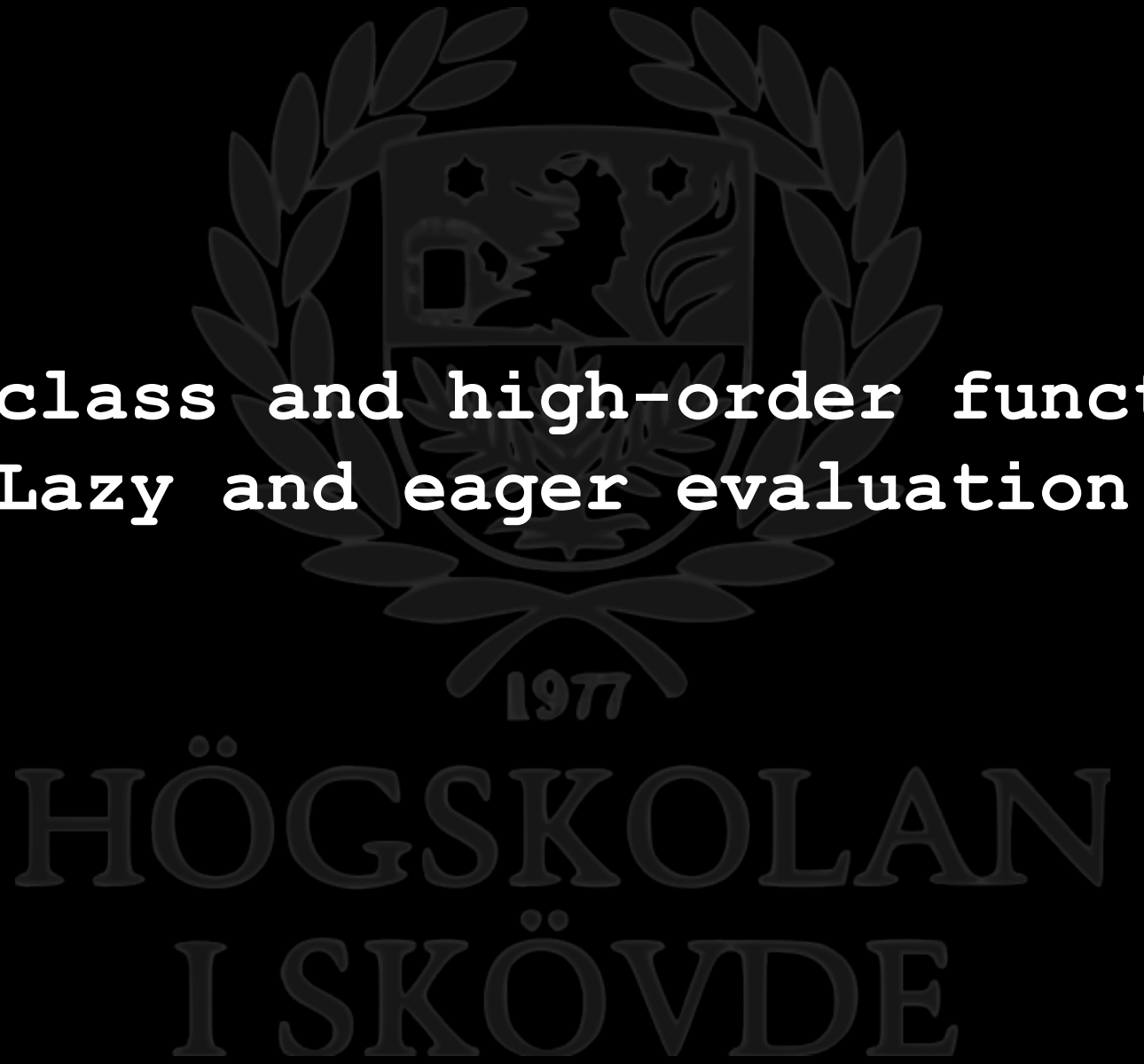
First-class and high-order functions. Lazy and eager evaluation



First-class functions

- May be called as a literal without an identifier.

```
(x: Int) => x * x
```

- May be contained by a value.

```
val pow = (x: Int) => x * x
```

- May be used as a parameter.

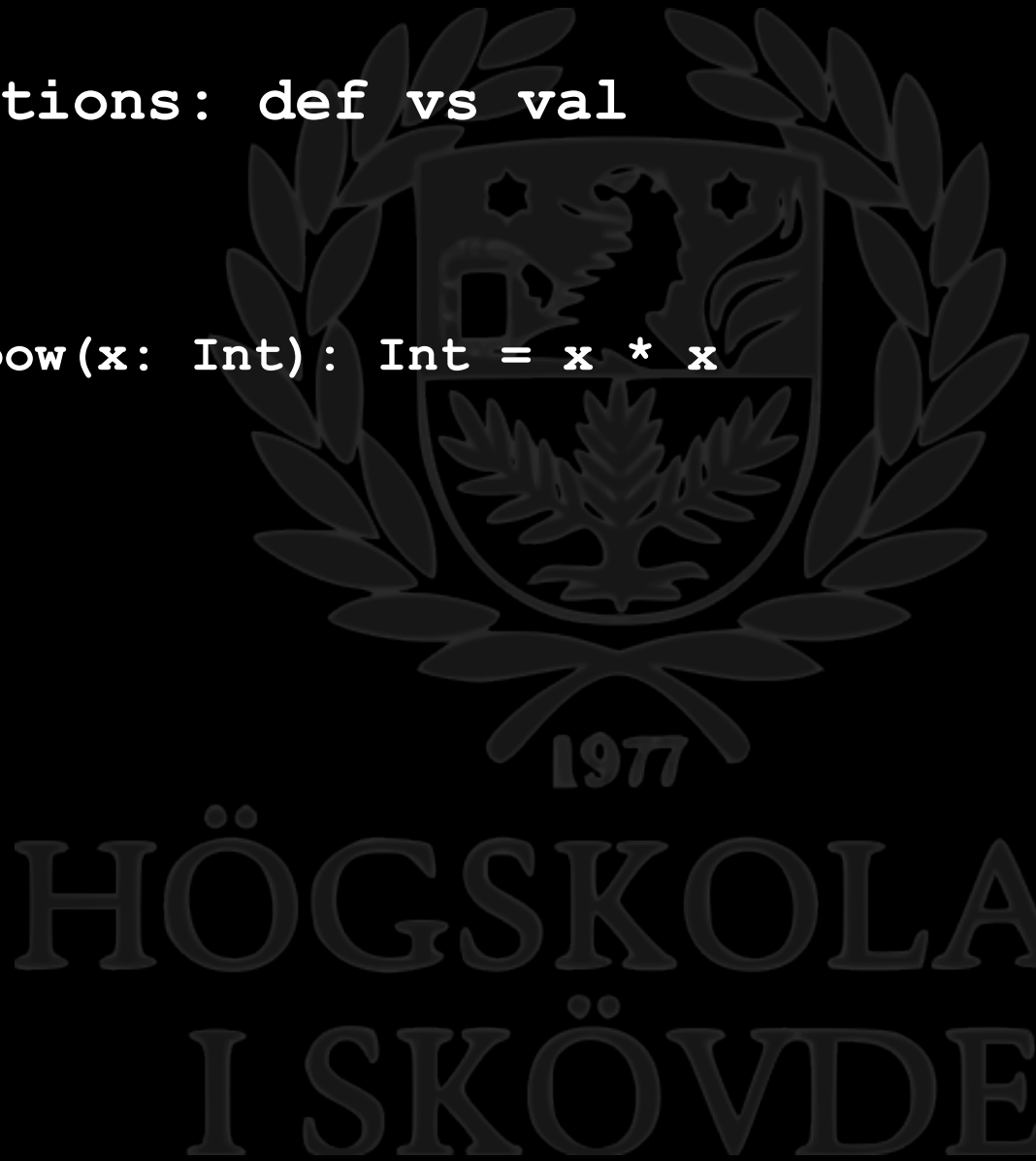
```
(f: Int => Int, x: Int) => f(x) * f(x)
```

- May be used as a return value.

```
(x: Int) => (y: Int) => x + y
```

Functions: def vs val

```
def pow(x: Int): Int = x * x
```

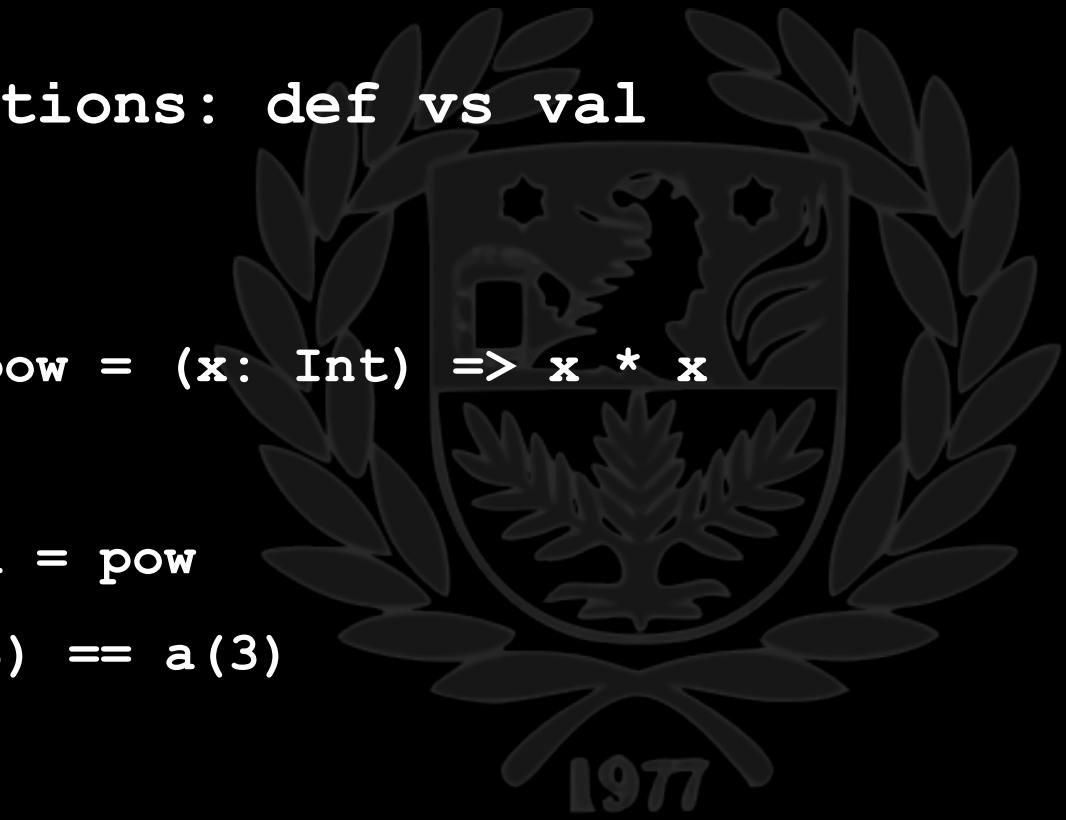


Functions: def vs val

```
def pow(x: Int): Int = x * x

val a = pow _

pow(3) == a(3)
```

Functions: def vs val

```
val pow = (x: Int) => x * x

val a = pow
pow(3) == a(3)
```



Functions: composition

```
val pow = (x: Int) => x * x

pow(pow(2)) // = 16
pow(pow(pow(2)))  // = 256
```

Functions: composition

```scala
val pow = (x: Int) => x * x

pow(pow(2)) == (pow andThen pow)(2) // = 16
pow(pow(pow(2))) = (pow andThen pow andThen pow)(2)    // = 256
```

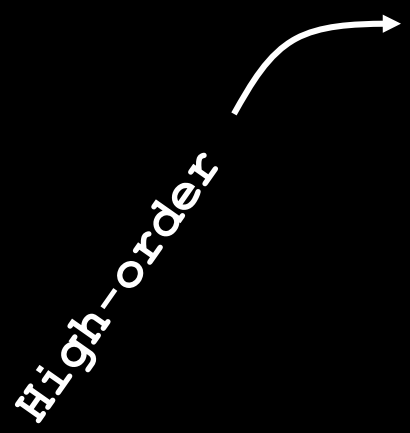
Functions: composition

```
val pow = (x: Int) => x * x
val plus1 = (x: Int) => x + 1

(plus1 andThen pow)(2)  // = pow(plus1(2)) = 9
(plus1 compose pow)(2)  // = plus1(pow(2)) = 5

(plus1 andThen pow)(2) == (pow compose plus1)(2)
```

Functions: composition

```
val times = (x: Int) => (y: Int) => x * y
val add = (x: Int) => (y: Int) => x + y
val substract = (x: Int) => (y: Int) => y - x

def myAge(shoeSize: Int, year: Int): Int =
  (substract(year) compose
  (times(5) andThen
  add(50) andThen
  times(20) andThen
  add(1017)))(shoeSize)
```

First class and high-order functions




```
def sum(x: Int, f: Int => Int): Int =
  f(x) + f(x)

def plus1(a: Int): Int = a + 1
sum(2, plus1)
```

First class and high-order functions

```
def sum(x: Int, f: Int => Int): Int =
  f(x) + f(x)

val plus1 = (a: Int) => a + 1
sum(2, plus1)
```

First-class and high-order functions

```
def sum(x: Int, f: Int => Int): Int =
  f(x) + f(x)


sum(2, (a: Int) => a + 1)
```

First-class and high-order functions

```
def sum(x: Int, f: Int => Int): Int =
  f(x) + f(x)


sum(2, a => a + 1)
```

First-class and high-order functions

```
def sum(x: Int, f: Int => Int): Int =
  f(x) + f(x)
```

```
sum(2, _ + 1)
```

Wildcard = Syntactic sugar

First-class and high-order functions

```scala
def compose(x: Int, f: Int => Int): Int = f(f(x))

val square = (y: Int) => y * y
compose(2, square)
```

First class and high-order functions

```
def compose(x: Int, f: Int => Int): Int = f(f(x))

val square = (y: Int) => y * y
compose(2, square)
compose(2, (y: Int) => y * y)(2)
compose(2,  y       => y * y)(2)
```

First class and high-order functions

```
def compose(n: Int, x: Int, f: Int => Int): Int = ???
```

```
val square = (y: Int) = x * x
def compose(1, 2, square)      // = square(2)
def compose(2, 2, square)      // = square(square(2))
def compose(3, 2, square)      // = square(square(square(2)))
```

First class and high-order functions

```scala
def square(l: List[Int]): List[Int] = l match {
  case Nil => ???
  case h :: t => ???
}

def map(l: List[Int], ???): List[Int] = l match {
  case Nil => Nil
  case h :: t => ???
}
```

First class and high-order functions

```
def sum(l: List[Int]): Int = l match {
  case Nil => ???
  case h :: t => ???
}

def reduce(l: List[Int], ???): Int = l match {
  case Nil => ???
  case h :: t => ???
}
```

First class and high-order functions

```
def squareAndSum(l: List[Int]): Int = l match {
  case Nil => ???
  case h :: t => ???
}

def mapReduce(l: List[Int], ???, ???): Int = l match {
  case Nil => ???
  case h :: t => ???
}
```

First class and high-order functions

```
def add(x: Int): Int => Int =
  n => n + x

val add2 = add(2)
val add3 = add(3)

add2(5)      // = ???
add3(5)      // = ???
```

First class and high-order functions

```
def pow(n: Int): Int => Int =
  if (n <= 1) x => x
  else x => x * pow(n - 1)(x)

val pow1 = pow(1)
val pow2 = pow(2)
val pow3 = pow(3)
```

First class and high-order functions

```
def compose(f: Int => Int): Int => Int =
  n => f(f(n))

val square = (x: Int) => x * x
compose(square)(2)
```

First class and high-order functions

```
def compose(f: Int => Int, n: Int): Int => Int = ???
```

```
compose(x => x * x, 2)(2)
```

Wildcards

```
def apply(x: Int, f: (Int, Int, Int) => Int): Int =
  f(x, x, x)

apply(4, (a, b, c) => a + b * c)    // 4 + 4 * 4
```

Wildcards

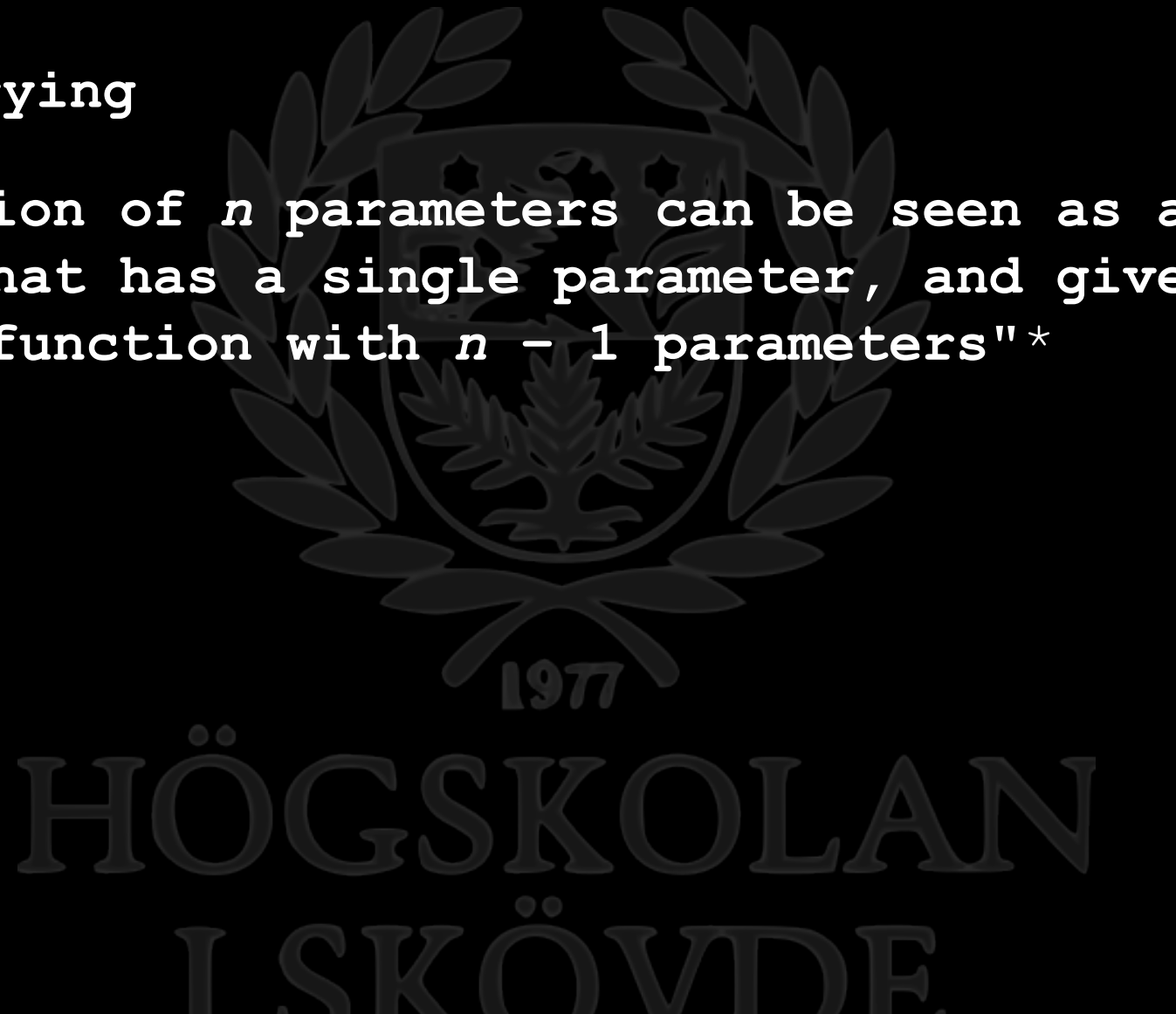
```
def apply(x: Int, f: (Int, Int, Int) => Int): Int =
  f(x, x, x)

apply(4, _ * _ + _)                    // 4 + 4 * 4
```

Wildcards

```
List(1, 2, 3, 4).filter(e => e % 2 == 0)
List(1, 2, 3, 4).filter(_ % 2 == 0)

List(1, 2, 3, 4).map(e => e.toString)
List(1, 2, 3, 4).map(_.toString)

List(1, 2, 3, 4).reduce((a, b) => a + b)
List(1, 2, 3, 4).reduce(_ + _)
```

Currying

"Any function of *n* parameters can be seen as a function that has a single parameter, and given it, it returns a function with *n* – 1 parameters"*



*Torra, V. (2016). Scala: From a functional programming perspective: An introduction to the programming language (Vol. 9980). Springer.

Currying

"Any function of *n* parameters can be seen as a function that has a single parameter, and given it, it returns a function with *n* – 1 parameters"*

```
val foo = (a: Int, b: Int, c: Int) => a + b + c
```


*Torra, V. (2016). Scala: From a functional programming perspective: An introduction to the programming language (Vol. 9980). Springer.

Currying

"Any function of *n* parameters can be seen as a function that has a single parameter, and given it, it returns a function with *n* – 1 parameters"*

```
val foo = (a: Int) =>
  (b: Int, c: Int) => a + b + c
```

*Torra, V. (2016). Scala: From a functional programming perspective: An introduction to the programming language (Vol. 9980). Springer.

Currying

"Any function of *n* parameters can be seen as a function that has a single parameter, and given it, it returns a function with *n* – 1 parameters"*

```
val foo = (a: Int) =>
  (b: Int) =>
    (c: Int) => a + b + c
```

*Torra, V. (2016). Scala: From a functional programming perspective: An introduction to the programming language (Vol. 9980). Springer.

Lazy and eager evaluation

```
val a = {
  println("hello!")
  "hello"
}
```

```
def b = {
  println("hello!")
  "hello"
}
```

```
lazy val c = {
  println("hello!")
  "hello"
}
```

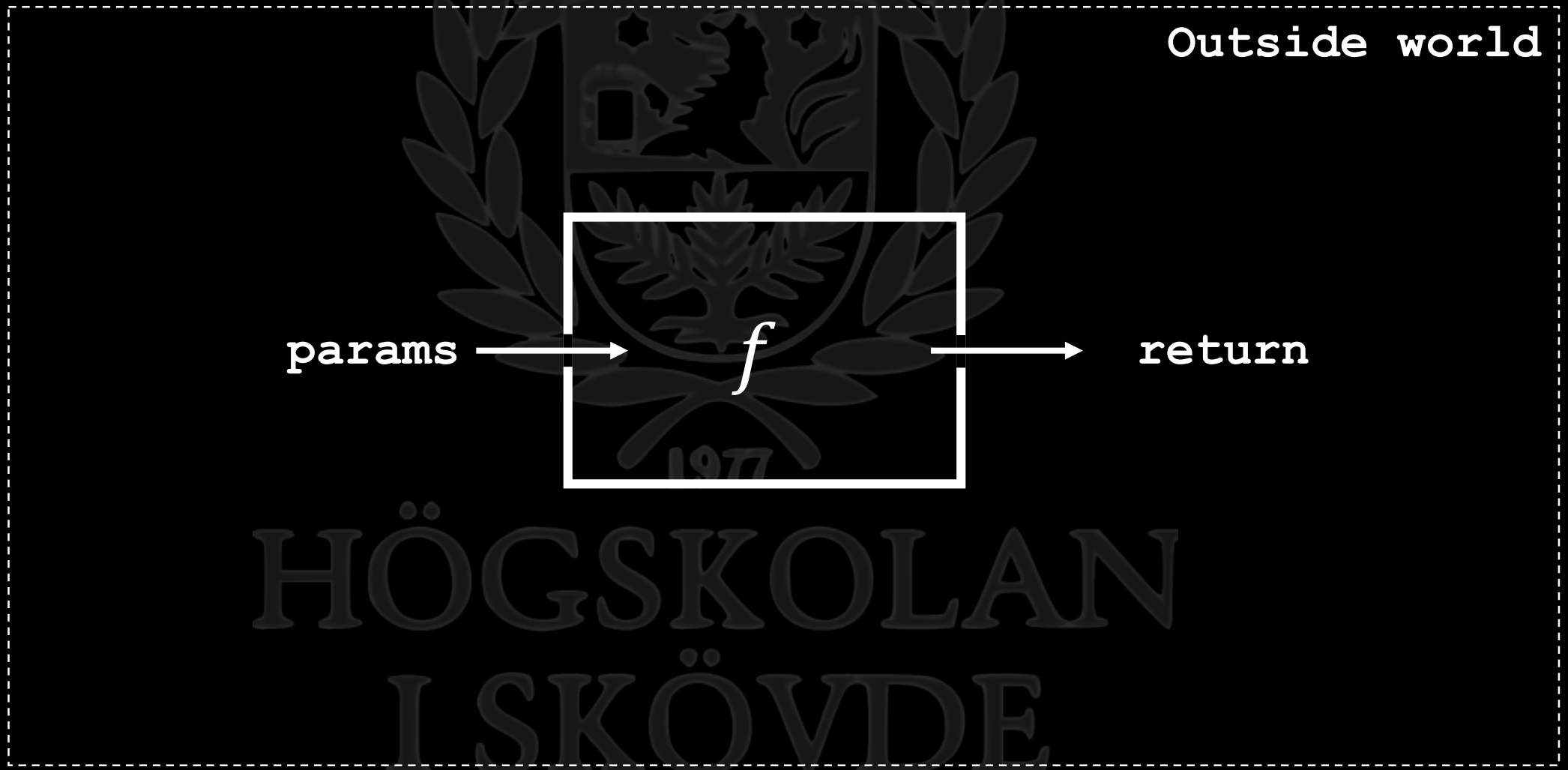
Lazy and eager evaluation


```
def pow2(a: => Int): Int = a + a

pow2({println("I am lazy"); 5})
```

Lazy and eager evaluation

```
def eval(execute: Boolean, code: => Int): Int =
  if (execute) code
  else -1

eval(false, 5 / 0)
```

Referential transparency



Referential transparency

"A property of functions that are independent of temporal context and have no side effects. For a particular input, an invocation of a referentially transparent function can be replaced by its result without changing the program semantics"*

"An expression *e* is referentially transparent with regard to a program p if every occurrence of *e* in *p* can be replaced by the result of evaluating *e* without affecting the meaning of *p*"**

*Odersky, M., Spoon, L., & Venners, B. (2010). Programming in Scala: A comprehensive step-by-step guide, Artima. Inc.,.

**Chiusano, P., & Bjarnason, R. (2014). Functional programming in Scala. Manning Publications Co..

No side effects + independent of context

A function does not have any direct influence on external states

A function is only influenced by its parameters and not by any other external state

```
var arr = Array(1, 2, 3)
def update(i: Int, x: Int) =
  arr(i) = x
```

No side effects + independent of context

A function does not have any direct influence on external states

A function is only influenced by its parameters and not by any other external state

```
var arr = Array(1, 2, 3)
def update(i: Int, x: Int) =
  arr(i) = x
```

```
def get(i: Int) =
  arr(i)
```