Modularity à la ML

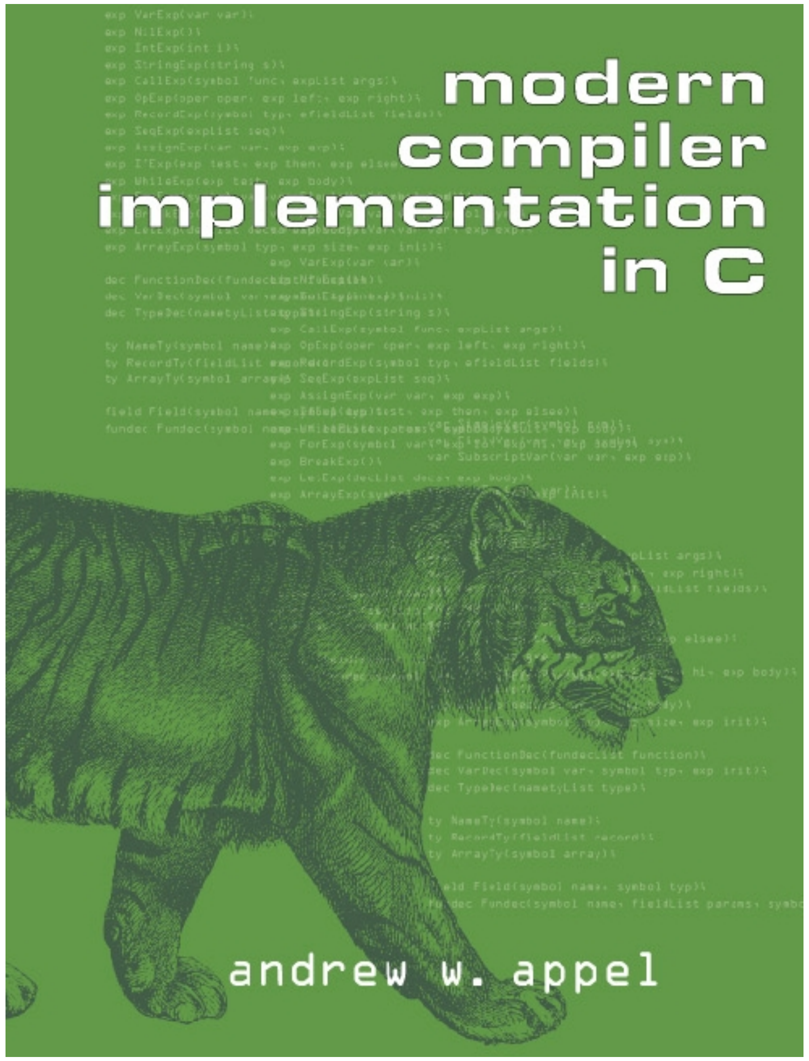
Ionuț G. Stan — igstan.ro — ionut@eloquentix.com

About Me

- Software Developer at eloquentix
- Worked with Scala for the past 5 years
- FP, programming languages, compilers
- Mostly-tech blog at igstan.ro

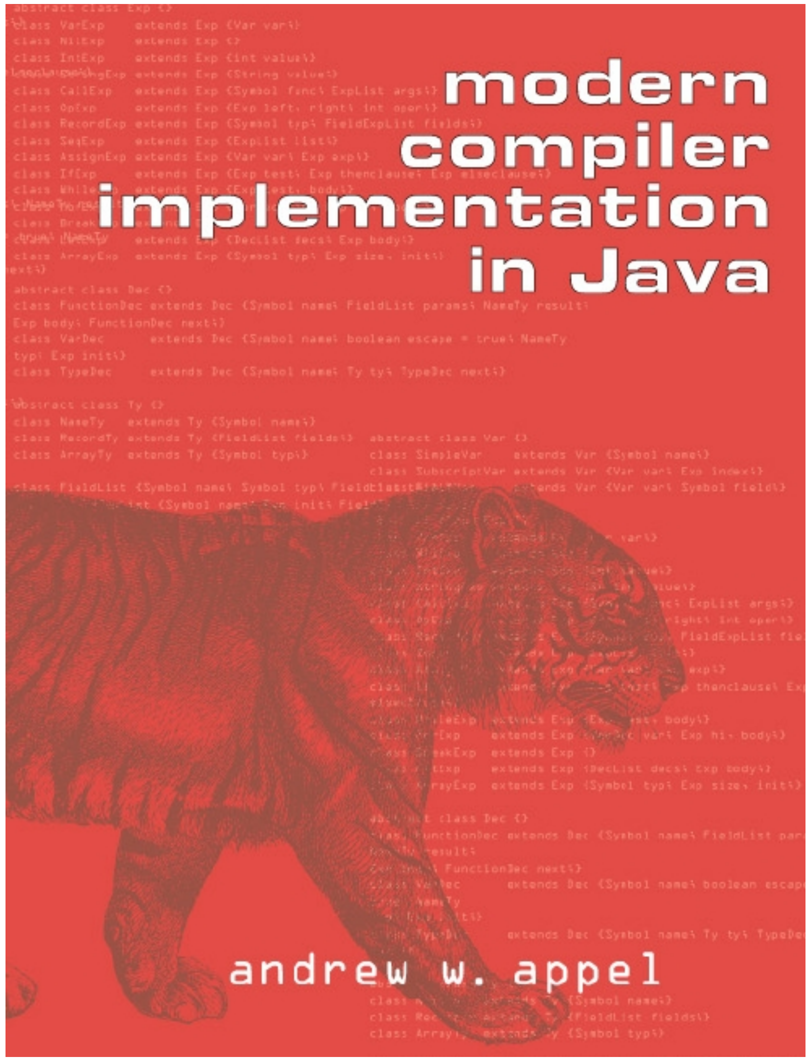
Why ML?

Why ML



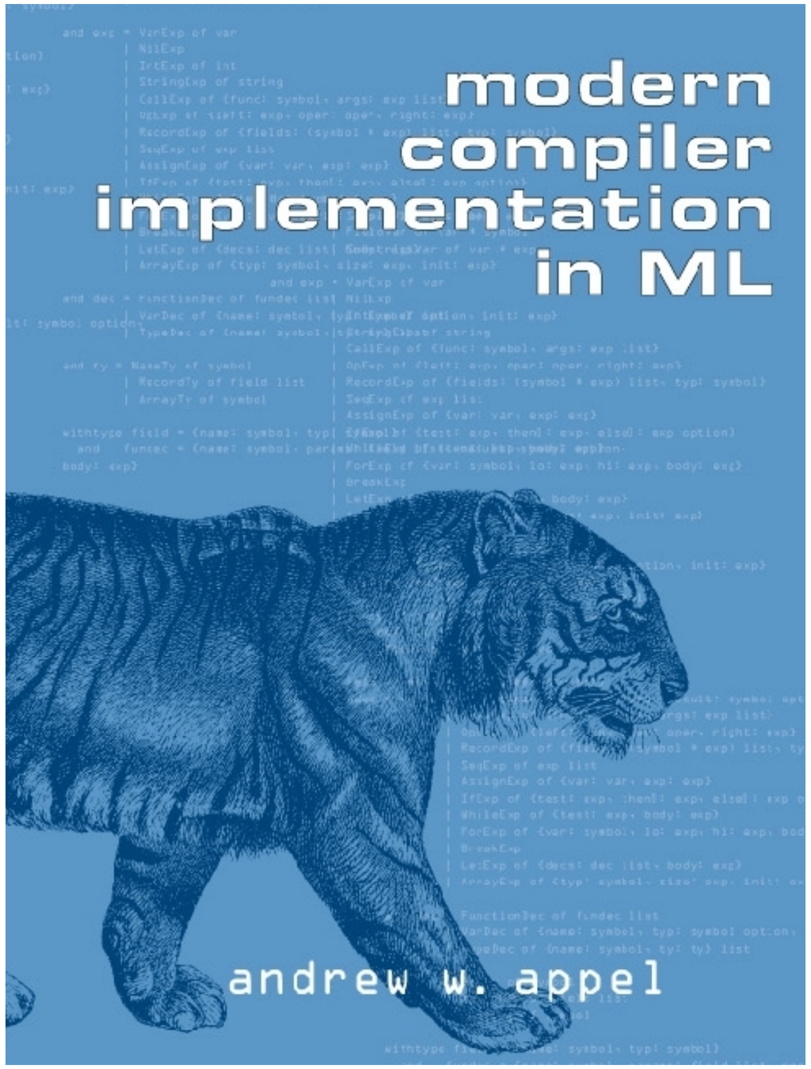
modern compiler implementation in C

andrew w. appel



modern compiler implementation in Java

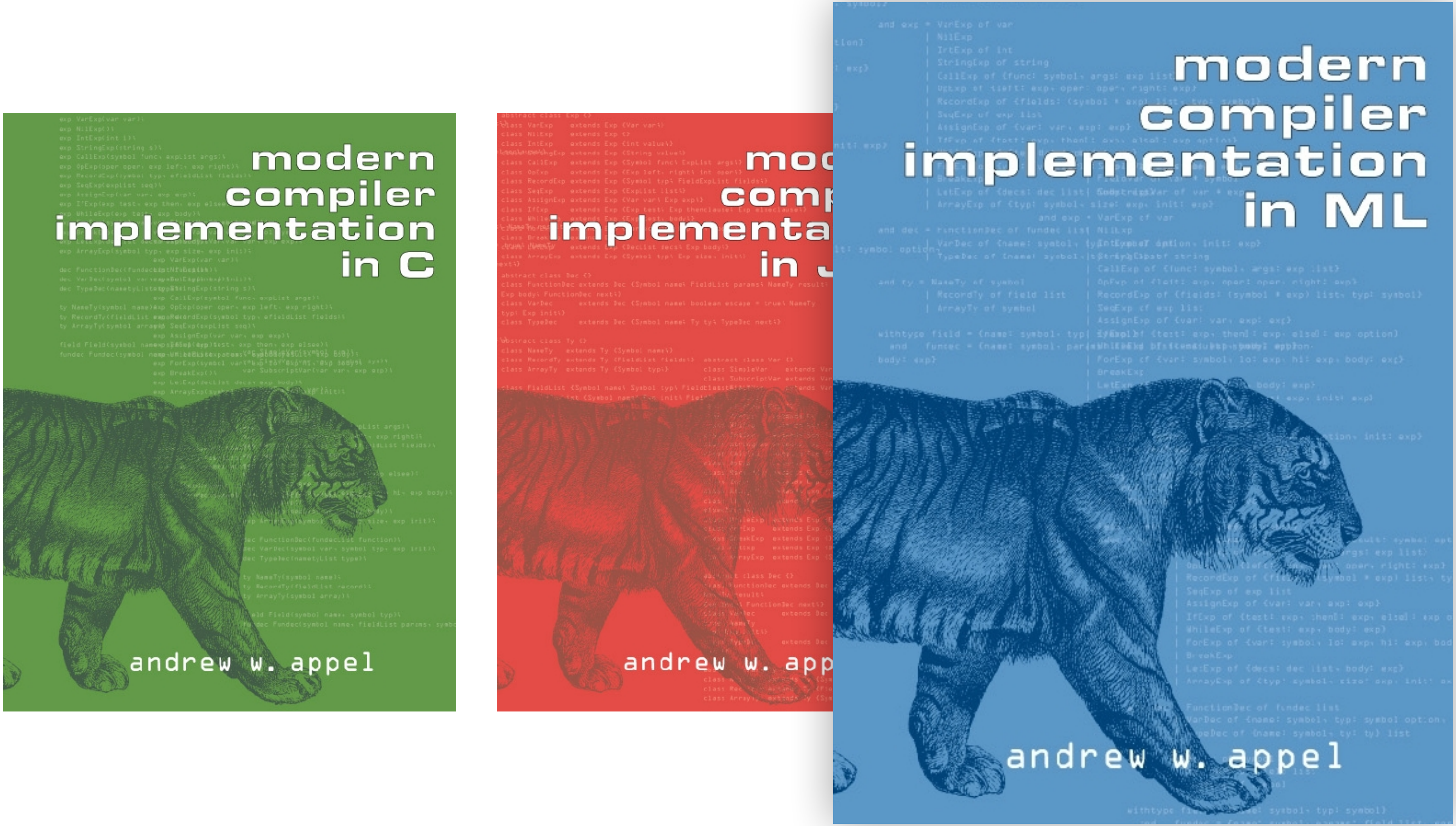
andrew w. appel



modern compiler implementation in ML

andrew w. appel

Why ML


modern compiler implementation in C
andrew w. appel
modern compiler implementation in J
andrew w. appel
modern compiler implementation in ML
andrew w. appel

A Taste of Standard ML

```
datatype    option =
datatype 'a option =
datatype 'a option =
  NONE
```

```
datatype 'a option =
  NONE
| SOME of 'a
```

```
datatype 'a option =
  NONE
| SOME of 'a

fun map
```

```
datatype 'a option =
  NONE
| SOME of 'a

fun map f
```

```
datatype 'a option =
  NONE
| SOME of 'a

fun map f option =
datatype 'a option =
  NONE
| SOME of 'a

fun map f option =
  case option of
```

```
datatype 'a option =
  NONE
| SOME of 'a

fun map f option =
  case option of
    NONE => NONE
```

```
datatype 'a option =
  NONE
| SOME of 'a

fun map f option =
  case option of
    NONE => NONE
  | SOME a => SOME (f a)
```

```
$ sml
-
```

```
$ sml
- datatype 'a option =
…   NONE
… | SOME of 'a;
```

```
$ sml
- datatype 'a option =
…   NONE
… | SOME of 'a;
datatype 'a option = NONE | SOME of 'a
-
```

```
$ sml
- datatype 'a option =
…   NONE
… | SOME of 'a;
datatype 'a option = NONE | SOME of 'a
-
- fun map f option =
…   case option of
…     NONE => NONE
…   | SOME a => SOME (f a);
```

```
$ sml
- datatype 'a option =
…   NONE
… | SOME of 'a;
datatype 'a option = NONE | SOME of 'a
-
- fun map f option =
…   case option of
…     NONE => NONE
…   | SOME a => SOME (f a);
val map = fn : ('a -> 'b) -> 'a option -> 'b option
-
```

```
$ sml
- datatype 'a option =
…   NONE
… | SOME of 'a;
datatype 'a option = NONE | SOME of 'a
-
- fun map f option =
…   case option of
…     NONE => NONE
…   | SOME a => SOME (f a);
val map = fn : ('a -> 'b) -> 'a option -> 'b option
-
- val a = SOME 1;
```

```
$ sml
- datatype 'a option =
…   NONE
… | SOME of 'a;
datatype 'a option = NONE | SOME of 'a
-
- fun map f option =
…   case option of
…     NONE => NONE
…   | SOME a => SOME (f a);
val map = fn : ('a -> 'b) -> 'a option -> 'b option
-
- val a = SOME 1;
val a = SOME 1 : int option
-
```

```
$ sml
- datatype 'a option =
…   NONE
… | SOME of 'a;
datatype 'a option = NONE | SOME of 'a
-
- fun map f option =
…   case option of
…     NONE => NONE
…   | SOME a => SOME (f a);
val map = fn : ('a -> 'b) -> 'a option -> 'b option
-
- val a = SOME 1;
val a = SOME 1 : int option
-
- map (fn a => a + 1) a;
```

```
$ sml
- datatype 'a option =
…   NONE
… | SOME of 'a;
datatype 'a option = NONE | SOME of 'a
-
- fun map f option =
…   case option of
…     NONE => NONE
…   | SOME a => SOME (f a);
val map = fn : ('a -> 'b) -> 'a option -> 'b option
-
- val a = SOME 1;
val a = SOME 1 : int option
-
- map (fn a => a + 1) a;
val it = SOME 2 : int option
```

Modules

```
datatype 'a option =
  NONE
| SOME of 'a

fun map f option =
  case option of
    NONE => NONE
  | SOME a => SOME (f a)
```

```
struct
  datatype 'a option =
    NONE
  | SOME of 'a

  fun map f option =
    case option of
      NONE => NONE
    | SOME a => SOME (f a)
end
```

```
struct
  datatype 'a option =
    NONE
  | SOME of 'a

  fun map f option =
    case option of
      NONE => NONE
    | SOME a => SOME (f a)
end
```

```
structure Option =
  struct
    datatype 'a option =
      NONE
    | SOME of 'a

    fun map f option =
      case option of
        NONE => NONE
      | SOME a => SOME (f a)
  end
```

```
structure Option =
  struct
    datatype 'a option =
      NONE
    | SOME of 'a

    fun map f option =
      case option of
        NONE => NONE
      | SOME a => SOME (f a)
  end
```

```
$ sml
- use "option.sml";
-
```

```
$ sml
- use "option.sml";
-
- val a = Option.SOME 1;
```

```
$ sml
- use "option.sml";
-
- val a = Option.SOME 1;
val a = SOME 1 : int Option.option
-
```

```
$ sml
- use "option.sml";
-
- val a = Option.SOME 1;
val a = SOME 1 : int Option.option
-
- Option.map (fn a => a + 1) a;
```

```
$ sml
- use "option.sml";
-
- val a = Option.SOME 1;
val a = SOME 1 : int Option.option
-
- Option.map (fn a => a + 1) a;
val it = SOME 2 : int Option.option
```

Functors

```
structure IntListSet =
  struct









  end
```

```
structure IntListSet =
  struct
    val empty = []

  end
```

```
structure IntListSet =
  struct
    val empty = []

    fun add set elem =







  end
```

```
structure IntListSet =
  struct
    val empty = []

    fun add set elem =
      case set of






  end
```

```
structure IntListSet =
  struct
    val empty = []

    fun add set elem =
      case set of
        [] => [elem]





  end
```

```
structure IntListSet =
  struct
    val empty = []

    fun add set elem =
      case set of
        [] => [elem]
      | head :: tail =>




  end
```

```
structure IntListSet =
  struct
    val empty = []

    fun add set elem =
      case set of
        [] => [elem]
      | head :: tail =>
          case Int.compare (head, elem) of



  end
```

```
datatype order = LESS | EQUAL | GREATER
```

```
structure IntListSet =
  struct
    val empty = []

    fun add set elem =
      case set of
        [] => [elem]
      | head :: tail =>
          case Int.compare (head, elem) of
            LESS =>
          | EQUAL =>
          | GREATER =>
  end
```

```
structure IntListSet =
  struct
    val empty = []

    fun add set elem =
      case set of
        [] => [elem]
      | head :: tail =>
          case Int.compare (head, elem) of
            LESS => head :: (add tail elem)
          | EQUAL =>
          | GREATER =>
  end
```

```
structure IntListSet =
  struct
    val empty = []

    fun add set elem =
      case set of
        [] => [elem]
      | head :: tail =>
          case Int.compare (head, elem) of
            LESS => head :: (add tail elem)
          | EQUAL => set
          | GREATER =>
  end
```

```
structure IntListSet =
  struct
    val empty = []

    fun add set elem =
      case set of
        [] => [elem]
      | head :: tail =>
          case Int.compare (head, elem) of
            LESS => head :: (add tail elem)
          | EQUAL => set
          | GREATER => elem :: set
  end
```

```
structure StringListSet =
  struct
    val empty = []

    fun add set elem =
      case set of
        [] => [elem]
      | head :: tail =>
          case String.compare (head, elem) of
            LESS => head :: (add tail elem)
          | EQUAL => set
          | GREATER => elem :: set
  end
```

```
structure StringListSet =
  struct
    val empty = []

    fun add set elem =
      case set of
        [] => [elem]
      | head :: tail =>
          case String.compare (head, elem) of
            LESS => head :: (add tail elem)
          | EQUAL => set
          | GREATER => elem :: set
  end
```

```
functor ListSet(          ) =
  struct
    val empty = []

    fun add set elem =
      case set of
        [] => [elem]
      | head :: tail =>
          case     .compare (head, elem) of
            LESS => head :: (add tail elem)
          | EQUAL => set
          | GREATER => elem :: set
  end
```

```
functor ListSet(          ) =
  struct
    val empty = []

    fun add set elem =
      case set of
        [] => [elem]
      | head :: tail =>
          case     .compare (head, elem) of
            LESS => head :: (add tail elem)
          | EQUAL => set
          | GREATER => elem :: set
  end
```

In Standard ML, a **functor** is a **module-level function** that takes a module as argument and produces a module as a result.

Note: There's no relationship between an SML functor and the `Functor` type-class as defined by the cats or scalaz libraries.

```
functor ListSet(          ) =
  struct
    val empty = []

    fun add set elem =
      case set of
        [] => [elem]
      | head :: tail =>
          case     .compare (head, elem) of
            LESS => head :: (add tail elem)
          | EQUAL => set
          | GREATER => elem :: set
  end
```

```
functor ListSet(Elem      ) =
  struct
    val empty = []

    fun add set elem =
      case set of
        [] => [elem]
      | head :: tail =>
          case Elem.compare (head, elem) of
            LESS => head :: (add tail elem)
          | EQUAL => set
          | GREATER => elem :: set
  end
```

Signatures

```
functor ListSet(Elem      ) =
  struct
    val empty = []

    fun add set elem =
      case set of
        [] => [elem]
      | head :: tail =>
          case Elem.compare (head, elem) of
            LESS => head :: (add tail elem)
          | EQUAL => set
          | GREATER => elem :: set
  end
```

```
functor ListSet(Elem : ORD) =
  struct
    val empty = []

    fun add set elem =
      case set of
        [] => [elem]
      | head :: tail =>
          case Elem.compare (head, elem) of
            LESS => head :: (add tail elem)
          | EQUAL => set
          | GREATER => elem :: set
  end
```

```
functor ListSet(Elem : ORD) =
  struct
    val empty = []

    fun add set elem =
      case set of
        [] => [elem]
      | head :: tail =>
          case Elem.compare (head, elem) of
            LESS => head :: (add tail elem)
          | EQUAL => set
          | GREATER => elem :: set
  end
```

```
functor ListSet(Elem : ORD) =
  struct
    val empty = []

    fun add set elem =
      case set of
        [] => [elem]
      | head :: tail =>
          case Elem.compare (head, elem) of
            LESS => head :: (add tail elem)
          | EQUAL => set
          | GREATER => elem :: set
  end
```

```
val compare : t * t -> order
```

```
type t
val compare : t * t -> order
```

```
sig
  type t
  val compare : t * t -> order
end
```

```
signature ORD =
  sig
    type t
    val compare : t * t -> order
  end
```

A **signature** can be seen as the type of a module.

It specifies the **types** and **values** that a module must define.

```
functor ListSet(Elem : ORD) =
  struct
    val empty = []

    fun add set elem =
      case set of
        [] => [elem]
      | head :: tail =>
          case Elem.compare (head, elem) of
            LESS => head :: (add tail elem)
          | EQUAL => set
          | GREATER => elem :: set
  end
```

```
functor ListSet(Elem : ORD) : SET =
 struct
  val empty = []

  fun add set elem =
   case set of
    [] => [elem]
   | head :: tail =>
     case Elem.compare (head, elem) of
      LESS => head :: (add tail elem)
     | EQUAL => set
     | GREATER => elem :: set
 end
```

```
functor ListSet(Elem : ORD) : SET =
  struct
    val empty = []

    fun add set elem =
      case set of
        [] => [elem]
      | head :: tail =>
          case Elem.compare (head, elem) of
            LESS => head :: (add tail elem)
          | EQUAL => set
          | GREATER => elem :: set
  end
```

```
signature SET =
  sig




  end
```

```
signature SET =
  sig


    val empty : t

  end
```

```
signature SET =
  sig
    type t

    val empty : t

  end
```

```
signature SET =
  sig
    type t

    val empty : t
    val add : t -> 🟨 -> t
  end
```

```
signature SET =
  sig
    type t
    type key
    val empty : t
    val add : t -> 🟨 -> t
  end
```

```
signature SET =
  sig
    type t
    type key
    val empty : t
    val add : t -> key -> t
  end
```

```
signature SET =
  sig
    type t
    type key
    val empty : t
    val add : t -> key -> t
  end
```

```
functor ListSet(Elem : ORD) : SET =
  struct
    type t = Elem.t list

    type key = Elem.t

    val empty = []

    fun add set elem =
      case set of
        [] => [elem]
      | head :: tail =>
          case Elem.compare (head, elem) of
            LESS => head :: (add tail elem)
          | EQUAL => set
          | GREATER => elem :: set
  end
```

```
functor ListSet(Elem : ORD) : SET =
  struct
    type t = Elem.t list

    type key = Elem.t

    val empty = []

    fun add set elem =
      case set of
        [] => [elem]
      | head :: tail =>
          case Elem.compare (head, elem) of
            LESS => head :: (add tail elem)
          | EQUAL => set
          | GREATER => elem :: set
  end
```

```
structure IntListSet = ListSet(IntOrd)
structure IntOrd =
  struct
    type t = Int.int
    val compare = Int.compare
  end


structure IntListSet = ListSet(IntOrd)
structure IntOrd =
  struct
    type t = Int.int
    val compare = Int.compare
  end


structure IntListSet = ListSet(IntOrd)


structure StringListSet = ListSet(struct
  type t = String.string
  val compare = String.compare
end)
```

Similarities with Scala?

Similarities

Standard ML	**Scala**

Similarities

Standard ML	**Scala**
`structure`	`object`

Similarities

Standard ML	Scala
`structure`	`object`
`signature`	`trait`

Similarities

Standard ML	Scala
`structure`	`object`
`signature`	`trait`
`functor`	`class / def`

Differences & Limitations

Limitations in SML

Limitations in SML

Q

If a functor is like a function, can we pass functors to functors, just like we can pass functions to functions?

Limitations in SML

Q

If a functor is like a function, can we pass functors to functors, just like we can pass functions to functions?

A

No. Standard ML does not have **higher-order functors**. OCaml and some other ML dialects have it, though.

Limitations in SML

Limitations in SML

Q

So we can't return functors from functors, either?

Limitations in SML

Q So we can't return functors from functors, either?

A No, we cannot in Standard ML.

Limitations in Scala

Limitations in Scala

Q

If Scala classes are the equivalent of SML functors, are they higher-order or not?

Limitations in Scala

Q If Scala classes are the equivalent of SML functors, are they higher-order or not?

A They're not. One cannot, save for reflection, pass classes as arguments to classes or produce classes from classes.

Limitations in SML

Limitations in SML

Q

In Scala, we can store objects in variables, pass them to functions or return them from functions. Does SML allow this with structures and functors?

Limitations in SML

Q

In Scala, we can store objects in variables, pass them to functions or return them from functions. Does SML allow this with structures and functors?

A

No. In Standard ML, modules are not **first-class**. Values and modules form two different, separate languages — the so-called **core** and **module** languages.

Limitations in SML

Limitations in SML

Q Why aren't modules first-class values in Standard ML?

Limitations in SML

Q Why aren't modules first-class values in Standard ML?

A Let's see...

```
trait Ord {
  type T
  def compare(a: T, b: T): Int
}
```

```
object IntOrd extends Ord {
  type T = Int
  def compare(a: T, b: T): Int = a - b
}
```

```
trait Set {
  type T
  type K

  def empty: T
  def add(set: T, key: K): T
}
```

```
class ListSet(val ord: Ord) extends Set {
  type K = ord.T
  type T = List[ord.T]

  def empty: T = List.empty
  def add(set: T, key: K): T = ???
}
```

```
object TwitterClient {



}
```

```
object TwitterClient {
  object UserSet extends ListSet(UserOrd)









}
```

```scala
object TwitterClient {
  object UserSet extends ListSet(UserOrd)

  def followers(username: String) = {






  }
}
```

```
object TwitterClient {
  object UserSet extends ListSet(UserOrd)

  def followers(username: String) = {


    val users = UserSet.empty



  }
}
```

```scala
object TwitterClient {
  object UserSet extends ListSet(UserOrd)

  def followers(username: String) = {


    val users = UserSet.empty

    // add users and return them
    users
  }
}
```

```scala
object TwitterClient {

  def followers(username: String) = {
    object UserSet extends ListSet(UserOrd)

    val users = UserSet.empty

    // add users and return them
    users
  }
}
```

```
object TwitterClient {

  def followers(username: String) = {
    val userSet = new ListSet(UserOrd)

    val users = UserSet.empty

    // add users and return them
    users
  }
}
```

```
object TwitterClient {

  def followers(username: String) = {
    val userSet = new ListSet(UserOrd)

    val users:           = UserSet.empty

    // add users and return them
    users
  }
}
```

```
object TwitterClient {

  def followers(username: String) = {
    val userSet = new ListSet(UserOrd)

    val users: userSet.T = UserSet.empty

    // add users and return them
    users
  }
}
```

```
object TwitterClient {

  def followers(username: String):

  = {
    val userSet = new ListSet(UserOrd)

    val users: userSet.T = UserSet.empty

    // add users and return them
    users
  }
}
```

```
object TwitterClient {

  def followers(username: String):
    List[userSet.ord.T] forSome { val userSet: ListSet }
  = {
    val userSet = new ListSet(UserOrd)

    val users: userSet.T = UserSet.empty

    // add users and return them
    users
  }
}
```

```
object TwitterClient {
  import scala.language.existentials

  def followers(username: String):
    List[userSet.ord.T] forSome { val userSet: ListSet }
  = {
    val userSet = new ListSet(UserOrd)

    val users: userSet.T = UserSet.empty

    // add users and return them
    users
  }
}
```

```
object TwitterClient {
  import scala.language.existentials

  def followers(username: String):
    userSet.T forSome { val userSet: Set }
  = {
    val userSet = new ListSet(UserOrd)

    val users: userSet.T = UserSet.empty

    // add users and return them
    users
  }
}
```

```
object TwitterClient {

  def followers(username: String):
    Set#T
  = {
    val userSet = new ListSet(UserOrd)

    val users: userSet.T = UserSet.empty

    // add users and return them
    users
  }
}
```

Limitations in SML

Q

Why aren't modules first-class values in Standard ML?

Limitations in SML

Q

Why aren't modules first-class values in Standard ML?

A

Having types as components of a signature seems to require the notion of dependent types if the language were to support first-class modules. Scala has path-dependent types.

Other Differences

Other Differences

- SML's modules are not (mutually) recursive, while Scala's objects and classes are.

Other Differences

- SML's modules are not (mutually) recursive, while Scala's objects and classes are.
- Because objects, traits and classes are values, they're also types in Scala.

Other Differences

- SML's modules are not (mutually) recursive, while Scala's objects and classes are.
- Because objects, traits and classes are values, they're also types in Scala.
- Objects come with a concept of `this` (open recursion), SML modules do not.

Other Differences

- SML's modules are not (mutually) recursive, while Scala's objects and classes are.

- Because objects, traits and classes are values, they're also types in Scala.

- Objects come with a concept of `this` (open recursion), SML modules do not.

- SML modules allow some sort of inheritance, but not overriding, as there's no `this`.

Dependency Injection

Dependency Injection

Dependency Injection

- Functor params look a lot like constructor injection.

Dependency Injection

- Functor params look a lot like constructor injection.
- The `Reader` monad is usually advocated by FP people for doing "functional" DI.

Dependency Injection

- Functor params look a lot like constructor injection.
- The `Reader` monad is usually advocated by FP people for doing "functional" DI.
- But `Reader` only injects values, not types.

Dependency Injection

- Functor params look a lot like constructor injection.
- The `Reader` monad is usually advocated by FP people for doing "functional" DI.
- But `Reader` only injects values, not types.
- A functor-like approach, i.e., constructor injection, is still useful.

Dependency Injection

- Functor params look a lot like constructor injection.
- The `Reader` monad is usually advocated by FP people for doing "functional" DI.
- But `Reader` only injects values, not types.
- A functor-like approach, i.e., constructor injection, is still useful.
- Scala alternative: implicit params.

Dependency Injection

- We should distinguish between:

Dependency Injection

- We should distinguish between:
  - **static dependencies**: dependencies are known at compile-time. Employ constructor injection or type-classes (coherent implicit params).

Dependency Injection

- We should distinguish between:
  - **static dependencies**: dependencies are known at compile-time. Employ constructor injection or type-classes (coherent implicit params).
  - **dynamic dependencies**: dependencies are known at runtime. Employ constructor injection, implicit params, `Reader` monad.

Scala's object system can and should be seen as a **first-class module system**.

Thank You!

Questions!