of a recursively defined set. For example, we may want to prove that the function $\oplus$ corresponds to the actual addition function $+$ on natural numbers. In other words, we may want to prove that the property

$$D(x \oplus y) = D(x) + D(y)$$

holds for all elements $x, y \in N$.

In general, if we want to prove that all $x \in N$ satisfy a given property A, we can proceed as follows:

1. Prove that $\langle\rangle$ satisfies A.

2. Prove that for all $x \in N$, if x satisfies A, then $\langle x \rangle$ satisfies A.

We call the proof rule that we just formulated the *induction principle* for N. We can write this rule more compactly as an inference rule:

$$\frac{A(\langle\rangle) \qquad \forall x \in N : A(x) \Rightarrow A(\langle x \rangle)}{\forall x \in N :\ A(x)}$$

Notice that the premises of this inference rule

(1) $A(\langle\rangle)$

(2) $\forall x \in N : A(x) \Rightarrow A(\langle x \rangle)$

are derived directly from the defining rules of N. The first premise is called the *base case* of the induction rule and the second premise is called the *induction step*. The left-hand side of the implication in the induction step is called the *induction hypothesis*. When we prove $A(\langle x \rangle)$ in the induction step for a particular A, we can assume that the induction hypothesis $A(x)$ holds.

For every set that is defined by structural recursion there is a corresponding induction principle that is derived from the definition of the set. Every rule that is an axiom yields a base case and every other rule yields an induction step.

In the next section, we will prove the correctness of the induction principle for N by proving the correctness of a more general induction principle. However, you should already try to develop some intuition for why the induction principle for N is correct. Perhaps the following explanation helps you if you have trouble understanding the rule. Let A be some property for which the premises (1) and (2) of the induction principle hold. We validate that from these premises follows the validity of

$$A(\langle\langle\langle\langle\rangle\rangle\rangle\rangle)$$

That is, we want to validate that A holds for our representation of the number 3. First, from premise (1) follows:

$$A(\langle\rangle)$$

From this fact and premise (2) we conclude:

$$A(\langle\langle\rangle\rangle)$$

Applying premise (2) twice more we obtain the property we wanted to show:

$$A(\langle\langle\langle\langle\rangle\rangle\rangle\rangle)\ .$$

The trick lies in premise (2), which states that for any $x \in N$, if $A(x)$ holds, then also $A(\langle x \rangle)$ holds.

Let us use the induction principle to prove the equivalence of the functions $\oplus$ on N and $+$ on $\mathbb{N}$. To this end, we define the property

$$A(x) \stackrel{\text{def}}{=} \forall y \in N :\ D(x \oplus y) = D(x) + D(y)$$

and then prove that for all $x \in N$, $A(x)$ holds. For the proof to succeed, it is important that we let the induction go over x rather than y because we will need to use the recursive definitions of the functions D and $\oplus$ in the proof. Recall that the recursion for $\oplus$ goes over its first argument, which is x in this case.

The induction principle tells us that it is sufficient to prove the following two properties (these are the premises of the induction principle instantiated with the concrete A that we defined above):

(1) $\forall y \in N : D(\langle\rangle \oplus y) = D(\langle\rangle) + D(y)$

(2) $\forall x \in N : (\forall y \in N : D(x \oplus y) = D(x) + D(y))$
$\Rightarrow (\forall y \in N : D(\langle x \rangle + y) = D(\langle x \rangle) + D(y))$

It is not difficult to prove these properties.

To minimize the amount of writing we have to do (and to improve clarity), we follow a specific pattern when we write induction proofs. We demonstrate this pattern in the proof below.

Lemma 2.1. $\forall x \in N : \forall y \in N : D(x \oplus y) = D(x) + D(y)$

Proof. By structural induction over $x \in N$:

Let $x = \langle\rangle$ and $y \in N$. Then

$$\begin{aligned} D(x \oplus y) &= D(\langle\rangle + y) \\ &= D(y) && \text{Definition of } \oplus \\ &= 0 + D(y) && 0 \text{ neutral element of } + \\ &= D(\langle\rangle) + D(y) && \text{Definition of } D \\ &= D(x) \oplus D(y) \end{aligned}$$

Let $x = \langle x' \rangle$ and $y \in N$. Then

$$\begin{aligned} D(x \oplus y) &= D(\langle x' \rangle \oplus y) \\ &= D(\langle x' \oplus y \rangle) && \text{Definition of } \oplus \\ &= 1 + D(x' \oplus y) && \text{Definition of } D \\ &= 1 + (D(x') + D(y)) && \text{Induction hypothesis} \\ &= (1 + (D(x')) + D(y) && \text{Associativity of } + \\ &= D(\langle x' \rangle) + D(y) && \text{Definition of } D \\ &= D(x) + D(y) \end{aligned}$$

□

2.2.4 Well-founded Induction (optional)

Recursion and induction appear in many variants. The common idea behind all these variants can be formulated using the notion of well-founded relations.

Let X be a set and $\succ \subseteq X \times X$ a binary relation over X. A (possibly infinite) sequence $x_1, x_2, x_3, \ldots$ of elements in X is called a *descending chain* of $\succ$ if $x_{i-1} \succ x_i$ for all members x_i of the sequence, $i > 0$. The relation $\succ$ is called *well-founded* if it has no infinite descending chains. We call a pair $(X, \succ)$ consisting of a set X and a well-founded relation $\succ$ on X a *well-founded set*.

Let $(X, \succ)$ be a well-founded set. With the notation $x \succ y$ we intuitively mean that x is in some sense larger than y. Well-foundedness then means that starting from any element $x_1 \in X$, we cannot find smaller and smaller elements $x_1 \succ x_2 \succ x_3 \succ \cdots$. At some point a long the chain, we must arrive at an element x_n such that no other element in X is smaller than x_n.

As an example, let us reconsider the set N. The relation

$$\succ_N : N \times N$$
$$x \succ_N y \overset{\text{def}}{\iff} x = \langle y \rangle$$

is a well-founded relation on N.[6] The reflexive and transitive closure of $\succ_N$ corresponds to the canonical ordering on natural numbers $\geq \subseteq \mathbb{N} \times \mathbb{N}$.

Let $(X, \succ)$ be a well-founded set and $M \subseteq X$. An element $x \in M$ is called *minimal element* of M if there exists no $y \in M$ such that $x \succ y$.

Lemma 2.2. *Let $(X, \succ)$ be a well-founded set. Then every nonempty subset of X has at least one minimal element.*

Proof. By contradiction. Let M be a nonempty subset of X that has no minimal element. Then there exists for every $x \in M$ some $y \in M$ such that $x \succ y$. Since M contains at least one element, we can construct an infinite descending chain of elements in M, and thus in X. It follows that $\succ$ is not well-founded. Contradiction. □

Let $(X, \succ)$ be a well-founded set and $A(x)$ a property for $x \in X$. We call the following inference rule the *well-founded induction principle*[7] for X, $\succ$, and A:

$$\frac{\forall x \in X : (\forall y \in X : x \succ y \Rightarrow A(y)) \Rightarrow A(x)}{\forall x \in X : A(x)}$$

The following theorem states the correctness of this rule.

Theorem 2.3. *Let $(X, \succ)$ be a well-founded set and $A(x)$ a property of $x \in X$. If*

$$\forall x \in X : (\forall y \in X : x \succ y \Rightarrow A(y)) \Rightarrow A(x)$$

then for all $x \in X$, $A(x)$ holds.

[6] This follows from our definition of tuples in terms of sets and the *axiom of foundation* of set theory.

[7] Sometimes, well-founded induction is also called Noetherian induction, named after the mathematician Emmy Noether.

Proof. By contradiction. Let M be the subset of X that contains all elements for which A does not hold. We assume that M is nonempty. Then we can choose a minimal element x_0 in M according to Lemma 2.2. Since M contains all elements of X for which A does not hold, it follows that

$$\forall y \in X : x_0 \succ y \Rightarrow A(y)$$

Then $A(x_0)$ follows from the premise of the induction rule. Contradiction. $\square$

The rule for well-founded induction only has a single premise and does not distinguish between base case and induction step. This is possible due to the more general formulation of the induction hypothesis in the premise of this rule. When we instantiate the rule for a particular well founded set $(X, \succ)$, then the base case and induction step typically emerge from further case analysis. For example, the induction principle for N is obtained from the well-founded induction principle by instantiating the latter with N for X and $\succ_N$ for $\succ$. The premise of the instantiated rule is:

$$\forall x \in N : (\forall y \in N : x \succ_N y \Rightarrow A(y)) \Rightarrow A(x)$$

To see that this premise is equivalent to the two premises of the induction principle for N, let us first replace $\succ_N$ by its definition:

$$\forall x \in N : (\forall y \in N : x = \langle y \rangle \Rightarrow A(y)) \Rightarrow A(x)$$

Now, in order to prove this premise for a particular A, we have to distinguish the two possible cases how each $x \in N$ was constructed, corresponding to the structurally recursive definition of N. This gives us two cases that we need to consider:

1. $(\forall y \in N : \langle\rangle = \langle y \rangle \Rightarrow A(y)) \Rightarrow A(\langle\rangle)$
2. $\forall x' \in N : (\forall y \in N : \langle x' \rangle = \langle y \rangle \Rightarrow A(y)) \Rightarrow A(\langle x' \rangle)$

The first case simplifies to $A(\langle\rangle)$ (i.e., the first premise of the induction principle for N) because $\langle\rangle = \langle y \rangle$ does not hold for any $y \in N$ and hence the left side of the outer implication is trivially true. The second case simplifies to

$$\forall x' \in N : (\forall y \in N : x' = y \Rightarrow A(y)) \Rightarrow A(\langle x' \rangle)$$

which can be further simplified to just

$$\forall x' \in N : A(x') \Rightarrow A(\langle x' \rangle) \ .$$

Renaming x' to x yields the second premise of the induction principle for N.

Note that well-founded relations are also closely related to the notion of termination measures that we discussed in Section 1.3.1.

2.3 Lists

Lists are one of the most important data structures in functional programming languages. We will consider lists that are sequences of data values of some common element type, e.g., a sequence of integer numbers $3, 6, 1, 2$. Unlike linked lists, which you have studied in your Data Structures course, lists in functional programming languages are *immutable*. That is, once a list has been created, it cannot be changed, e.g. by removing or adding an element in the middle of the list. Such data structures are also called *persistent*.

Persistent data structures have the advantage that their representation in memory can be shared across different instances of the data structure. For example, the two lists $1, 4, 3$ and $5, 2, 4, 3$ have the common sublist $4, 3$. If the two lists are stored in memory at the same time, the shared sublist $4, 3$ only needs to be represented once. If used properly, this feature yields space-efficient, high-level implementations of algorithms over persistent lists. In this section, we will define persistent lists using structural recursion and see that this definition corresponds to the `List` data type defined in Scala's standard library.

2.3.1 Defining Lists using Structural Recursion

Mathematically, we can represent lists of integers as nested tuples. For example, the empty list is represented by the empty tuple $\langle\rangle$, and the list containing the sequence of numbers 5, 2, and 3 is represented by the tuple $\langle 5, \langle 2, \langle 3, \langle\rangle\rangle\rangle\rangle$. The following structurally recursive definition formalizes this idea:

$$\langle\rangle \in \mathit{List} \qquad\qquad \frac{\mathit{hd} \in \mathbb{Z} \qquad \mathit{tl} \in \mathit{List}}{\langle \mathit{hd}, \mathit{tl}\rangle \in \mathit{List}}$$

For a non-empty list ℓ of the form $\langle \mathit{hd}, \mathit{tl}\rangle$, we refer to the integer number hd as the *head* of ℓ, and we call the remaining list tl the *tail* of ℓ. For example, the head of the list $\langle 4, \langle 2, \langle\rangle\rangle\rangle$ is 4 and its tail is $\langle 2, \langle\rangle\rangle$. We also refer to a non-empty list as a *cons cell*. To improve readability, we denote the empty list $\langle\rangle$ by *nil*.

In Scala, we can define lists of integers using an algebraic data type[8]:

```
enum List:
  case Nil
  case Cons(hd: Int, tl: List)
```

Here, `Nil` represents the empty list and a cons cell $\langle \mathit{hd}, \mathit{tl}\rangle$ is represented by `Cons(hd, tl)`. Note again the close resemblance between the mathematical definition and the Scala definition of lists.

As an example, let us construct a Scala list containing the values $1, 4, 2$:

```
scala> val l = Cons(1, Cons(4, Cons(2, Nil)))
val l: List = Cons(1, Cons(4, Cons(2, Nil)))
```

[8] The Scala standard library actually provides a generic `List` type that is parametric in its element type. The definition of this type is similar to the one that we give here. We will study Scala's `List` type more closely later.

We can also use pattern matching to deconstruct lists into their components:

```
scala> val Cons(h, t) = l
val h: Int = 1
val t: List = Cons(4, Cons(2, Nil))

scala> l match
         case Nil => println("l␣is␣empty")
         case Cons(h, t) => println(s"l's␣head␣is␣$h.")
l's␣head␣is␣1.
```

2.3.2 Functions on Lists

Using structural recursion we can now define simple functions on lists. For example, the following function computes the length of a given list:

$$\begin{aligned}
&\mathit{length} : \mathit{List} \to \mathbb{N}\\
\mathit{length}(\mathit{nil}) &= 0\\
\mathit{length}(\langle \mathit{hd}, \mathit{tl}\rangle) &= 1 + \mathit{length}(\mathit{tl})
\end{aligned}$$

The next function is more interesting, it takes two lists ℓ_1 and ℓ_2 and creates a new list by concatenating ℓ_1 and ℓ_2.

$$\begin{aligned}
&\mathit{append} : \mathit{List} \times \mathit{List} \to \mathit{List}\\
\mathit{append}(\mathit{nil}, \ell_2) &= \ell_2\\
\mathit{append}(\langle \mathit{hd}, \mathit{tl}\rangle, \ell_2) &= \langle \mathit{hd}, \mathit{append}(\mathit{tl}, \ell_2)\rangle
\end{aligned}$$

For example, for $\ell_1 = \langle 4, \langle 6, \langle 1, \langle\rangle\rangle\rangle\rangle$ and $\ell_2 = \langle 5, \langle 1, \mathit{nil}\rangle\rangle$ we get

$$\mathit{append}(\ell_1, \ell_2) = \langle 4, \langle 6, \langle 1, \langle 5, \langle 1, \mathit{nil}\rangle\rangle\rangle\rangle\rangle \ .$$

Finally, using *append* we can define a function *reverse* that takes a list ℓ and creates a new list that contains the elements of ℓ in reverse order:

$$\begin{aligned}
&\mathit{reverse} : \mathit{List} \to \mathit{List}\\
\mathit{reverse}(\mathit{nil}) &= \mathit{nil}\\
\mathit{reverse}(\langle \mathit{hd}, \mathit{tl}\rangle) &= \mathit{append}(\mathit{reverse}(\mathit{tl}), \langle \mathit{hd}, \mathit{nil}\rangle)
\end{aligned}$$

For example, we have $\mathit{reverse}(\langle 4, \langle 2, \mathit{nil}\rangle\rangle) = \langle 2, \langle 4, \mathit{nil}\rangle\rangle$.

Note that the definition of *reverse* is still structurally recursive since in the recursive case *reverse* is only applied to the tail *tl* of the input list.

The mathematical definitions of *length*, *append*, and *reverse* directly translate to corresponding Scala functions:

```
def length(l: List): Int = l match
  case Nil => 0
  case Cons(hd, tl) => 1 + length(tl)
```

```
def append(l1: List, l2: List): List = l1 match
  case Nil => l2
  case Cons(hd, tl) => Cons(hd, append(tl, l2))

def reverse(l: List): List = l match
  case Nil => Nil
  case Cons(hd, tl) => append(reverse(tl), Cons(hd, Nil))
```

Unfortunately, these Scala functions are not very efficient. For example, the running time of `reverse` is quadratic in the length of the list `l`. Moreover, the `reverse` function is not tail-recursive and hence requires linear space in the length of `l`. The implementations of `length` and `append` are also not tail-recursive. While we typically do not care about computational efficiency when we define mathematical functions, we do care about it when we write programs. To obtain efficient implementations we would rather implement these functions tail-recursively. For example, we can rewrite `reverse` so that it runs in linear time and constant space:

```
def reverse2(l: List): List =
  def rev(l: List, acc: List): List = l match
    case Nil => acc
    case Cons(h, t) => rev(t, Cons(h, acc))
  rev(l, Nil)
```

Exercise 2.1. *Give the mathematical definition of the tail-recursive* `reverse2` *function. Call this function* $reverse_2$*. Hint: to define* $reverse_2$*, first define an auxiliary function* $rev : List \times List \to List$*.*

Exercise 2.2. *Define tail-recursive Scala versions of the functions* `length` *and* `append`*. Hint: use* `reverse2` *in the definition of* `append`*.*

2.3.3 Proving Properties of Functions on Lists

Lists are defined by structural recursion. Hence, we can use structural induction to prove properties about functions (and programs) that operate on lists. Following the discussion in section 2.2.3, we derive the following structural induction principle for *List*:

$$\frac{A(nil) \qquad \forall hd \in \mathbb{N}, tl \in List : A(tl) \Rightarrow A(\langle hd, tl \rangle)}{\forall \ell \in List :\ A(\ell)}$$

As an example, the following proposition states that the length of a list obtained by appending two lists ℓ_1 and ℓ_2 is equal to the sum of the lengths of ℓ_1 and ℓ_2.

Proposition 2.4. *For all* $\ell_1, \ell_2 \in List$ *the following property holds*

$$length(append(\ell_1, \ell_2)) = length(\ell_1) + length(\ell_2)\ .$$

Proof. Let $\ell_1, \ell_2 \in \mathit{List}$. Since *append* is defined by structural recursion on its first argument, the proof proceeds by structural induction on ℓ_1: Base case: assume $\ell_1 = \mathit{nil}$. Then

$$\begin{aligned}
\mathit{length}(\mathit{append}(\ell_1, \ell_2)) &= \mathit{length}(\mathit{append}(\mathit{nil}, \ell_2)) & \\
&= \mathit{length}(\ell_2) & \text{Def. of } \mathit{append} \\
&= 0 + \mathit{length}(\ell_2) & \text{0 is neutral element} \\
&= \mathit{length}(\mathit{nil}) + \mathit{length}(\ell_2) & \text{Def. of } \mathit{length} \\
&= \mathit{length}(\ell_1) + \mathit{length}(\ell_2) &
\end{aligned}$$

Induction step: assume $\ell_1 = \langle hd, tl \rangle$. Then

$$\begin{aligned}
\mathit{length}(\mathit{append}(\ell_1, \ell_2)) &= \mathit{length}(\mathit{append}(\langle hd, tl \rangle, \ell_2)) & \\
&= \mathit{length}(\langle hd, \mathit{append}(tl, \ell_2) \rangle) & \text{Def. of } \mathit{append} \\
&= 1 + \mathit{length}(\mathit{append}(tl, \ell_2)) & \text{Def. of } \mathit{length} \\
&= 1 + (\mathit{length}(tl) + \mathit{length}(\ell_2)) & \text{Induction hypothesis} \\
&= (1 + \mathit{length}(tl)) + \mathit{length}(\ell_2) & \text{Associativity of } + \\
&= \mathit{length}(\langle hd, tl \rangle) + \mathit{length}(\ell_2) & \text{Def. of } \mathit{length} \\
&= \mathit{length}(\ell_1) + \mathit{length}(\ell_2) &
\end{aligned}$$

□

Exercise 2.3. *For the function* $\mathit{reverse}_2$ *that you defined in Exercise 2.1, use structural induction to prove that it computes the same function as* $\mathit{reverse}$*. That is, for all* $\ell \in \mathit{List}$*,* $\mathit{reverse}(\ell) = \mathit{reverse}_2(\ell)$*.*