Week 4: Proofs, Generics, List Operators

CS-214 Software Construction

Outline

1. Proving Program Properties: Models, Induction, IntList
2. Proving properties on trees: IntTree
3. Automated proof checking
4. Parametric polymorphism, example of list and getting an element
5. Higher-order list functions defined on toy list
6. Scala lists: Construction, Patterns, Operations, isort
7. Tuples and generic methods, merge sort

Proving Program Properties

CS-214 Software Construction

Proof by Induction over Integers

Say we wish to prove that some property, P(n), holds for all integers starting from 0:

$P(0), P(1), P(2), \ldots$

We need to show all of the above instances.

$\boxed{\text{We start by proving P(0).}}$ This is *base case*.

Often property becomes harder the larger n is. Our strategy is, therefore, to show that the property *extends* from smaller to larger n:

$\boxed{\text{Given any } n \geq 0 \text{, if } P(n) \text{ then also } P(n+1).}$ This is *inductive case*.

Say we have shown base case and inductive case. Then the property holds for all n. As an example, take $n = 3$. We have shown $P(0)$. By inductive case for $n = 0$, from $P(0)$ it follows $P(1)$, then similarly from $P(1)$ follows $P(2)$, and from $P(2)$ follows $P(3)$.

Proofs over Integers Starting from a Given Bound

Let a be any integer. To prove $\forall n \geq a.\ P(n)$, it suffices to show two things:

- Base case: $P(a)$
- Inductive case: $\forall n \geq a.\ \ P(n) \Rightarrow P(n+1)$

It can be easier to prove $P(n+1)$ if we already know $P(n)$ holds.

$\boxed{\text{Example}}$ $P(n)$, for $n \geq 1$:

$$1 + \ldots + n = \frac{n(n+1)}{2}$$

- Base case, $P(1)$ becomes $1 = 1 \cdot (1+1)/2$
- Inductive case: suppose $1 + \ldots + n = n(n+1)/2$, then

$$1 + \ldots + n + (n+1) = \frac{n(n+1)}{2} + (n+1) \overset{IH}{=} (n+1)\left(\frac{n}{2} + 1\right) = \frac{(n+1)(n+2)}{2}$$

Inductive Proof about a Function

```
def sum(n: Int): Int =
  if n == 0 then 1
  else n + sum(n - 1)
```

Prove: $\boxed{\text{for every n >= 0, it holds that: sum(n) = n*(n+1)/2 + 1}}$

Base case: `n = 0` Use substitution model to expand `sum` on the left side:

```
sum(0) = if 0 == 0 then 1
           else 0 + sum(0 - 1)
        = 1
```

Use evaluation of arithmetic operations for the right side:

```
0*(0+1)/2 + 1 = 0*1/2 + 1 = 0/2 + 1 = 0 + 1 = 1
```

Induction Step of `sum(n) = n*(n+1)/2 + 1`

$\boxed{\textbf{Induction step: } \texttt{n+1}}$ (where n >= 0):

Assume Inductive Hypothesis (IH): `sum(n) = n*(n+1)/2 + 1`

```
def sum(n: Int): Int =
  if n == 0 then 1
  else n + sum(n - 1)
```

```
sum(n + 1)
= if n + 1 == 0 then 1
  else n + 1 + sum((n + 1) - 1)  // symbolic evaluation
= if false then 1                // n + 1 >= 1
  else n + 1 + sum(n + 1 - 1)
= n + 1 + sum(n + 1 - 1)         // if false
= n + 1 + sum(n)                 // cancel ‘+ 1 - 1‘
= n + 1 + n*(n+1)/2 + 1          // Inductive Hypothesis
= (2*n+2 + n*n+n))/2 + 1         // arithmetic
= (n+1)*((n+1)+1)/2 + 1          // arithmetic
```

Functions in Programs vs Math

Calling `sum` for large values of `n` may lead to outcomes for which proof does not apply:

- StackOverflow error. Stack is a data structure used to keep track of values of different recursive invocations of a function, but (especially in Java/Scala, has limited size), so deeply nested recursive calls result in error during execution.
- integer overflow: Int.MaxValue == 2147483647 but Int.MaxValue + 1 == -2147483648

Possibly approaches for correct reasoning:

- use a precise model of machine integers: map results of arithmetic operations to $[-2^{31}, 2^{31} - 1]$ modulo 2^{32}
- check that values are not too large, then the result is the same as in math
- use unbounded integers for computation (as in Python or Haskell)

Unbounded Integers: BigInt

```
def sum(n: BigInt): BigInt =   // using BigInt instead of Int as type
  if n == 0 then 1
  else n + sum(n - 1)
```

```
scala> val x: BigInt =  2147483647
val x: BigInt = 2147483647
scala> x + 1
val res0: BigInt = 2147483648
scala> x*x
val res1: BigInt = 4611686014132420609
scala> BigInt("7" * 60) + 1
val res2: BigInt = 777777777777777777777777777777777777777777777777777777777778
```

Exercise with sumAcc

Test the following function with large values of $n \geq 0$:

```
def sumAcc(s: BigInt, n: BigInt): BigInt = {
  if n == 0 then s
  else sumAcc(s + n, n - 1)
}
```

- Are you obtaining StackOverflow?
- Are you observing integer overflows?
- Propose a specification for the function and prove it by induction.
- Replace BigInt with Int, then find n such that sumAcc(1, n) returns a negative Int.

From Numbers to Lists

Lists are a generalization of non-negative integers. Take a list of Int-s.

```
sealed trait IntList
case object Nil extends IntList
case class Cons(head: Int, tail: IntList) extends IntList
```

For each non-negative integer *n*, there are (often many) lists of length *n*.

```
extension (xs: IntList)
  def length: BigInt =
    xs match
      case Nil => 0
      case Cons(h, t) => 1 + t.length
```

Concatenation (++) of Lists

```
extension (xs: IntList)
  def ++(ys: IntList): IntList =
    xs match
      case Nil => ys
      case Cons(h, t) => Cons(h, t ++ ys)
```

Example:

```
val l1 = Cons(1, Cons(2, Nil))
val l2 = Cons(10, Cons(20, Cons(30, Nil)))
l1 ++ l2  // Cons(1,Cons(2,Cons(10,Cons(20,Cons(30,Nil)))))
```

Concatenation (++) of Lists

```
extension (xs: IntList)
  def ++(ys: IntList): IntList =
    xs match
      case Nil => ys
      case Cons(h, t) => Cons(h, t ++ ys)
```

Example:

```
val l1 = Cons(1, Cons(2, Nil))
val l2 = Cons(10, Cons(20, Cons(30, Nil)))
l1 ++ l2  // Cons(1,Cons(2,Cons(10,Cons(20,Cons(30,Nil)))))
```

As a mathematical operation on sequences, which laws does ++ satisfy?

Concatenation (++) of Lists

```
extension (xs: IntList)
  def ++(ys: IntList): IntList =
    xs match
      case Nil => ys
      case Cons(h, t) => Cons(h, t ++ ys)
```

Example:

```
val l1 = Cons(1, Cons(2, Nil))
val l2 = Cons(10, Cons(20, Cons(30, Nil)))
l1 ++ l2  // Cons(1,Cons(2,Cons(10,Cons(20,Cons(30,Nil)))))
```

As a mathematical operation on sequences, which laws does ++ satisfy?

A small shorthand: `x :: xs` abbreviates `Cons(x, xs)`

Laws of ++ That Hold by Definition

Two cases in the pattern match

```
extension (xs: IntList)
  def ++(ys: IntList): IntList =
    xs match
      case Nil => ys                               // Nil case
      case Cons(h, t) => Cons(h, t ++ ys)          // Cons case
```

give us a two mathematical equations that hold by definition:

```
Nil ++ ys  =  ys                         // Nil case - Nil is a left-neutral element
(h :: t) ++ ys == h :: (t ++ ys)         // Cons case
```

Associativity and Neutral Element of ++

We would like to prove the following additional laws:

```
xs ++ Nil  =  xs                             // Nil is right-neutral element
(xs ++ ys) ++ zs  =  xs ++ (ys ++ zs)        // concatenation is associative
```

We will use *structural induction* on lists.

This principle is analogous to induction on non-negative integers.

Structural Induction on Lists

To prove a property $\mathtt{P(xs)}$ for all lists $\mathtt{xs}$,

- show that $\mathtt{P(Nil)}$ holds (*base case*),
- for an arbitrary list $\mathtt{xs}$ and an element $\mathtt{x}$, show the *induction step*: *if* $\mathtt{P(xs)}$ *holds, then* $\mathtt{P(x :: xs)}$ *also holds.*

Consider any list $x_n :: (x_{n-1} \dots :: (x_1 :: \mathtt{Nil}) \dots)$. Then

$$\mathtt{P(Nil)} \Rightarrow \mathtt{P}(x_1 :: \mathtt{Nil}) \Rightarrow \mathtt{P}(x_2 :: (x_1 :: \mathtt{Nil})) \Rightarrow \dots \Rightarrow \mathtt{P}(x_n :: (x_{n-1} :: \dots :: (x_1 :: \mathtt{Nil}) \dots)$$

Each implication holds by the induction step.

The first step holds by base case, so the conclusion follows.

It also follows from the fact that each list has a finite non-negative length; we can do induction on this length.

Proof: Nil is Right-Neutral Element of ++

Remember defining equations:

```
Nil ++ ys  =  ys                        // Nil case - Nil is a left-neutral element
(h :: t) ++ ys == h :: (t ++ ys)        // Cons case
```

We show by induction on xs that `xs ++ Nil = xs`

Base case:

```
  Nil ++ Nil = Nil                 // by Nil case
```

Induction Hypothesis: `xs ++ Nil = xs`

```
  (x :: xs) ++ Nil
  = x :: (xs ++ Nil)               // by Cons case
  = x :: xs                        // by Induction Hypothesis
```

Proof: Nil is Right-Neutral Element of ++

Remember defining equations:

```
Nil ++ ys   =  ys                         // Nil case - Nil is a left-neutral element
(h :: t) ++ ys == h :: (t ++ ys)          // Cons case
```

We show by induction on `xs` that `xs ++ Nil` $=$ `xs`

Base case:

```
  Nil ++ Nil = Nil                   // by Nil case
```

Induction Hypothesis: `xs ++ Nil` $=$ `xs`

```
  (x :: xs) ++ Nil
  = x :: (xs ++ Nil)                 // by Cons case
  = x :: xs                          // by Induction Hypothesis
```

Proof: Nil is Right-Neutral Element of ++

Remember defining equations:

```
Nil ++ ys  =  ys                        // Nil case - Nil is a left-neutral element
(h :: t) ++ ys == h :: (t ++ ys)        // Cons case
```

We show by induction on `xs` that `xs ++ Nil = xs`

Base case:

```
  Nil ++ Nil = Nil                      // by Nil case
```

Induction Hypothesis: `xs ++ Nil = xs`

```
  (x :: xs) ++ Nil
  = x :: (xs ++ Nil)                    // by Cons case
  = x :: xs                             // by Induction Hypothesis
```

Proving Associativity

Let us now show that, for any three lists `xs`, `ys`, `zs`:

```
  (xs ++ ys) ++ zs  =  xs ++ (ys ++ zs)
```

We again use the defining equations of $++$

```
Nil ++ ys  =  ys                          // Nil case - Nil is a left-neutral element
(h :: t) ++ ys == h :: (t ++ ys)          // Cons case
```

We use structural induction on `xs`.

Base Case

Proving

```
  (xs ++ ys) ++ zs  =  xs ++ (ys ++ zs)
```

Base case: xs is `Nil`

For the left-hand side we have:

```
 (Nil ++ ys) ++ zs
 =   ys ++ zs         // by Nil case
```

Base Case

Proving

```
(xs ++ ys) ++ zs  =  xs ++ (ys ++ zs)
```

Base case: xs is `Nil`

For the left-hand side we have:

```
(Nil ++ ys) ++ zs
=   ys ++ zs          // by Nil case
```

For the right-hand side, we have:

```
Nil ++ (ys ++ zs)
=   ys ++ zs          // by Nil case, take ys in the case to be (ys ++ zs)
```

The base case is therefore proven.

Induction Step: Assume (xs ++ ys) ++ zs = xs ++ (ys ++ zs)

Induction step: first argument is `x :: xs`

For the left-hand side, we have:

```
((x :: xs) ++ ys) ++ zs
=   (x :: (xs ++ ys)) ++ zs    // by Cons case
=   x :: ((xs ++ ys) ++ zs)    // by Cons case
=   x :: (xs ++ (ys ++ zs))    // by Induction Hypothesis
```

For the right hand side we have:

```
(x :: xs) ++ (ys ++ zs)
=   x :: (xs ++ (ys ++ zs))    // by Cons case
```

Left side equals right side because they are both equal to the same thing:

```
((x :: xs) ++ ys) ++ zs = x :: (xs ++ (ys ++ zs))
```

Substitution Model Review: List length

```
def length: BigInt = xs match
  case Nil => 0                        // Nil case
  case Cons(h, t) => 1 + t.length      // Cons case
```

Defining equations:

```
Nil.length = 0                          // Nil case
(h :: t).length = 1 + t.length          // Cons case
```

Evaluation using substitution model:

```
  (42 :: (69 :: Nil)).length
= 1 + (69 :: Nil).length               // Cons case
= 1 + (1 + Nil.length)                 // Cons case
= 1 + (1 + 0)                          // Nil case
= 2                                    // arithmetic
```

Symbolic Execution

Let x, y be arbitrary values. Then using the same steps:

```
    (x :: (y :: Nil)).length
  = 1 + (y :: Nil).length             // Cons case
  = 1 + (1 + Nil.length)              // Cons case
  = 1 + (1 + 0)                       // Nil case
  = 2                                 // arithmetic
```

We proved the following theorem: for every x, y

```
    (x :: (y :: Nil)).length == 2
```

We can hence extend the substitution model to work not only with concrete inputs to functions, but also symbolic values and establish certain simple theorems. We call this symbolic execution.

Symbolic Execution

Let x, y be arbitrary values. Then using the same steps:

```
  (x :: (y :: Nil)).length
= 1 + (y :: Nil).length             // Cons case
= 1 + (1 + Nil.length)              // Cons case
= 1 + (1 + 0)                       // Nil case
= 2                                 // arithmetic
```

We proved the following theorem: for every x, y

```
  (x :: (y :: Nil)).length == 2
```

We can hence extend the substitution model to work not only with concrete inputs to functions, but also symbolic values and establish certain simple theorems. We call this symbolic execution.

Summary: Where Valid Equations Come From

1. Symbolic execution of functions (using defining equations)
2. reflexivity: `E = E`
3. symmetry: if `E = F` then `F = E`
4. transitivity: if `E = F` and `F = G`, then `E = G` (chaining: `E = F = G`)
5. instantiation: if `A = B` holds for all values of symbolic `x`, then
   `A[C/x] = B[C/x]` where `C` is any expression denoting a value.
   Example: if `(x :: (y :: Nil)).length = 2` then `(42 :: (y :: Nil)).length = 2`
6. substituting `E` by its equal `F` in another equality: if `E = F` and `A = B` then also
   `A[F/E] = B[F/E]`.
   Example: if `f(x) = x + 1` then `3 + f(x) = 3 + x + 1`
7. Using structural induction to prove new equalities

Structural Induction on Trees

CS-214 Software Construction

Structural Induction on Trees

Structural induction is not limited to lists; it applies to any tree structure.

The general induction principle is the following:

To prove a property `P(t)` for all trees `t` of a certain type,

- show that `P(lf)` holds for all leaves `lf` of a tree,
- for each type of internal node `t` with subtrees $s_1, \ldots, s_n$, show that $P(s_1) \wedge \ldots \wedge P(s_n)$ *implies* $P(t)$.

Example: IntSets

Define binary trees that store sets of integers, faster than linear search:

```
sealed trait IntSet
case object Leaf extends IntSet
case class Node(left: IntSet, elem: Int, right: IntSet) extends IntSet

extension (s: IntSet)
  def contains(x: Int): Boolean =  // is an element in the set?
    s match
      case Leaf => false
      case Node(l, e, r) =>
        if x < e then l.contains(x)
        else if x > e then r.contains(x)
        else true
```

Inserting into InSet: Preserving the Order

```
extension (s: IntSet)
  def incl(x: Int): IntSet =  // a new set extended with a given element
    s match
      case Leaf => Node(Leaf, x, Leaf)
      case Node(l, e, r) =>
        if x < e then Node(l.incl(x), e, r)
        else if x > e then Node(l, e, r.incl(x))
        else s
```

The Laws of IntSet

What does it mean to prove the correctness of this implementation?

One way to define and show the correctness of an implementation consists of proving the laws that it respects.

In the case of `IntSet`, the following three laws of interest:

For any set `s`, and elements `x` and `y`:

```
Leaf.contains(x)          =  false
s.incl(x).contains(x)     =  true              // included x is present
s.incl(x).contains(y)     =  s.contains(y),   if x != y
```

This will show that IntSet behaves as a set with `incl` as operation that inserts an element into the set, and `contains` as set membership relation.

The first law follows by symbolic execution. We prove the other two by induction.

Included Present: `s.incl(x).contains(x)`

Structural induction on `s`.

Base case: `Leaf` follows by symbolic execution:

```
Leaf.incl(x).contains(x)
=   Node(Leaf, x, Leaf).contains(x) // case Leaf of incl
=   true                            // case Node, x=e of contains
```

Induction step: `s = Node(l, e, r)`, assuming the inductive hypotheses:

```
l.incl(x).contains(x) = true
r.incl(x).contains(x) = true
```

We have three cases: $x < e$, $x > e$, $x = e$.

Included Present: `s.incl(x).contains(x)`, cases `x < e`, `x > e`

Induction step: `s = Node(l, e, r)` **where** `x < e`

```
Node(l, e, r).incl(x).contains(x)
=   Node(l.incl(x), e, r).contains(x) // case Node of incl, x < e
=   r.incl(x).contains(x)             // case Node of contains, x < e
=   true                              // Inductive Hypothesis
```

Induction step: `s = Node(l, e, r)` **where** `x > e` is analogous:

```
Node(l, e, r).incl(x).contains(x)
=   Node(l, e, r.incl(x)).contains(x) // case Node of incl, x > e
=   r.incl(x).contains(x)             // case Node of contains, x > e
=   true                              // Inductive Hypothesis
```

Included is Present: `s.incl(x).contains(x) = true`, case `x = e`

$\boxed{\textbf{Induction step: } \texttt{s = Node(l, e, r)}}$, `x = e`

```
Node(l, e, r).incl(x).contains(x)
=   Node(l, x, r).contains(x)        //  case Node of incl, x = e
=   true                             //  case Node of contains, x = e
```

This completes the proof by structural induction that, for all trees s and elements x,

```
s.incl(x).contains(x) = true
```

Others Remain Present

We now wish to prove the second key property of incl: if `x != y` then

```
s.incl(x).contains(y)  =  s.contains(y)
```

(Elements other than `x` remain in or out of the extended set, as in `s` itself.)

Proof is by structural induction on `s`. Assume that `x < y`

(The case `x > y` is analogous.)

Base case: `Leaf`

```
Leaf.incl(y).contains(x)
=   Node(Leaf, x, Leaf).contains(x)     // case Leaf of incl
=   Leaf.contains(x)                    // case Node of contains, e=x
```

Others Remain Present: Inductive Cases

Proving: `s.incl(x).contains(y)  =  s.contains(y)` when `x < y`

$\boxed{\textbf{Induction step: } \texttt{s = Node(l, e, r)}}$. Inductive hypotheses:

```
l.incl(x).contains(y)  =  l.contains(y)
r.incl(x).contains(y)  =  r.contains(y)
```

We distinguish five cases:

1. `e = y`
2. `e = x`
3. `e < x < y`
4. `x < e < y`
5. `x < y < e`

First Two Cases

$\boxed{\textbf{Induction step: } \texttt{Node(l, e, r)}}$, e = y, x < y (so x < e)

```
Node(l, e, r).incl(x).contains(y)
= Node(l.incl(x), e, r).contains(y)   // case Node of incl, x < e
= true                                // case Node of contains, y=e
= Node(l, e, r).contains(y)           // case Node of contains, y=e
```

$\boxed{\textbf{Induction step: } \texttt{Node(l, e, r)}}$, e = x, x < y (so e < y)

```
Node(l, e, r).incl(x).contains(y)
= Node(l, e, r).contains(y)         // case Node of incl, e=x
```

Case e < x < y

$\boxed{\textbf{Induction step: } \texttt{Node(l, e, r)}}$, e < x < y

```
Node(l, e, r).incl(x).contains(y)
= Node(l, e, r.incl(x)).contains(y)  // case Node of incl, e < x
= r.incl(x).contains(y)              // case Node of contains, e < y
= r.contains(y)                      // Induction Hypothesis
= Node(l, e, r).contains(y)          // case Node of contains, e < y
```

Case x < e < y

$\boxed{\textbf{Induction step: } \texttt{Node(l, e, r)}}$, x < e < y

```
Node(l, e, r).incl(x).contains(y)
= Node(l.incl(x), e, r).contains(y)  // case Node of incl, x < e
= r.contains(y)                      // case Node of contains, e < y
= Node(l, e, r).contains(y)          // case Node of contains, e < y
```

Case x < y < e

Induction step: `Node(l, e, r)`, `x < y < e`

```
Node(l, e, r).incl(x).contains(y)
= Node(l.incl(x), e, r).contains(y)  // case Node of incl, x < e
= l.incl(x).contains(y)              // case Node of incl, x < e
= l.contains(y)                      // Induction Hypothesis
= Node(l, e, r).contains(y)          // case Node of contains, y < e
```

These are all the cases, so the proposition is established.

Automated Proof Checking and Search

CS-214 Software Construction

Proof Checking

It is easy to make a mistake when doing long proofs with many cases.

Proof checkers are programs that examine proof steps and make sure that each step is valid according to rules of logic.

Proof checkers are a basis of *proof assistants*, such as:

- Coq proof assistant: `https://coq.inria.fr/`
- Isabelle proof assistant: `https://isabelle.in.tum.de/`
- HOL prover: `https://hol-theorem-prover.org/`
- Lean proof assistant: `https://lean-lang.org/`
- Lisa proof framework: `https://github.com/epfl-lara/lisa`

In exercises you will use a small Scala library that checks proofs such as the ones you saw today when you execute it. Internally, it relies on Lisa.

Proof Search

Programs that automatically and systematically search for proofs are called *Automated Theorem Provers* (ATPs).

First-Order Logic Provers are based on first-order logic with equality.

Examples include provers called: E, SPASS, Vampire

A proof format, collection of challenges and a competition: `https://www.tptp.org/`

SMT Solvers are implemented on top of SAT solvers.

Contain specialized algorithms for solving constraints in, e.g., linear arithmetic (Simplex), non-linear arithmetic, equality, theories of arrays, case classes, strings, ...

Examples SMT solvers: z3, cvc5, Princess (written in Scala)

A proof format, challanges, and a competition: `https://smt-lib.org/`

Program Verifiers

Use SMT solvers to automate proving properties of programs.

Examples:

- Stainless verifier (for Scala): `https://github.com/epfl-lara/stainless/`
- Dafny verifier for Dafny language, used in Amazon: `https://dafny.org/`
- F^* verifier for an ocaml-like functional language: `https://fstar-lang.org/`
- Liquid Haskell: a verifier for Haskell, `https://ucsd-progsys.github.io/liquidhaskell/`

Proof assistants can also be used to verify programs.

Generics (Parametric Polymorphism)

CS-214 Software Construction

Type Parameters

Instead of defining list only for `Int` as in `IntList`, we can define it in general.

For this, we use generics: type parameters.

Type parameters are written in square brackets, e.g. `[T]`

- correspond approximately to <T> of Java

We can now generalize IntList using a type parameter:

```
sealed trait List[T]
case class Nil[T]() extends List[T]  // made Nil a class to make T known
case class Cons[T](head: Int, tail: List[T]) extends List[T]
```

Many List Methods Do Not Care about Type

```
extension[T] (xs: List[T])
  def length: BigInt =
    xs match
      case Nil() => 0
      case Cons(h, t) => 1 + t.length

  def ++(ys: List[T]): List[T] =
    xs match
      case Nil() => ys
      case Cons(h, t) => Cons(h, t ++ ys)
```

Generic Functions

Like classes, functions can have type parameters.

```
def singleton[T](elem: T) = Cons[T](elem, Nil[T]())
```

We can then write:

```
singleton[Int](1)
singleton[Boolean](true)
```

Scala compiler can often deduce the correct type parameters from the value arguments of a function call, so we can write simply:

```
singleton(1)
singleton(true)
```

Exercise with Generics

Write a function nth that takes a list and an integer n and selects the n'th element of the list.

Elements are numbered from 0.

If index is outside the range from 0 up the the length of the list minus one, a IndexOutOfBoundsException should be thrown.

```
def nth[T](xs: List[T], n: Int): T =


  xs match
      case Nil() => throw new IndexOutOfBoundsException()
      case Cons(h, t) =>
        if n == 0 then h
        else nth(t, n - 1)
```

Exercise with Generics

Write a function `nth` that takes a list and an integer `n` and selects the `n`'th element of the list.

Elements are numbered from 0.

If index is outside the range from `0` up the the length of the list minus one, a `IndexOutOfBoundsException` should be thrown.

```
def nth[T](xs: List[T], n: Int): T =


  xs match
      case Nil() => throw new IndexOutOfBoundsException()
      case Cons(h, t) =>
        if n == 0 then h
        else nth(t, n - 1)
```

Types and Evaluation

Type parameters do not affect evaluation in Scala.

We can assume that all type parameters and type arguments are removed before evaluating the program.

This is also called *type erasure*.

Languages that use type erasure include Java, Scala, Haskell, ML, OCaml.

Some other languages keep the type parameters around at run time, for example, C#

Higher-Order List Functions

CS-214 Software Construction

Recurring Patterns for Computations on Lists

Examples in labs and exercises have shown that functions on lists often have similar structures.

We can identify several recurring patterns, like,

- transforming each element in a list in a certain way,
- retrieving a list of all elements satisfying a criterion,
- combining the elements of a list using an operator.

Scala allows programmers to write generic functions that such patterns once using polymorphic higher-order functions.

Proper use of higher-order functions *may* make code clearer and shorter.

Applying a Function to Elements of a List

A common operation is to transform each element of a list and then return the list of results.

For example, to multiply each element of a list by the same factor, you could write:

```
def scaleList(xs: List[Double], factor: Double): List[Double] = xs match
  case Nil      => xs
  case y :: ys => y * factor :: scaleList(ys, factor)
```

Mapping

This scheme can be generalized to the method map of the List class. A simple way to define map is as follows:

```
extension [T](xs: List[T])
  def map[U](f: T => U): List[U] = xs match
    case Nil      => xs
    case x :: xs => f(x) :: xs.map(f)
```

Using map, scaleList can be written more concisely.

```
def scaleList(xs: List[Double], factor: Double) =
  xs.map(x => x * factor)
```

We Need Type Parameters to Define map

In the boid lab so far, we avoided type parameters.

This required us to duplicate functions.

For example, you can apply `mapBoid` to many functions `f` but they all must accept a `Boid`:

```
def mapBoid(f: Boid => Boid): BoidSequence =
  this match
    case BoidNil()              => BoidNil()
    case BoidCons(head, tail) => BoidCons(f(head), tail.mapBoid(f))
```

Exercise

Consider a function to square each element of a list, and return the result. Complete the two following equivalent definitions of squareList.

```
def squareList(xs: List[Int]): List[Int] = xs match
  case Nil     => ???
  case y :: ys => ???

def squareList(xs: List[Int]): List[Int] =
  xs.map(???)
```

Exercise

Consider a function to square each element of a list, and return the result. Complete the two following equivalent definitions of `squareList`.

```
def squareList(xs: List[Int]): List[Int] = xs match
  case Nil     => Nil
  case y :: ys => y * y :: squareList(ys)

def squareList(xs: List[Int]): List[Int] =
  xs.map(x => x * x)
```

Filtering

Another common operation on lists is the selection of all elements satisfying a given condition. For example:

```
def posElems(xs: List[Int]): List[Int] = xs match
  case Nil     => xs
  case y :: ys => if y > 0 then y :: posElems(ys) else posElems(ys)
```

Filter

This pattern is generalized by the method `filter` of the `List` class:

```
extension [T](xs: List[T])
  def filter(p: T => Boolean): List[T] = this match
    case Nil      => this
    case x :: xs => if p(x) then x :: xs.filter(p) else xs.filter(p)
```

Using `filter`, `posElems` can be written more concisely.

```
def posElems(xs: List[Int]): List[Int] =
  xs.filter(x => x > 0)
```

Variations of Filter

Besides filter, there are also the following methods that extract sublists based on a predicate:

`xs.filterNot(p)`	Same as `xs.filter(x => !p(x))`; The list consisting of those elements of `xs` that do not satisfy the predicate `p`.
`xs.partition(p)`	Same as `(xs.filter(p), xs.filterNot(p))`, but computed in a single traversal of the list `xs`.
`xs.takeWhile(p)`	The longest prefix of list `xs` consisting of elements that all satisfy the predicate `p`.
`xs.dropWhile(p)`	The remainder of the list `xs` after any leading elements satisfying `p` have been removed.
`xs.span(p)`	Same as `(xs.takeWhile(p), xs.dropWhile(p))` but computed in a single traversal of the list `xs`.

Reduction of Lists

Another common operation on lists is to combine the elements of a list using a given operator.

For example:

```
sum(List(x1, ..., xn))         =  0 + x1 + ... + xn
product(List(x1, ..., xn))     =  1 * x1 * ... * xn
```

We can implement this with the usual recursive schema:

```
def sum(xs: List[Int]): Int = xs match
  case Nil      => 0
  case y :: ys => y + sum(ys)
```

ReduceLeft

This pattern can be abstracted out using the generic method `reduceLeft`:

`reduceLeft` inserts a given binary operator between adjacent elements of a list:

```
List(x1, ..., xn).reduceLeft(op)   =  x1.op(x2). ... .op(xn)
```

Using `reduceLeft`, we can simplify:

```
def sum(xs: List[Int])     = (0 :: xs).reduceLeft((x, y) => x + y)
def product(xs: List[Int]) = (1 :: xs).reduceLeft((x, y) => x * y)
```

A Shorter Way to Write Functions

Instead of `((x, y) => x * y))`, one can also write shorter:

```
(_ * _)
```

Every _ represents a new parameter, going from left to right.

The parameters are defined at the next outer pair of parentheses (or the whole expression if there are no enclosing parentheses).

So, `sum` and `product` can also be expressed like this:

```
def sum(xs: List[Int])     = (0 :: xs).reduceLeft(_ + _)
def product(xs: List[Int]) = (1 :: xs).reduceLeft(_ * _)
```

FoldLeft

The function `reduceLeft` is defined in terms of a more general function, `foldLeft`.

`foldLeft` is like `reduceLeft` but takes an *accumulator*, `z`, as an additional parameter, which is returned when `foldLeft` is called on an empty list.

```
List(x1, ..., xn).foldLeft(z)(op)   =  z.op(x1).op ... .op(xn)
```

So, `sum` and `product` can also be defined as follows:

```
def sum(xs: List[Int])      =  xs.foldLeft(0)(_ + _)
def product(xs: List[Int])  =  xs.foldLeft(1)(_ * _)
```

Implementations of ReduceLeft and FoldLeft

`foldLeft` and `reduceLeft` can be implemented in class `List` as follows.

```
abstract class List[T]:

  def reduceLeft(op: (T, T) => T): T = this match
    case Nil     => throw IllegalOperationException(”Nil.reduceLeft”)
    case x :: xs => xs.foldLeft(x)(op)

  def foldLeft[U](z: U)(op: (U, T) => U): U = this match
    case Nil     => z
    case x :: xs => xs.foldLeft(op(z, x))(op)
```

FoldRight and ReduceRight

Applications of `foldLeft` and `reduceLeft` create computation trees that lean to the left.

They have two dual functions, `foldRight` and `reduceRight`, which produce trees which lean to the right, e.g.,

```
List(x1, x2, x3).reduceRight(op)    = x1.op(x2.op(x3))
List(x1, x2, x3).foldRight(z)(op)   = x1.op(x2.op(x3.op(z)))
```

Implementation of FoldRight and ReduceRight

They are defined as follows

```
def reduceRight(op: (T, T) => T): T = this match
  case Nil      => throw UnsupportedOperationException(”Nil.reduceRight”)
  case x :: Nil => x
  case x :: xs  => op(x, xs.reduceRight(op))

def foldRight[U](z: U)(op: (T, U) => U): U = this match
  case Nil     => z
  case x :: xs => op(x, xs.foldRight(z)(op))
```

Difference between FoldLeft and FoldRight

For operators that are associative and commutative, `foldLeft` and `foldRight` are equivalent (even though there may be a difference in efficiency).

But sometimes, only one of the two operators is appropriate.

Exercise

```
def foldLeft[U](z: U)(op: (U, T) => U): U = this match
  case Nil      => z
  case x :: xs => xs.foldLeft(op(z, x))(op)

def foldRight[U](z: U)(op: (T, U) => U): U = this match
  case Nil      => z
  case x :: xs => op(x, xs.foldRight(z)(op))
```

Here is another formulation of `concat`:

```
def concat[T](xs: List[T], ys: List[T]): List[T] =
  xs.foldRight(ys)(_ :: _)
```

Here, it isn't possible to replace `foldRight` by `foldLeft`. What would go wrong?

Back to Reversing Lists

We now develop a function for reversing lists which has a linear cost.

The idea is to use the operation `foldLeft`:

```
def reverse[T](xs: List[T]): List[T] = xs.foldLeft(z?)(op?)
```

All that remains is to replace the parts `z?` and `op?`.

Let's try to *compute* them from examples.

Deduction of Reverse (1)

To start computing `z?`, let's consider `reverse(Nil)`.

We know `reverse(Nil) == Nil`, so we can compute as follows:

```
Nil
```

Deduction of Reverse (1)

To start computing `z?`, let's consider `reverse(Nil)`.

We know `reverse(Nil) == Nil`, so we can compute as follows:

```
Nil

=   reverse(Nil)
```

Deduction of Reverse (1)

To start computing `z?`, let's consider `reverse(Nil)`.

We know `reverse(Nil) == Nil`, so we can compute as follows:

```
Nil

=   reverse(Nil)

=   Nil.foldLeft(z?)(op)
```

Deduction of Reverse (1)

To start computing `z?`, let's consider `reverse(Nil)`.

We know `reverse(Nil) == Nil`, so we can compute as follows:

```
  Nil

=   reverse(Nil)

=   Nil.foldLeft(z?)(op)

=   z?
```

Consequently, `z? = Nil`

Deduction of Reverse (2)

We still need to compute `op?`. To do that let's plug in the next simplest list after `Nil` into our equation for `reverse`:

```
List(x)
```

Deduction of Reverse (2)

We still need to compute `op?`. To do that let's plug in the next simplest list after `Nil` into our equation for `reverse`:

```
List(x)

=   reverse(List(x))
```

Deduction of Reverse (2)

We still need to compute `op?`. To do that let's plug in the next simplest list after `Nil` into our equation for `reverse`:

```
List(x)

=    reverse(List(x))

=    List(x).foldLeft(Nil)(op?)
```

Deduction of Reverse (2)

We still need to compute `op?`. To do that let's plug in the next simplest list after `Nil` into our equation for `reverse`:

```
List(x)

=   reverse(List(x))

=   List(x).foldLeft(Nil)(op?)

=   op?(Nil, x)
```

Consequently, `op?(Nil, x) = List(x) = x :: Nil`.

This suggests to take for `op?` the operator `::` but with its operands swapped.

Deduction of Reverse(3)

We thus arrive at the following implementation of `reverse`.

```scala
def reverse[T](xs: List[T]): List[T] =
  xs.foldLeft(Nil)((xs, x) => x :: xs)
```

Exercise

Complete the following definitions of the basic functions `map` and `length` on lists, such that their implementation uses `foldRight`:

```
def mapFun[T, U](xs: List[T], f: T => U): List[U] =
  xs.foldRight(Nil)( ??? )

def lengthFun[T](xs: List[T]): Int =
  xs.foldRight(0)( ??? )
```

Exercise

Complete the following definitions of the basic functions `map` and `length` on lists, such that their implementation uses `foldRight`:

```
def mapFun[T, U](xs: List[T], f: T => U): List[U] =
  xs.foldRight(Nil)((y, ys) => f(y) :: ys)

def lengthFun[T](xs: List[T]): Int =
  xs.foldRight(0)((y, n) => n + 1)
```

More on Scala Lists

CS-214 Software Construction

More on Lists in Scala

The list is a fundamental data structure in functional programming. In Scala, Lists are defined in the standard library.

A list having $x_1, \ldots, x_n$ as elements is written `List(`$x_1, \ldots, x_n$`)`

Example

```
val fruit  = List("apples", "oranges", "pears")
val nums   = List(1, 2, 3, 4)
val diag3  = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
val empty  = List()
```

There are two important differences between lists and arrays.

- Lists are immutable — the elements of a list cannot be changed.
- Lists are recursive, while arrays are flat.

Lists

```
val fruit  = List(”apples”, ”oranges”, ”pears”)
val diag3  = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
```

The List Type

The type of a list with elements of type `T` is written `scala.List[T]` or shorter just `List[T]`

Each element of such list has type `T`

Example

```
val fruit: List[String]    = List("apples", "oranges", "pears")
val nums : List[Int]       = List(1, 2, 3, 4)
val diag3: List[List[Int]] = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
val empty: List[Nothing]   = List()
```

Constructors of Lists

All lists are constructed from:

- the empty list `Nil`, and
- the construction operation `::` (pronounced *cons*):
  `x :: xs` gives a new list with the first element `x`, followed by the elements of `xs`.

For example:

```
fruit = ”apples” :: (”oranges” :: (”pears” :: Nil))
nums  = 1 :: (2 :: (3 :: (4 :: Nil)))
empty = Nil
```

Right Associativity

Convention: Operators ending in ":" associate to the right.

`A :: B :: C` is interpreted as `A :: (B :: C)`.

We can thus omit the parentheses in the definition above.

Example

```
val nums = 1 :: 2 :: 3 :: 4 :: Nil
```

Operations on Lists

All operations on lists can be expressed in terms of the following three:

`head`	the first element of the list
`tail`	the list composed of all the elements except the first.
`isEmpty`	'true' if the list is empty, 'false' otherwise.

These operations are defined as methods of objects of type `List`. For example:

```
fruit.head      == "apples"
fruit.tail.head == "oranges"
diag3.head      == List(1, 0, 0)
empty.head      == throw NoSuchElementException("head of empty list")
```

Decomposing Lists using Patterns

```
myList match
  case p =>
```

Patterns `p` look analogous to operations to construct lists:

`Nil`	The `Nil` constant
`p :: ps`	A pattern that matches a list with a `head` matching `p` and a `tail` matching `ps`.
`List(p1, ..., pn)`	same as `p1 :: ... :: pn :: Nil`

Example

`1 :: 2 :: xs`	Lists of that start with `1` and then `2`
`x :: Nil`	Lists of length 1
`List(x)`	Same as `x :: Nil`
`List()`	The empty list, same as `Nil`
`List(2 :: xs)`	List that contains as only element a list starting with `2`

Exercise

Consider the pattern `x :: y :: List(xs, ys) :: zs`

What is the condition that describes most accurately the length `L` of the lists it matches?

O	`L == 3`
O	`L == 4`
O	`L == 5`
O	`L >= 3`
O	`L >= 4`
O	`L >= 5`

Exercise

Consider the pattern `x :: y :: List(xs, ys) :: zs`.

What is the condition that describes most accurately the length `L` of the lists it matches?

O	`L == 3`
O	`L == 4`
O	`L == 5`
X	`L >= 3`
O	`L >= 4`
O	`L >= 5`

Sorting Lists

Suppose we want to sort `List(7, 3, 9, 2)` in ascending order:

1. One way is to sort the tail, `List(3, 9, 2)` to obtain `List(2, 3, 9)`
2. Then, insert the head `7` in the right place to obtain `List(2, 3, 7, 9)`.

This idea describes *Insertion Sort* :

```
def isort(xs: List[Int]): List[Int] = xs match
  case List()  => List()
  case y :: ys => insert(y, isort(ys))
```

Exercise

Complete the definition insertion sort by filling in the ???s in the definition below:

```
def insert(x: Int, xs: List[Int]): List[Int] = xs match
  case List() => List(x)
  case y :: ys =>
```

Exercise

Complete the definition insertion sort by filling in the ???s in the definition below:

```
def insert(x: Int, xs: List[Int]): List[Int] = xs match
  case List() => List(x)
  case y :: ys =>
    if x < y then x :: xs else y :: insert(x, ys)
```

What is the asymptotic worst-case complexity of `isort` relative to the length of the input list N?

```
o      constant time: O(1)
o      linear time: O(N)
o      O(N * log(N))
o      O(N * N)
```

Exercise

Complete the definition insertion sort by filling in the ???s in the definition below:

```
def insert(x: Int, xs: List[Int]): List[Int] = xs match
  case List() => List(x)
  case y :: ys =>
    if x < y then x :: xs else y :: insert(x, ys)
```

What is the asymptotic worst-case complexity of `isort` relative to the length of the input list `N`?

```
o      constant time: O(1)
o      linear time: O(N)
o      O(N * log(N))
o      O(N * N)
```

Exercise

Complete the definition insertion sort by filling in the ???s in the definition below:

```
def insert(x: Int, xs: List[Int]): List[Int] = xs match
  case List() => List(x)
  case y :: ys =>
    if x < y then x :: xs else y :: insert(x, ys)
```

What is the asymptotic worst-case complexity of `isort` relative to the length of the input list `N`?

```
  o       constant time: O(1)
  o       linear time: O(N)
  o       O(N * log(N))
==>       O(N * N)
```

Lists so far: Recap

Lists are the core data structure we will work with over the next weeks.

Type: `List[Fruit]`

Construction:

```
val fruits = List(”Apple”, ”Orange”, ”Banana”)
val nums = 1 :: 2 :: Nil
```

Decomposition:

```
fruits.head      // ”Apple”
nums.tail        // 2 :: Nil
nums.isEmpty     // false

nums match
  case x :: y :: _ => x + y   // 3
```

List Methods (1)

Sublists and element access:

`xs.length`	The number of elements of `xs`.
`xs.last`	The list's last element, exception if `xs` is empty.
`xs.init`	A list consisting of all elements of `xs` except the last one, exception if `xs` is empty.
`xs.take(n)`	A list consisting of the first `n` elements of `xs`, or `xs` itself if it is shorter than `n`.
`xs.drop(n)`	The rest of the collection after taking `n` elements.
`xs(n)`	(or, written out, `xs.apply(n)`). The element of `xs` at index `n`.

List Methods (2)

Creating new lists:

`xs ++ ys`	The list consisting of all elements of `xs` followed by all elements of `ys`.
`xs.reverse`	The list containing the elements of `xs` in reversed order.
`xs.updated(n, x)`	The list containing the same elements as `xs`, except at index `n` where it contains `x`.

Finding elements:

`xs.indexOf(x)`	The index of the first element in `xs` equal to `x`, or -1 if `x` does not appear in `xs`.
`xs.contains(x)`	same as `xs.indexOf(x) >= 0`

Exercise

Remove the `n`'th element of a list `xs`. If `n` is out of bounds, return `xs` itself.

```
def removeAt[T](n: Int, xs: List[T]) = ???
```

For example:

```
removeAt(1, List('a', 'b', 'c', 'd'))
```

should give:

```
List(a, c, d)
```

Merge Sort. Tuples and Generic Methods

CS-214 Software Construction

Sorting Lists Faster

As a non-trivial example, let's design a function to sort lists that is more efficient than insertion sort.

A good algorithm for this is *merge sort*. The idea is as follows:

If the list consists of zero or one elements, it is already sorted.

Otherwise,

- Separate the list into two sub-lists, each containing around half of the elements of the original list.
- Sort the two sub-lists.
- Merge the two sorted sub-lists into a single sorted list.

First MergeSort Implementation

Here is the implementation of that algorithm in Scala:

```scala
def msort(xs: List[Int]): List[Int] =
  val n = xs.length / 2
  if n == 0 then xs
  else
    def merge(xs: List[Int], ys: List[Int]) = ???
    val (fst, snd) = xs.splitAt(n)
    merge(msort(fst), msort(snd))
```

The SplitAt Function

The `splitAt` function on lists returns two sublists

- the elements up the the given index
- the elements from that index

The lists are returned in a *pair*.

Detour: Pair and Tuples

The pair consisting of `x` and `y` is written `(x, y)` in Scala.

Example

```
val pair = (”answer”, 42)   > pair : (String, Int) = (answer,42)
```

The type of `pair` above is `(String, Int)`.

Pairs can also be used as patterns:

```
val (label, value) = pair   > label: String = answer, value: Int = 42
```

This works analogously for tuples with more than two elements.

If *p* is a pair, we can get its first element with `p._1`, second by `p._2`

Definition of Merge

Here is a definition of the `merge` function:

```
def merge(xs: List[Int], ys: List[Int]) = (xs, ys) match
  case (Nil, ys) => ys
  case (xs, Nil) => xs
  case (x :: xs1, y :: ys1) =>
    if x < y then x :: merge(xs1, ys)
    else y :: merge(xs, ys1)
```

Making Sort More General

Problem: How to parameterize `msort` so that it can also be used for lists with elements other than `Int`?

```
def msort[T](xs: List[T]): List[T] = ???
```

does not work, because the comparison < in `merge` is not defined for arbitrary types `T`.

Idea: Parameterize `merge` with the necessary comparison function.

Parameterization of Sort

The most flexible design is to make the function `sort` polymorphic and to pass the comparison operation as an additional parameter:

```
def msort[T](xs: List[T])(lt: (T, T) => Boolean) =
  ...
    merge(msort(fst)(lt), msort(snd)(lt))
```

Merge then needs to be adapted as follows:

```
def merge[T](xs: List[T], ys: List[T]) = (xs, ys) match
  ...
  case (x :: xs1, y :: ys1) =>
    if lt(x, y) then ...
    else ...
```

Calling Parameterized Sort

We can now call `msort` as follows:

```
val xs = List(-5, 6, 3, 2, 7)
val fruits = List(”apple”, ”pear”, ”orange”, ”pineapple”)

msort(xs)((x: Int, y: Int) => x < y)
msort(fruits)((x: String, y: String) => x.compareTo(y) < 0)
```

Or, since parameter types can be inferred from the call `msort(xs)`:

```
msort(xs)((x, y) => x < y)
```