Software-Komponenten

Th. Letschert

THM

University of Applied Sciences

Algebraischer Entwurf, initial und final codierte Daten

- Datenabstraktion: OO und Funktional
- Algebraischer Entwurf: Funktionaler Entwurf
- Initiale und finale Datencodierung
- Interpreter-Muster, Ausdrucks- und Deserialisierungsproblem

Interface und Typklasse:

Äquivalente Softwaretechnische Basis-Konzepte

Interface: Kernkonzept der objektorientierten SWT:

Konzentriere dich auf die Methoden an der Schnittstelle der Objekte!
Ignoriere ihre innere (Kassen-basierte) Struktur!

Typklasse: Kernkonzept der funktionalen SWT:

Konzentriere dich auf die Funktionen, die auf den Daten (Werten) definiert sind!
Ignoriere ihre genaue (Daten-) Struktur.

OO-Datenabstraktion: Interface statt Klasse

Programmiere stets gegen ein Interface nicht gegen eine Klasse!

Basis einer algorithmischen Komponente.

```
trait MinMax {

  import Trees._
  import Player._

  trait Config {
    def score(): Int
    def successors(player: Player): Seq[Config]
    def isWinning(): Option[Player]
    def isFinal(): Boolean =
     isFull() || isWinning().isDefined
    def isFull(): Boolean
  }

  private enum MinMaxNode(val config: Config) {
    case MaxNode(override val config: Config) extends MinMaxNode(config)
    case MinNode(override val config: Config) extends MinMaxNode(config)
  }

  import MinMaxNode._

  private final case class MinMaxTree(node: MinMaxNode, successors: Seq[MinMaxTree])

  def bestNextConfigByMinMax(config: Config, player: Player): Config = { ... }

}
```

Das Interface einer Konfiguration – statt einer Klasse.

Unverändert: Der Algorithmus nutzt nur das Interface der Konfigurationen

OO-Datenabstraktion: Interface statt Klasse

Programmiere stets gegen ein Interface nicht gegen eine Klasse!

```
object MinMaxWithVectorConfig extends MinMax {

  class VectorConfig (val configVector: Vector[Byte]) extends Config {
    ...
  }

}

import MinMaxWithVectorConfig.VectorConfig
import Player._

val actConfig: VectorConfig = VectorConfig( Vector(
  1, -1, 0,
  -1,  1, 0,
  1, -1, 0))

val expected: VectorConfig = VectorConfig( Vector(
  1, -1,  1,
  -1,  1,  0,
  1, -1,  0))

val nextConfig = MinMaxWithVectorConfig.bestNextConfigByMinMax(actConfig, WhitePlayer)
```

Spezialisierung: Eine Algorithmische Komponente die mit Vektor-Konfigurationen arbeitet

Funktionale Datenabstraktion

Typklasse statt Interface

Typklassen sind das funktionale Äquivalent zu Interfaces, wenn es um Datenabstraktion geht.

Datenabstraktion: Kapselung von Implementierungs-Entscheidungen über Datenstrukturen / Datentypen (funktional verstanden)

```
trait MinMax {

  enum Player {
    case WhitePlayer
    case BlackPlayer
  }
  import Player._

  trait Config[C] {
    extension (c: C) def score(): Int
    extension (c: C) def successors(player: Player): Seq[C]
    extension (c: C) def isFinal(): Boolean
  }

  private enum MinMaxNode[A](val config: A) {
    case MaxNode(override val config: A) extends MinMaxNode(config)
    case MinNode(override val config: A) extends MinMaxNode(config)
  }

  import MinMaxNode._

  def bestNextConfig[C: Config](config: C): C = { ... }

}
```

Die Typklasse (!) einer Konfiguration.

Das Interface (!) einer Konfiguration.

```
trait Config {
    def score(): Int
    def successors(player: Player): Seq[Config]
    def isWinning(): Option[Player]
    def isFinal(): Boolean =
     isFull() || isWinning().isDefined
    def isFull(): Boolean
  }
```

Datenabstraktion – OO und Funktional

Funktionale DA Instanz mit Vektor-Implementierung

Der Algorithmus operiert jetzt direkt auf den Vektoren. Es gibt keine Hüllklasse, in deren Methoden die Vektoroperationen eingepackt sind.

```
object MinMaxWithVectorConfig extends MinMax {

  given VectorConfig : Config[Vector[Int]] with {  // Die Hüllklasse fällt weg !!

    private def threeInARow(config: Vector[Int]): Option[(Int, Int, Int)] = Seq(
      (0, 1, 2), (3, 4, 5), (6, 7, 8),
      (0, 3, 6), (1, 4, 7), (2, 5, 8),
      (0, 4, 8), (2, 4, 6)).find {
      case (i, j, k) =>
        (config(j) == config(i)) && (config(k) == config(i)) && (config(i) != 0)
      case _ => false
    }

    extension (config: Vector[Int]) def successors(player: Player): Seq[Vector[Int]] =
      if (config.isFinal()) Nil
      else
        for (i <- 0 until 9
             if config(i) == 0
             ) yield config.updated(i, if (player == Player.WhitePlayer) 1 else -1)

    extension (config: Vector[Int]) def isFull(): Boolean =
      !config.exists(token => token != 0)

    extension (config: Vector[Int]) def isFinal(): Boolean =
      config.isFull() || threeInARow(config).isDefined

    extension (config: Vector[Int]) def score(): Int = {
      threeInARow(config) match {
        case Some((i, _, _)) if config(i) == 1 => 1
        case Some((i, _, _)) if config(i) == -1 => -1
        case _ => 0
      }
    }
  }
}
```

```
val actConfig: Vector[Int] = Vector(
  1, -1, 0,
  -1, 1, 0,
  1, -1, 0)

val expected = Vector(
  1, -1, 1,
  -1, 1, 0,
  1, -1, 0)

val nextConfig =

MinMaxWithVectorConfig.bestNextConfig(actConfig)

assert(expected == nextConfig)
```

Algebraischer Entwurf

MinMax => Game AI

- Funktionales Äquivalent zum Objektorientierten Entwurf
- Nächste Stufe der Abstraktion nach der Datenabstraktion
- Gestalte die Schnittstelle einer API: also einer Kollektion von Typen und Werten

Beispiel MinMax verallgemeinert zu AI-Komponente:

```
trait Game_AI {

  enum Player {
    case WhitePlayer
    case BlackPlayer
  }

  trait Config[C] {
    extension (c: C) def score(): Int
    extension (c: C) def successors(player: Player): Seq[C]
    extension (c: C) def isFinal(): Boolean
  }

  def bestNextConfig[C: Config](config: C): C

}
```

Die Schnittstelle (API) einer AI-Komponente:

- **ein Typ**
- **eine Typklasse**
- **eine Funktion**

Die Berechnung der besten nächsten Konfiguration mit einem AI-Algorithmus ist softwaretechnisch (!) trivial.

Mit der Implementierung eines kompletten Spiels sieht das natürlich völlig anders aus.

Keine große Sache, warum so ein toller Name – Algebraischer Entwurf?

Algebraischer Entwurf und Algebras

Algebraischer Entwurf ~ Funktionaler Entwurf :

Funktionales Äquivalent zum Objektorientierten Entwurf

- **Prinzip (funktionale Domänen-Modellierung)**
  Beginne den Entwurf mit einer Algebra, d.h.:
  - **Definiere relevante Mengen von Werten als (Daten-) Typen**
  - **Definiere Funktionen auf den Typen**
  - **Definiere Gesetze die bei Anwendung der Funktionen gelten sollen**

- **Eine Algebra A(A, F) ist**
  - **eine Menge A**
  - **mit Operationen F die auf dieser Menge definiert sind**
  - **Für die bestimmte Gesetze gelten**
  - **Beispiel: natürliche Zahlen mit + und 0 und den üblichen Rechenregeln**

- **Eine mehrsortige Algebra**
  umfasst mehrere Mengen und Funktionen zwischen diesen
  Algebraischer Entwurf: meist mehreren Mengen / Datentypen

Algebraischer Entwurf und algebraische Struktur

Algebraische Struktur A(A, F): Muster von Algebren

- eine nicht festgelegte Menge A
- mit nicht festgelegten Operationen F die auf dieser Menge definiert sind
- Für die bestimmte Gesetze gelten
- Beispiel: Monoid(M, {0, •})

Damit gelten folgende grobe Entsprechungen:

Mathe	OO	Funktional
Algebra	Klasse	Typ
Algebraische Struktur	Interface	Typklasse

Programme und DSLs

DSL : Domänenspezifische Sprache

- **Externe DSL:** DSL ist eigene Sprache
- **Eingebettete DSL:** DSL ist Teilsprache einer allgemeinen „Wirts-“ Sprache

funktionales Programm / Modul*: Ausdruck dessen Wert zu berechnen ist

funktionales Programm / Modul einer bestimmten Domäne:

- **Ausdruck einer eingebetteten DSL**
- **DSL ist als Algebra definiert**

Beispiel:

```
trait Monoid[M] {
  def combine(x: M, y: M): M
  def unit: M
}

def sumList[M: Monoid](lst: List[M]): M =
  lst.foldLeft(summon[Monoid[M]].unit)( (acc, a) =>
    summon[Monoid[M]].combine(acc, a))


def s[M: Monoid] =
  sumList(
    List[M](
      summon[Monoid[M]].unit,
      summon[Monoid[M]].unit))
```

DSL / Algebraische Struktur

Programm

**Ein Modul ist eine Komponente auf der Ebene des Quellcodes.*

Programme und DSLs

Eine Programmkomponente (Modul)

- ist ein generischer Ausdruck, einer DSL
- der mit einer oder mehreren Algebren sowie sowie eventuell
- Eingabewerten

einen Ergebniswert berechnet

Beispiel:

```
given Monoid[Int] with {
  def combine(x: Int, y: Int): Int = x + y
  def unit: Int = 0
}
```

Algebra

```
def sumList[M: Monoid](lst: List[M]): M =
  lst.foldLeft(summon[Monoid[M]].unit)( (acc, a) =>
    summon[Monoid[M]].combine(acc, a))


def s[M: Monoid] =
  sumList(
    List[M](
      summon[Monoid[M]].unit,
      summon[Monoid[M]].unit))
```

Komponente / Modul: generischer Ausdruck in einer DSL

```
val res = s // 0
```

```
trait Monoid[M] {
  def combine(x: M, y: M): M
  def unit: M
}
```

DSL / Algebraische Struktur

Definiert mögliche Programme / Module

Initial und Final codierte Daten

Tagless-Final

Hinter dem sogenannten Tagless-Final Ansatz steht folgende Idee:

- Programmkomponenten / Module sind Ausdrücke einer DSL , die
- von einem Interpreter ausgewertet werden

Tagless Final: ein Interpreter-Muster

Interpreter-Muster:

- Ausdrücke (einer DSL)
- Ein Interpreter, der die Ausdrücke auswertet

Final

Das „Tagless“ ignorieren wir für’s erste und behandeln Initial und Final

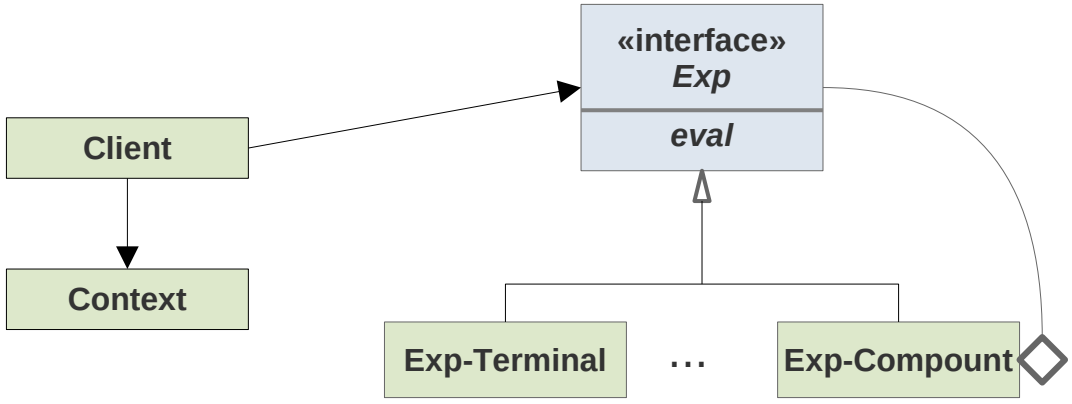
Interpreter-Muster

Interpreter: ein klassisches GoF OO-Entwurfsmuster*

Bestandteile:

- Die **Syntax** einer Ausdruckssprache (interne DSL)
- Ein **Interpreter**, der Ausdrücke dieser DSL auswerten kann
- Ein **Kontext** (optional) in dem die Ausdrücke ausgewertet werden

Syntax (ein ADT) und Interpreter werden dabei im klassischen OO-Stil modelliert:
Interface mit abstrakter Methode (Interpreter) + konkrete Klassen die das Interface implementieren



* https://de.wikipedia.org/wiki/Interpreter_(Entwurfsmuster)

Interpreter-Muster

Beispiel – OO

Arithmetische Ausdrücke mit Konstanten deren Wert vom Kontext bestimmt wird.

Syntax – ein Interface mit implementierenden Klassen (ADT mit eval-Methode) in enum-Notation

Semantik – Interpreter als eval-Methode mit Implementierungen in den konkreten Klassen

```
enum Expr(val eval: Map[String, Int] => Int) {

  case Const(number: Int)
    extends Expr(context => number)

  case Variable(name: String)
    extends Expr(context => context.getOrElse(name, 0))

  case Plus(left: Expr, right: Expr)
    extends Expr(context => left.eval(context) + right.eval(context))

  case Minus(left: Expr, right: Expr)
    extends Expr(context => left.eval(context) - right.eval(context))

}


import Expr._

val expr = Minus(Const(50), Plus(Variable("five"), Variable("three")))
val context = Map("three" -> 3, "five" -> 5)
val res = expr.eval(context) // 42
```

Interpreter-Muster

Beispiel – Funktional

Das Gleiche in funktionaler Notation

Syntax als ADT

```
enum Expr {
  case Const(number: Int)
  case Variable(name: String)
  case Plus(left: Expr, right: Expr)
  case Minus(left: Expr, right: Expr)
}
import Expr._

type Env = Map[String, Int]

def eval(expr: Expr): Env => Int = expr match {
  case Const(number) =>
    env => number
  case Variable(name: String) =>
    env => env(name)
  case Plus(left: Expr, right: Expr) =>
    env => eval(left)(env) + eval(right)(env)
  case Minus(left: Expr, right: Expr) =>
    env => eval(left)(env) - eval(right)(env)
}

val expr = Minus(Const(50), Plus(Variable("five"), Variable("three")))
val context = Map("three" -> 3, "five" -> 5)
val res = eval(expr)(context) // 42
```

Semantik als Rekursion über die Syntax

Interpreter-Muster: Ein initialer Ansatz

Syntax: Instanz einer Typklasse (algebraische Struktur)

```scala
trait F_Exp[E] {
  def const(number: Int): E
  def variable(name: String): E
  def plus(left: E, right: E): E
  def minus(left: E, right: E): E
}
```

Typklasse (Algebraische Struktur)

Expr ist Instanz von F_Expr

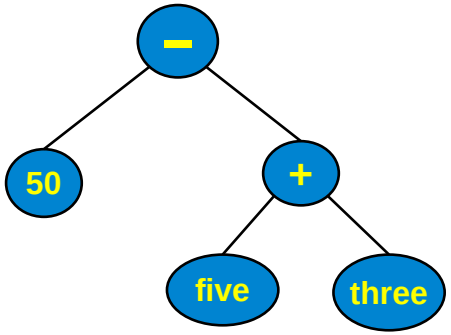
```scala
import Expr._

given F_Exp[Expr] with {
  def const(number: Int): Expr =
    Const(number)
  def variable(name: String): Expr =
    Variable(name)
  def plus(left: Expr, right: Expr): Expr =
    Plus(left, right)
  def minus(left: Expr, right: Expr): Expr =
    Minus(left, right)
}
```

Syntax als Instanz der Typklasse (Algebra der Bäume Expr ist Instanz der algebraischen Struktur F_Expr)

```scala
enum Expr {
  case Const(number: Int)
  case Variable(name: String)
  case Plus(left: Expr, right: Expr)
  case Minus(left: Expr, right: Expr)
}
```

Syntax als ADT (Algebra)

tree ist Exemplar des Typs Expr



tree ein Ausdrucksbaum

Interpreter-Muster: Ein initialer Ansatz

Syntax: Instanz einer Typklasse (algebraische Struktur)

```
trait F_Exp[E] {
  def const(number: Int): E
  def variable(name: String): E
  def plus(left: E, right: E): E
  def minus(left: E, right: E): E
}
object F_Exp {
  def apply[E:F_Exp] = summon[F_Exp[E]]
}

// ein 'generischer Ausdruck' entsprechend F_Exp
def expr[E: F_Exp] =
  F_Exp[E].minus(
    F_Exp[E].const(50),
    F_Exp[E].plus(
      F_Exp[E].variable("five"),
      F_Exp[E].variable("three")))
```

oder

```
trait F_Exp[E] {
  def const(number: Int): E
  def variable(name: String): E
  def plus(left: E, right: E): E
  def minus(left: E, right: E): E
}
object F_Exp {
  def apply[E:F_Exp] = summon[F_Exp[E]]
  def C[E:F_Exp](number: Int) =
    F_Exp[E].const(number)
  def V[E:F_Exp](name: String) =
    F_Exp[E].variable(name)
  def P[E:F_Exp](left: E, right: E) =
    F_Exp[E].plus(left, right)
  def M[E:F_Exp](left: E, right: E) =
    F_Exp[E].minus(left, right)
}

// ein 'generischer Ausdruck' entsprechend F_Exp
import F_Exp._
def expr[E: F_Exp] = M(C(50), P(V("five"), V("three")))
```

Allgemein / generisch

Interpreter-Muster: Ein initialer Ansatz

Syntax: Instanz einer Typklasse (algebraische Struktur)

```
enum Expr {
  case Const(number: Int)
  case Variable(name: String)
  case Plus(left: Expr, right: Expr)
  case Minus(left: Expr, right: Expr)
}
import Expr._


// Instanz gibt generischen Ausdrücken einen konkreten Wert
given Syntax : F_Exp[Expr] with {
  def const(number: Int): Expr =
    Const(number)
  def variable(name: String): Expr =
    Variable(name)
  def plus(left: Expr, right: Expr): Expr =
    Plus(left, right)
  def minus(left: Expr, right: Expr): Expr =
    Minus(left, right)
}

// generischer Wert interpretiert mit Syntax
// als Syntax-Baum von Typ Expr
val tree = expr // Minus(Const(50),Plus(Variable(five),Variable(three)))
```

Typ der Ausdrucksbäume

Ausdrucksbäume als Instanz von F_Exp

expr wird mit dieser Interpretation von einem generischen zu einem konkreten Ausdrucksbaum

Implizite Übergabe der Instanz der Typklasse

Interpreter-Muster: Ein initialer Ansatz

Syntax: Instanz einer **Typklasse** (algebraische Struktur)

Semantik: andere Instanz der gleichen **Typklasse**

```scala
trait F_Exp[E] {
  def const(number: Int): E
  def variable(name: String): E
  def plus(left: E, right: E): E
  def minus(left: E, right: E): E
}
```

Eine andere F_Exp passende Algebra: Die Semantik

```scala
type Env = Map[String, Int]

given Semantics : F_Exp[Env => Int] with {
  def const(number: Int): Env => Int =
    env => number
  def variable(name: String): Env => Int =
    env => env(name)
  def plus(left: Env => Int, right: Env => Int): Env => Int =
    env => left(env) + right(env)
  def minus(left: Env => Int, right: Env => Int): Env => Int =
    env => left(env) -  right(env)
}
```

```scala
// generischer Wert interpretiert
// als Berechnung vom Typ Env => Int
val evaluationInEnv = expr

val env = Map("three" -> 3, "five" -> 5)
val result = evaluationInEnv(env) // 42
```

Interpreter-Muster: Ein initialer Ansatz

Syntax: Instanz einer **Typklasse** (algebraische Struktur)

Semantik: andere Instanz der gleichen **Typklasse**

```
trait F_Exp[E] {
  def const(number: Int): E
  def variable(name: String): E
  def plus(left: E, right: E): E
  def minus(left: E, right: E): E
}
```

Typklasse


```
given Syntax : F_Exp[Expr] with {
  def const(number: Int): Expr =
    Const(number)
  def variable(name: String): Expr =
    Variable(name)
  def plus(left: Expr, right: Expr): Expr =
    Plus(left, right)
  def minus(left: Expr, right: Expr): Expr =
    Minus(left, right)
}
```

Syntax

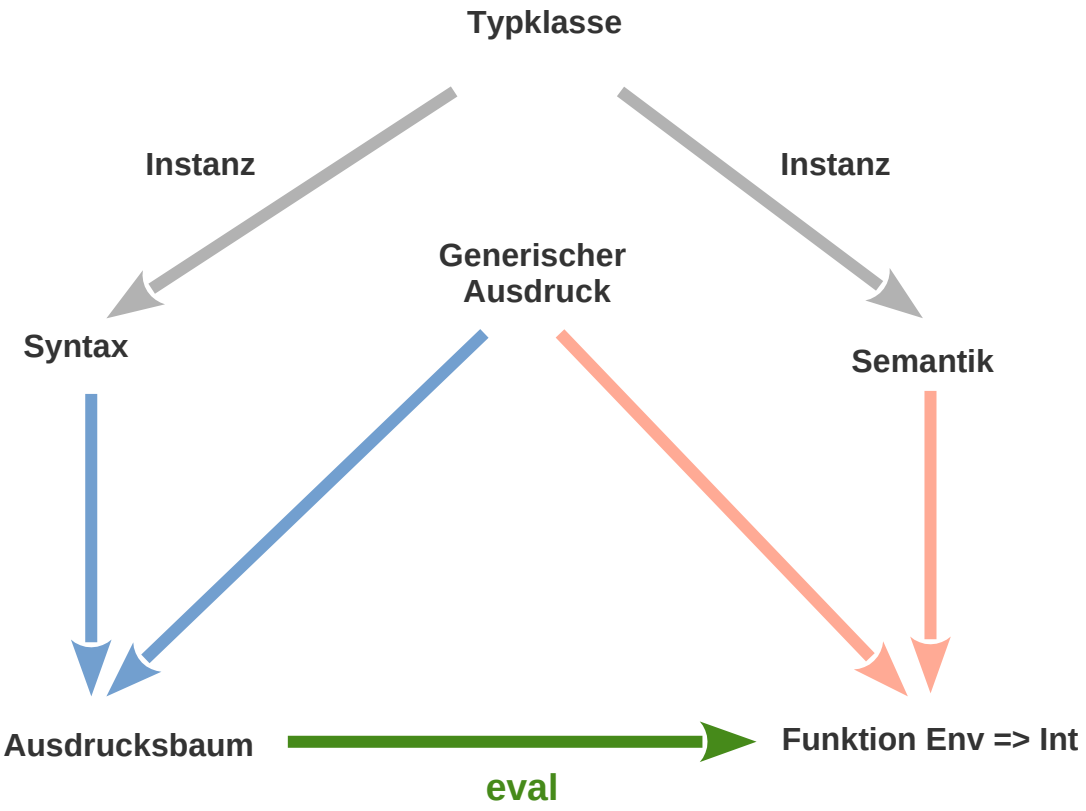
```
given Semantics : F_Exp[Env => Int] with {
  def const(number: Int): Env => Int =
    env => number
  def variable(name: String): Env => Int =
    env => env(name)
  def plus(left: Env => Int, right: Env => Int): Env => Int =
    env => left(env) + right(env)
  def minus(left: Env => Int, right: Env => Int): Env => Int =
    env => left(env) -  right(env)
}
```

Semantik

Interpreter-Muster: Ein initialer Ansatz

Syntax: Instanz einer **Typklasse** (algebraische Struktur)

Semantik: andere Instanz der gleichen **Typklasse**



Interpreter-Muster: Ein initialer Ansatz

Beobachtung: Die Definition der Syntax geht direkt aus der Typklasse hervor, reine Mechanik, keine Wahlmöglichkeit.

```scala
trait F_Exp[E] {
  def const(number: Int): E
  def variable(name: String): E
  def plus(left: E, right: E): E
  def minus(left: E, right: E): E
}
```

Typklasse

Selbstverständliche Konstruktion ohne relevante Wahlmöglichkeiten oder Alternativen

```scala
enum Expr {
  case Const(number: Int)
  case Variable(name: String)
  case Plus(left: Expr, right: Expr)
  case Minus(left: Expr, right: Expr)
}
```

```scala
given Syntax : F_Exp[Expr] with {
  def const(number: Int): Expr =
    Const(number)
  def variable(name: String): Expr =
    Variable(name)
  def plus(left: Expr, right: Expr): Expr =
    Plus(left, right)
  def minus(left: Expr, right: Expr): Expr =
    Minus(left, right)
}
```

Interpreter-Muster: Ein initialer Ansatz

Die Typklasse F definiert implizit eine F-Algebra, die Algebra der Terme T(F)

```
trait F_Exp[E] {
  def const(number: Int): E
  def variable(name: String): E
  def plus(left: E, right: E): E
  def minus(left: E, right: E): E
}
```

Algebraische Struktur F

definiert

```
enum Expr {
  case Const(number: Int)
  case Variable(name: String)
  case Plus(left: Expr, right: Expr)
  case Minus(left: Expr, right: Expr)
}
```

(Term-) Algebra T(F)

```
given Syntax : F_Exp[Expr] with {
  def const(number: Int): Expr =
    Const(number)
  def variable(name: String): Expr =
    Variable(name)
  def plus(left: Expr, right: Expr): Expr =
    Plus(left, right)
  def minus(left: Expr, right: Expr): Expr =
    Minus(left, right)
}
```

Die (Term-) Algebra ist eine F-Algebra

Interpreter-Muster: Ein initialer Ansatz

Die Term-Algebra T(F) ist nicht irgendeine F-Algebra, sie ist initial
das heißt für jede andere F-Algebra S gibt es eine Funktion eval: T => S

```
trait F_Exp[E] {
  def const(number: Int): E
  def variable(name: String): E
  def plus(left: E, right: E): E
  def minus(left: E, right: E): E
}
```

Algebraische Struktur F

definiert

Instanz von F

```
given Semantics : F_Exp[Env => Int] with {
  def const(number: Int): Env => Int =
    env => number
  def variable(name: String): Env => Int =
    env => env(name)
  def plus(left: Env => Int, right: Env => Int): Env => Int =
    env => left(env) + right(env)
  def minus(left: Env => Int, right: Env => Int): Env => Int =
    env => left(env) -  right(env)
}
```

```
enum Expr {
  case Const(number: Int)
  case Variable(name: String)
  case Plus(left: Expr, right: Expr)
  case Minus(left: Expr, right: Expr)
}
```

(Term-) Algebra T(F), die initiale F-Algebra

eval

S = Env => Int

Eine andere F-Algebra S

Existiert für jedes S: F

Interpreter-Muster: Ein initialer Ansatz

Die Term-Algebra T(F) ist nicht irgendeine F-Algebra, sie ist initial
das heißt für jede andere F-Algebra S gibt es eine Funktion eval: T => S

```
trait F_Exp[E] {
  def const(number: Int): E
  def variable(name: String): E
  def plus(left: E, right: E): E
  def minus(left: E, right: E): E
}
```

Algebraische Struktur F

definiert

Instanz von F

```
given Semntics : F_Exp[String] with {
  def const(number: Int): String =
    number.toString
  def variable(name: String): String =
    name.toString
  def plus(left: String, right: String): String =
    s"($left + $right)"
  def minus(left: String, right: String): String =
    s"($left - $right)"
}
```

```
enum Expr {
  case Const(number: Int)
  case Variable(name: String)
  case Plus(left: Expr, right: Expr)
  case Minus(left: Expr, right: Expr)
}
```

ADT der Terme: (Term-) Algebra T(F), die initiale F-Algebra

eval

String

Eine andere F-Algebra S

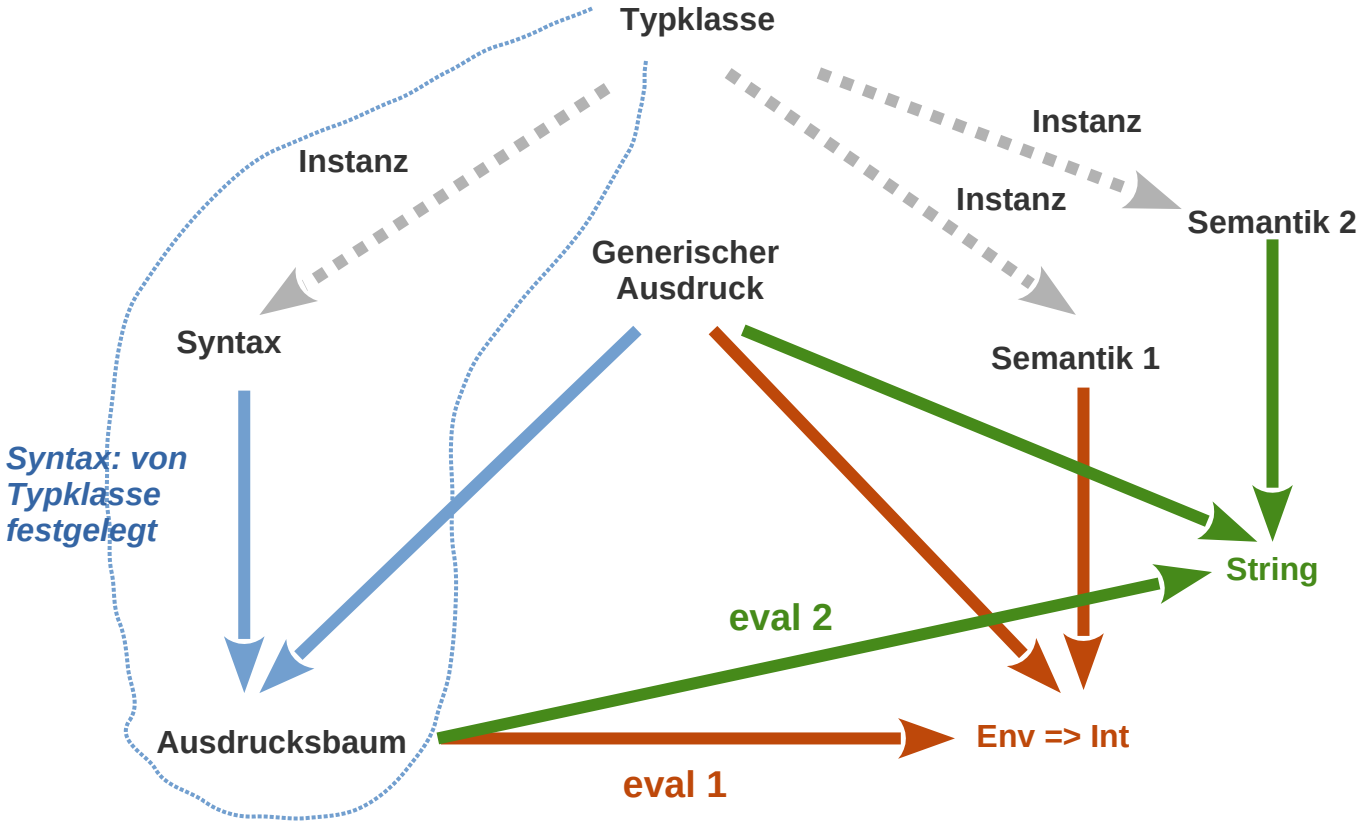
Existiert für jedes S: F

Interpreter-Muster: Ein initialer Ansatz

Syntax: Instanz einer **Typklasse** (algebraische Struktur) von Typklasse festgelegt

Semantiken: andere Instanzen der gleichen **Typklasse**

eval: Abbildung Syntax => Semantik



Interpreter-Muster: Initiale Algebra + eval

Interpreter-Muster ~ Initialer Ansatz

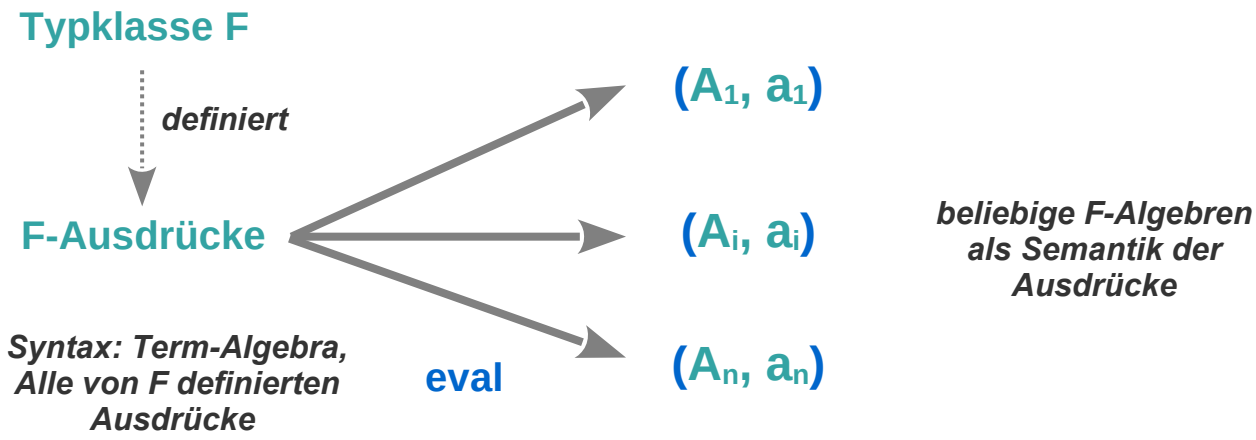
Syntax: initiales Objekt in der Kategorie der F-Algebren wobei F der Funktor ist, der mit der Syntax angegeben wird.

Ein Objekt ist initial in einer Kategorie, wenn es einen Morphismus von diesem Objekt zu jedem anderen in der Kategorie gibt.

Die Kategorie besteht aus den Algebren die die definierten Operationen haben

Der eindeutig bestimmte Morphismus ist die eval-Funktion:

Sie führt von der Syntax (initiales Objekt) zu jeder anderen Algebra mit den verlangten Operationen.



Der Initiale Ansatz

Interpreter-Muster: Initiale Algebra + eval

Signatur, definiert einen Funktor F und dieser die Kategorie der F-Algebren

Eine der F-Algebren: die initiale Algebra der F-Ausdrücke

```
trait F_Exp[E] {
  def const(number: Int): E
  def variable(name: String): E
  def plus(left: E, right: E): E
  def minus(left: E, right: E): E
}
```

F

definiert

```
given Syntax : F_Exp[Expr] with {
  def const(number: Int): Expr =
    Const(number)
  def variable(name: String): Expr =
    Variable(name)
  def plus(left: Expr, right: Expr): Expr =
    Plus(left, right)
  def minus(left: Expr, right: Expr): Expr =
    Minus(left, right)
}
```

Syntax, F-Ausdrücke: Die initiale F-Algebra

Eindeutig durch F definiert

Homomorphismus

eval

Eine andere F-Algebra

```
given Semantics : F_Exp[Env => Int] with {
  def const(number: Int): Env => Int =
    env => number
  def variable(name: String): Env => Int =
    env => env(name)
  def plus(left: Env => Int, right: Env => Int): Env => Int =
    env => left(env) + right(env)
  def minus(left: Env => Int, right: Env => Int): Env => Int =
    env => left(env) -  right(env)
}
```

Semantik: Auch eine F-Algebra

Der Initiale Ansatz

Interpreter-Muster: Initiale Algebra + eval

```
trait F_Exp[E] {
  def const(number: Int): E
  def variable(name: String): E
  def plus(left: E, right: E): E
  def minus(left: E, right: E): E
}
```

Typklasse

Definiert eindeutig

```
given Syntax : F_Exp[Expr] with {
  def const(number: Int): Expr =
    Const(number)
  def variable(name: String): Expr =
    Variable(name)
  def plus(left: Expr, right: Expr): Expr =
    Plus(left, right)
  def minus(left: Expr, right: Expr): Expr =
    Minus(left, right)
}
```

Terme / Ausdrücke:
Die initiale Algebra

eval

eval

eval

...

...

```
given ContextToIntAlgebra : F_Exp[Env => Int] with {
  def const(number: Int): Env => Int =
    env => number
  def variable(name: String): Env => Int =
    env => env(name)
  def plus(left: Env => Int, right: Env => Int): Env => Int =
    env => left(env) + right(env)
  def minus(left: Env => Int, right: Env => Int): Env => Int =
    env => left(env) -  right(env)
}
```

Interpretation der Terme / Ausdrücke durch ~~einen Homomorphismus~~ eine rekursiv über die Struktur der Ausdrücke definierte Funktion

Werte der Terme / Ausdrücke:
Andere Algebren

Der Final Ansatz

Finaler Ansatz

Beobachtung : Ausdrücke

- als Datenstruktur (ADT / „Baum“) sind nicht zwingend notwendig
- Ausdrücke können als Wert-erzeugende Funktionen definiert werden.

```
trait Expr {
  def eval(context: Map[String, Int]): Int
}

object Expr {
  def const(number: Int): Expr = (context: Map[String, Int]) => number

  def plus(left: Expr, right: Expr): Expr = (context: Map[String, Int]) =>
    left.eval(context) + right.eval(context)

  def minus(left: Expr, right: Expr): Expr = (context: Map[String, Int]) =>
    left.eval(context) - right.eval(context)

  def variable(name: String): Expr = (context: Map[String, Int]) => context.getOrElse(name, 0)
}

import Expr._
val expr = minus(const(50), plus(variable("five"), variable("three")))
val ctxt = Map("three" -> 3, "five" -> 5)
val res = expr.eval(ctxt) // 42
```

Keine „Syntax“ als Datenstruktur.

Ausdrücke sind Funktionen, die Werte erzeugen.

Der Final Ansatz

Finaler Ansatz

Beobachtung : Ausdrücke

- als Datenstruktur sind nicht zwingend notwendig
- Ausdrücke können als Wert-erzeugende Funktionen definiert werden.

Das ist nicht generisch, andere Interpretationen erfordern neue Definitionen

```
trait Expr {
  def eval(): String
}

object Expr {
  def const(number: Int): Expr = () => number.toString
  def plus(left: Expr, right: Expr): Expr = () =>
    s"(${left.eval()} + ${right.eval()})"
  def minus(left: Expr, right: Expr): Expr = () =>
    s"(${left.eval()} - ${right.eval()})"
  def variable(name: String): Expr = () => name
}

import Expr._
val expr = minus(const(50), plus(variable("five"), variable("three")))
val res = expr.eval() // (50 - (five + three))
```

Der Final Ansatz

Finaler Ansatz

Beobachtung : Ausdrücke

- als Datenstruktur sind nicht zwingend notwendig
- Ausdrücke können als Wert-erzeugende Funktionen definiert werden.

Das kann auch zu generischen Ausdrücken verallgemeinert werden

- deren Interpretation offen bleibt
- Ausdrücke als Typklasse ~ Syntax der Ausdrücke

```
trait Expr[A] {
  def const(number: Int): A
  def variable(name: String): A
  def plus(left: A, right: A): A
  def minus(left: A, right: A): A
}
```

Ausdrücke als Typklasse.

Die Interpretation der Ausdrücke ist jetzt offen: Die Typvariable A steht für irgendeinen Wertebereich der Ausdrücke.

Damit wird wieder reine Syntax definiert!

Der Final Ansatz

Finaler Ansatz

Ausdrücke

- Typklasse ~ **Syntax** der Ausdrücke
- Instanz der Typklase ~ **Semantik** der Ausdrücke

Syntax

```
trait Expr[A] {
  def const(number: Int): A
  def variable(name: String): A
  def plus(left: A, right: A): A
  def minus(left: A, right: A): A
}

object Expr {
  def apply[A](using exprRep: Expr[A]) = exprRep
}
```

Ein Ausdruck

```
def expr[A: Expr]: A =
  Expr[A].minus(
    Expr[A].const(50),
    Expr[A].plus(
      Expr[A].variable("five"),
      Expr[A].variable("three")))
```

Der Final Ansatz

Finaler Ansatz

Ausdrücke

- Typklasse ~ Syntax der Ausdrücke
- Instanz der Typklasse ~ Semantik der Ausdrücke

Die Semantik wird durch durch einen Interpreter definiert

Ein Interpreter wird als Instanz der Typklasse definiert:

```
type Context = Map[String, Int]

given Expr[Context => Int] with {
  def const(number: Int): Context => Int =
    context => number
  def variable(name: String): Context => Int =
    context => context.getOrElse(name, 0)
  def plus(left: Context => Int, right: Context => Int): Context => Int =
    context => left(context) + right(context)
  def minus(left: Context => Int, right: Context => Int): Context => Int =
      context => left(context) - right(context)
}
```

Semantik

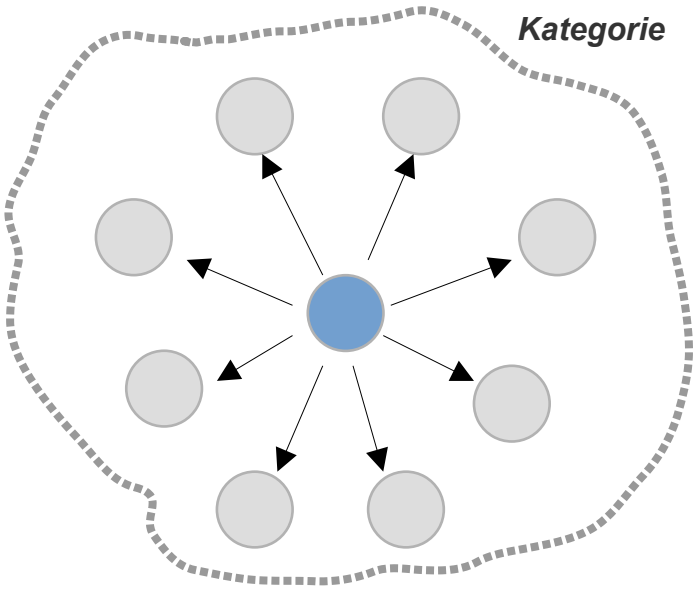
```
val context: Context = Map("three" -> 3, "five" -> 5)
val evalInContext = expr // implizite Übergabe der Instanz der Typklasse
val res = evalInContext(context) // 42
```

Ein Auswertung des Ausdrucks mit der Interpretation die die Semantik vorgibt

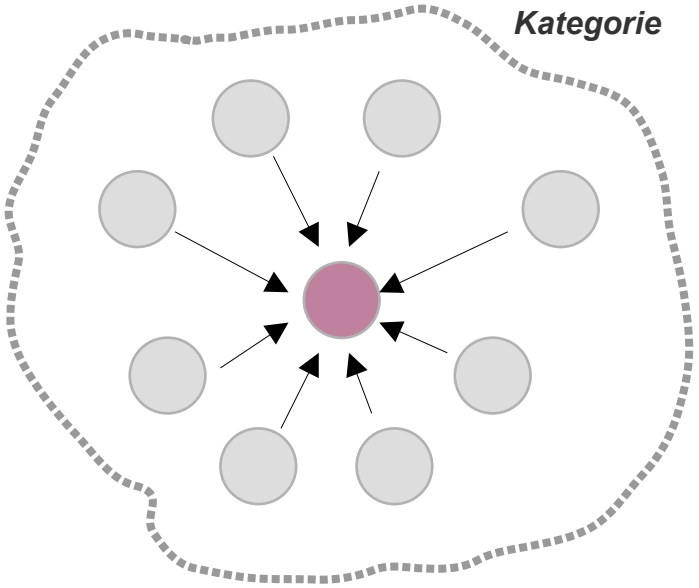
Finaler Ansatz

Initiales vs Finales (terminales) Objekt in einer Kategorie

- **initial**: Ein „Pfeil“ **zu** allen anderen
- **final**: Ein „Pfeil“ **von** jedem anderen (wird oft auch **terminal** genannt)



initial



final

Der Final Ansatz

H. PETER GUMM: STATE BASED SYSTEMS ARE COALGEBRAs

https://www.mathematik.uni-marburg.de/~gumm/Papers/Cubo.pdf

Finaler Ansatz

Initiales vs Finales Objekt in einer Kategorie

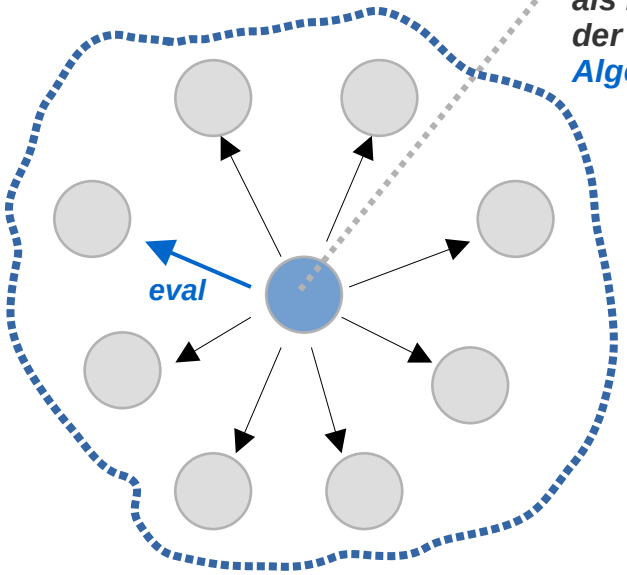
- **Initial ~ Ausdrücke ls ADT**
- **Final ~ Ausdrücke als Typklasse**

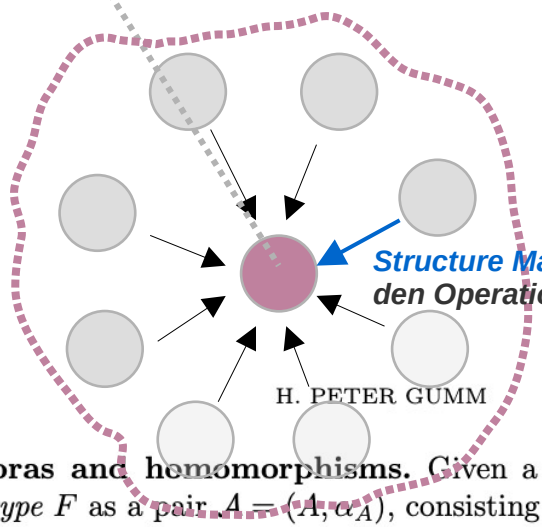
*Hier: Nur **vages Winken mit Begriffen**. Eine mathematisch korrekte Behandlung ist Thema der Theoretischen Informatik / Kategorientheorie.*

```
trait Expr[A] {
  def const(number: Int): A
  def variable(name: String): A
  def plus(left: A, right: A): A
  def minus(left: A, right: A): A
}
```

Definiert Syntax / ADT als initiales Objekt in der Kategorie der Expr-Algebren

Definiert finales Objekt in der Kategorie der Expr-Coalgebren





Structure Map: Wie wurde das Objekt mit den Operationen konstruiert

2.2. **Coalgebras and homomorphisms.** Given a type functor F, we define a *coalgebra of type* F as a pair $\mathcal{A} = (A, \alpha_A)$, consisting of a set A and a map

$$\alpha_A : A \to F(A).$$

We refer to A as the *base set* and to α_A as the *structure map* of $\mathcal{A}$.

Das Ausdrucksproblem

Finde eine Möglichkeit um sowohl die Struktur, als auch die Auswertung von Ausdrücken in modularer Art erweitern zu können.

Kein esoterisches Problem, da im modernen funktionalen Verständnis alle Programmkomponenten Ausdrücke sind.

Das Ausdrucksproblem

Ein bekanntes* Thema der Software-Technik / Programmiersprachen

Erweitere einfache Ausdrücke

- **um eine neue Variante der Struktur und / oder**
- **um einen neue Auswertungsfunktion**

Ist das auf einfache und modulare Art möglich?

Bekanntlich können

- **in einer OO-Sprache neue Strukturvarianten**
- **In einer funktionalen Sprache neue Auswertungsvarianten**

einfach und modular hinzugefügt werden

Beispiel: Erweitere das Ausdrucksbeispiel

- **um eine Multiplikation und / oder**
- **um eine *toString* Auswertung**

*** https://en.wikipedia.org/wiki/Expression_problem**

Das Ausdrucksproblem – OO

OO: modulare Erweiterung der Struktur

```
sealed abstract class Expr {
  val eval: Map[String, Int] => Int
}

final class Const(number: Int) extends Expr {
  override val eval = context => number
}

final class Variable(name: String) extends Expr {
  override val eval = context => context.getOrElse(name, 0)
}

final class Plus(left: Expr, right: Expr) extends Expr {
  override val eval = context => left.eval(context) + right.eval(context)
}

final class Minus(left: Expr, right: Expr) extends Expr {
  override val eval = context => left.eval(context) - right.eval(context)
}
```

```
// hinzugefügt
final class Mult(left: Expr, right: Expr) extends Expr {
  override val eval = context => left.eval(context) * right.eval(context)
}
```

Scala-3 Enums können nicht erweitert werden, darum etwas umständlicher als oben ohne Enum.

Einfache und modulare Erweiterung Bestehender Code muss nicht angefasst werden.

Das Ausdrucksproblem – OO

OO: un-modulare Erweiterung der Auswertung

```
sealed abstract class Expr {
  val eval: Map[String, Int] => Int
  val evalStr: String
}

final class Const(number: Int) extends Expr {
  override val eval = context => number
  override val evalStr: String = number.toString
}

final class Variable(name: String) extends Expr {
  override val eval = context => context.getOrElse(name, 0)
  override val evalStr: String = name.toString
}

final class Plus(left: Expr, right: Expr) extends Expr {
  override val eval = context => left.eval(context) + right.eval(context)
  override val evalStr: String = s"(${left.evalStr} + ${right.evalStr})"
}

final class Minus(left: Expr, right: Expr) extends Expr {
  override val eval = context => left.eval(context) - right.eval(context)
  override val evalStr: String = s"(${left.evalStr} - ${right.evalStr})"
}
```

Un-modulare Erweiterung der Auswertung: Bestehender Code muss erweitert werden.

Das Ausdrucksproblem – λ

Funktional: modulare Erweiterung der Auswertung

```
enum Expr {
  case Const(number: Int)
  case Variable(name: String)
  case Plus(left: Expr, right: Expr)
  case Minus(left: Expr, right: Expr)
}
import Expr._

type Env = Map[String, Int]

def eval(expr: Expr): Env => Int = expr match {
  case Const(number) =>
    env => number
  case Variable(name: String) =>
    env => env(name)
  case Plus(left: Expr, right: Expr) =>
    env => eval(left)(env) + eval(right)(env)
  case Minus(left: Expr, right: Expr) =>
    env => eval(left)(env) - eval(right)(env)
}
```

```
// hinzugefügt
def evalStr(expr: Expr): String = expr match {
  case Const(number) =>
    number.toString
  case Variable(name: String) =>
    name.toString
  case Plus(left: Expr, right: Expr) =>
    s"(${evalStr(left)} + ${evalStr(right)})"
  case Minus(left: Expr, right: Expr) =>
    s"(${evalStr(left)} + ${evalStr(right)})"
}
```

Einfache und modulare Erweiterung
Bestehender Code muss nicht angefasst werden.

Das Ausdrucksproblem – λ

Funktional: un-modulare Erweiterung der Struktur

```
enum Expr {
  case Const(number: Int)
  case Variable(name: String)
  case Plus(left: Expr, right: Expr)
  case Minus(left: Expr, right: Expr)
  case Mult(left: Expr, right: Expr)
}
import Expr._

type Env = Map[String, Int]

def eval(expr: Expr): Env => Int = expr match {
  case Const(number) =>
    env => number
  case Variable(name: String) =>
    env => env(name)
  case Plus(left: Expr, right: Expr) =>
    env => eval(left)(env) + eval(right)(env)
  case Minus(left: Expr, right: Expr) =>
    env => eval(left)(env) - eval(right)(env)
  case Mult(left: Expr, right: Expr) =>
    env => eval(left)(env) * eval(right)(env)
}
```

Un-modulare Erweiterung der Struktur: Bestehender Code muss erweitert werden.

Das Ausdrucksproblem

Beide, der oo- und der funktionale Ansatz sind nicht optimal.

Erweiterung	OO	λ
Struktur		
Auswertung		

Das Ausdrucksproblem

Bietet der finale Ansatz eine Lösung des Ausdrucksproblems?

Ein einfacher Test kann das zeigen

Ausgangspunkt: Eine gegebene Struktur und eine Auswertung 1

```
trait Expr[A] {
  def const(number: Int): A
  def variable(name: String): A
  def plus(left: A, right: A): A
  def minus(left: A, right: A): A
}

object Expr {
  def apply[A](using exprRep: Expr[A]) = exprRep
  def Const[A: Expr](number: Int) = Expr[A].const(number)
  def Variable[A:Expr](name: String): A = Expr[A].variable(name)
  def Plus[A: Expr](left: A, right: A): A = Expr[A].plus(left: A, right: A): A
  def Minus[A: Expr](left: A, right: A): A = Expr[A].minus(left: A, right: A): A
}

import Expr._
```

„Finale“ Definition der Ausdruckssyntax

Macht die Ausdrücke etwas schöner.

```
def expr[A: Expr]: A =
  Minus(
    Const(50),
    Plus(
      Variable("five"),
      Variable("three")))
```

Ein Ausdruck – uninterpretierte reine Syntax

Das Ausdrucksproblem

Ausgangspunkt: Eine gegebene Struktur und eine Auswertung 2

```
type Context = Map[String, Int]

given Expr[Context => Int] with {
  def const(number: Int): Context => Int =
    context => number
  def variable(name: String): Context => Int =
    context => context.getOrElse(name, 0)
  def plus(left: Context => Int, right: Context => Int): Context => Int =
    context => left(context) + right(context)
  def minus(left: Context => Int, right: Context => Int): Context => Int =
    context => left(context) - right(context)
}


val context: Context = Map("three" -> 3, "five" -> 5)
val evalInContext = expr // implizite Übergabe der Instanz der Typklasse
val res = evalInContext(context) // 42
```

Auswertungsfunktion

Der Final Ansatz

Das Ausdrucksproblem

Der finale Ansatz ist offensichtlich äquivalent zum funktionalen

Strukturerweiterung: un-modular

```
trait Expr[A] {
  def const(number: Int): A
  def variable(name: String): A
  def plus(left: A, right: A): A
  def minus(left: A, right: A): A
  def mult(left: A, right: A): A
}

given Expr[Context => Int] with {
  def const(number: Int): Context => Int =
    context => number
  def variable(name: String): Context => Int =
    context => context.getOrElse(name, 0)
  def plus(left: Context => Int, right: Context => Int): Context => Int =
    context => left(context) + right(context)
  def minus(left: Context => Int, right: Context => Int): Context => Int =
    context => left(context) - right(context)
  def mult(left: Context => Int, right: Context => Int): Context => Int =
    context => left(context) * right(context)
}
```

Un-modulare Erweiterung der Struktur:
Bestehender Code muss erweitert werden.

Das Ausdrucksproblem

Der finale Ansatz ist offensichtlich äquivalent zum funktionalen Auswertungserweiterung: modular

```
type Context = Map[String, Int]

given Expr[Context => Int] with {
  def const(number: Int): Context => Int =
    context => number
  def variable(name: String): Context => Int =
    context => context.getOrElse(name, 0)
  def plus(left: Context => Int, right: Context => Int): Context => Int =
    context => left(context) + right(context)
  def minus(left: Context => Int, right: Context => Int): Context => Int =
    context => left(context) - right(context)
}
```

```
given Expr[String] with {
  def const(number: Int): String = number.toString
  def variable(name: String): String = name
  def plus(left: String, right: String): String =
    s"($left + $right)"
  def minus(left: String, right: String): String =
    s"($left - $right)"
}
```

Einfache und modulare Erweiterung: Bestehender Code muss nicht angefasst werden.

```
val context: Context = Map("three" -> 3, "five" -> 5)
val evalInContext: Context => Int = expr // implizite Übergabe der ersten Instanz der Typklasse
val evalToString: String = expr // implizite Übergabe der zweiten Instanz der Typklasse
val res1 = evalInContext(context) // 42
val res2 = evalToString // (50 - (five + three))
```

Das Ausdrucksproblem

Der finale Ansatz ist offensichtlich äquivalent zum funktionalen

Hmm – Wirklich – Problematisch ist die Strukturerweiterung:

```
trait Expr[A] {
  def const(number: Int): A
  def variable(name: String): A
  def plus(left: A, right: A): A
  def minus(left: A, right: A): A
  def mult(left: A, right: A): A
}
```

Der Final Ansatz

Das Ausdrucksproblem

Der finale Ansatz ist offensichtlich äquivalent zum funktionalen

Nein – Auch die Struktur kann modular erweitert werden:

```
trait Expr[A] {
  def const(number: Int): A
  def variable(name: String): A
  def plus(left: A, right: A): A
  def minus(left: A, right: A): A
}

object Expr {
  def apply[A](using exprRep: Expr[A]) = exprRep
  def Const[A: Expr](number: Int) = Expr[A].const(number)
  def Variable[A:Expr](name: String): A = Expr[A].variable(name)
  def Plus[A: Expr](left: A, right: A): A = Expr[A].plus(left: A, right: A): A
  def Minus[A: Expr](left: A, right: A): A = Expr[A].minus(left: A, right: A): A
}
import Expr._
```

```
trait Multiplication[A] {
  def mult(left: A, right: A): A
}
import Multiplication._

object Multiplication {
  def apply[A](using multRep: Multiplication[A]) = multRep
  def Mult[A: Multiplication](left: A, right: A): A = Multiplication[A].mult(left: A, right: A): A
}
```

Einfache und modulare Erweiterung der Struktur:
Bestehender Code muss nicht angefasst werden.

Das Ausdrucksproblem

Der finale Ansatz ist offensichtlich äquivalent zum funktionalen

Nein – Auch die Struktur kann modular erweitert werden:

```
// ein Ausdruck mit Multiplikation
def expr[A: Expr: Multiplication]: A =
  Minus(
    Mult(
      Variable("five"),
      Const(10)),
    Plus(
      Variable("five"),
      Variable("three")))
```

Ein Ausdruck in der erweiterten Syntax

```
given Expr[String] with {
  def const(number: Int): String = number.toString
  def variable(name: String): String = name
  def plus(left: String, right: String): String =
    s"($left + $right)"
  def minus(left: String, right: String): String =
    s"($left - $right)"
}
```

Auswertung der Basis-Syntax

```
given Multiplication[String] with {
  def mult(left: String, right: String): String =
    s"($left * $right)"
}
```

Auswertung der erweiterten Syntax

```
val res: String = expr // ((five * 10) - (five + three))
```

Der Final Ansatz und das Ausdrucksproblem

Der finale Ansatz ist eine Möglichkeit um sowohl die Struktur, als auch die Auswertung von Ausdrücken in modularer Art erweitern zu können.

Das Deserialisierungsproblem

Serialisierung / Deserialisierung

Serialisierung

Transformation einer Struktur in eine lineare (serielle) Darstellung (Text, Bytestrom, ...)

Deserialisierung

Rekonstruktion der Struktur aus ihrer seriellen Darstellung

Serialisierung und Deserialisierung finden wenig Interesse bei Akademikern

Es sind aber interessante Thematik von enormer praktischer Bedeutung

Das Deserialisierungsproblem

JSON

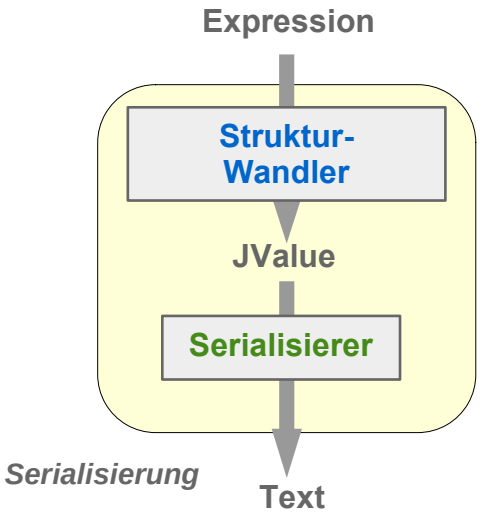
JSON*: serielle (Text-) Form beliebig strukturierter Daten

definiert als: Universelle Struktur (JSON-Value) in festgelegter Textcodierung

Universelle JSON-Struktur (JSON Value) in Scala beispielsweise als Enum-codierter ADT:

```
enum JValue {
  case JObject(prop:    Map[String, JValue])
  case JArray(elements: List[JValue])
  case JString(value:   String)
  case JNumber(value:   String)
  case JTrue
  case JFalse
  case JNull
}
```

Ein JSON-Wert ist ein Objekt, ein Array, ein String, eine Zahl, ein boolescher Wert oder null.



RFC 8259** definiert dazu eine textuelle Darstellung die leicht zu erzeugen und zu erkennen (parsen) ist.

Serialisierung: Anwendungsstruktur => Allgemeines Struktur => Text (Bytestrom)

Deserialisierung :

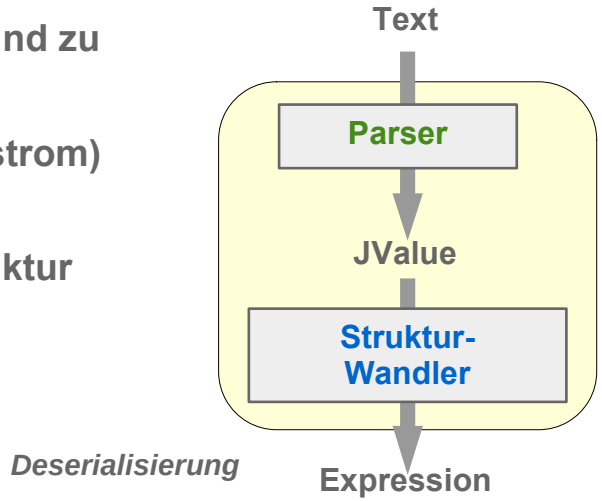
Text (Bytestrom) => Allgemeine Struktur (JValue) => Anwendungs-Struktur

Parsing: Standardkomponente (Bibliothek)

Allgemeine Struktur => Spezielle Struktur (Anwendung)

Essenz des Deserialisierungsproblems:

Strukturwandlung: Allgemeine Struktur => Spezielle Struktur



* https://de.wikipedia.org/wiki/JavaScript_Object_Notation

** https://tools.ietf.org/html/rfc8259

Das Deserialisierungsproblem

Beispiel

JSON-artige Struktur

Ausdrücke in finaler Codierung (generische Ausdruckswerte)

```
enum Tree {
  case Leaf(value: String)
  case Node(value: String, elements: List[Tree])
}
```

Allgemeine Struktur: Vereinfachte Version eines Json-Werts

```
trait Expr[A] {
  def const(number: Int): A
  def variable(name: String): A
  def plus(left: A, right: A): A
  def minus(left: A, right: A): A
}
```

Spezielle Struktur: Ausdrücke in finaler Codierung

```
def expr[A: Expr]: A =
  Minus(
    Plus(
      Variable("five"),
      Const(45)),
    Plus(
      Variable("five"),
      Variable("three")))
```

Ein („generischer“) Ausdruck

```
object Expr {
  def apply[A](using exprRep: Expr[A]) =
      exprRep
  def Const[A: Expr](number: Int) =
      Expr[A].const(number)
  def Variable[A:Expr](name: String): A =
      Expr[A].variable(name)
  def Plus[A: Expr](left: A, right: A): A =
      Expr[A].plus(left: A, right: A): A
  def Minus[A: Expr](left: A, right: A): A =
      Expr[A].minus(left: A, right: A): A
}
import Expr._
```

Zur Verschönerung der Ausdrücke

Das Deserialisierungsproblem

Serialisierung

1. Ausstattung von Tree mit toString

```
enum Tree {
  case Leaf(value: String)
  case Node(value: String, elements: List[Tree])
}
```



```
enum Tree(override val toString: String) {
  case Leaf(value: String)
    extends Tree ( toString = value )
  case Node(value: String, elements: List[Tree])
    extends Tree ( toString = s"{$value: [${elements.mkString(", ")}]}" )
}
```

Das Deserialisierungsproblem

Serialisierung

2. Tree als Instanz der Typklasse Expr

```
given ToTree as Expr[Tree] with {
  def const(number: Int): Tree =
    Node("C", List[Tree](Leaf(number.toString)))
  def variable(name: String): Tree =
    Node("V", List[Tree](Leaf(name)))
  def plus(left: Tree, right: Tree): Tree =
    Node("P", List[Tree](left, right))
  def minus(left: Tree, right: Tree): Tree =
    Node("M", List[Tree](left, right))
}
```

```
val tree: Tree = expr
```

Das Deserialisierungsproblem

Deserialisierung

Aus einem Tree-Wert soll ein generischer Ausdruck werden.

```
enum Tree {
  case Leaf(value: String)
  case Node(value: String, elements: List[Tree])
}
```

```
trait Expr[A] {
  def const(number: Int): A
  def variable(name: String): A
  def plus(left: A, right: A): A
  def minus(left: A, right: A): A
}
```

```
val tree: Tree =
  Node("minus",
    List(
      Node("plus",
      List(
        Node("var",
          List(Leaf("five"))),
        Node("const",
          List(Leaf("45"))))
      ),
      Node("plus",
        List(
          Node("var", List(Leaf("five"))),
          Node("var",List(Leaf("three")))))))
```

```
def expr[A: Expr]: A =
  Expr[A].minus(
    Expr[A].plus(
      Expr[A].variable("five"),
      Expr[A].const(45)),
    Expr[A].plus(
      Expr[A].variable("five"),
      Expr[A].variable("three")))
```

Das Deserialisierungsproblem

Deserialisierung

Aus einem Tree-Wert soll ein ~~generischer~~ konkreter Ausdruck werden.

Z.B. ein Exemplar der initialen Algebra

```
enum Tree {
  case Leaf(value: String)
  case Node(value: String, elements: List[Tree])
}
```

```
// ADT: initiale Algebra zur Typklasse Expr
enum ExprADT {
  case Const(number: Int)
  case Variable(name: String)
  case Plus(left: ExprADT, right: ExprADT)
  case Minus(left: ExprADT, right: ExprADT)
}
import ExprADT._
```

```
val tree: Tree =
  Node("minus",
    List(
      Node("plus",
      List(
        Node("var",
          List(Leaf("five"))),
        Node("const",
          List(Leaf("45"))))
      ),
      Node("plus",
        List(
          Node("var", List(Leaf("five"))),
          Node("var",List(Leaf("three")))))))
```

```
val adt =
  Minus(
    Plus(
      Variable("five"),
      Const(45)),
    Plus(
      Variable("five"),
      Variable("three")))
```

Das Deserialisierungsproblem

Deserialisierung konkret

Aus einem Tree-Wert soll ein ~~generischer~~ konkreter Ausdruck werden.

Z.B. ein Exemplar der initialen Algebra

```
def treeToADT(tree: Tree): ExprADT = tree match {
  case Node(tag, nodes) =>
    tag match {
      case "const" =>
        nodes match {
          case Leaf(str) :: Nil => Const(str.toInt)
        }
      case "var" =>
        nodes match {
          case Leaf(str) :: Nil => Variable(str)
        }
      case "plus" =>
        nodes match {
          case node0 :: node1 :: Nil =>
            Plus(treeToADT(node0), treeToADT(node1))
        }
      case "minus" =>
        nodes match {
          case node0 :: node1 :: Nil =>
            Minus(treeToADT(node0), treeToADT(node1))
        }
    }
}
```

Der Übersicht halber ohne jede Fehlerbehandlung

Das Deserialisierungsproblem

Deserialisierung konkret

ToADT **(1)**

= ToTree und dann treeToADT **(2)**

```scala
def genExpr[A: Expr]: A =
  Expr[A].minus(
    Expr[A].plus(
      Expr[A].variable("five"),
      Expr[A].const(45)),
    Expr[A].plus(
      Expr[A].variable("five"),
      Expr[A].variable("three")))
```

```scala
given ToADT : Expr[ExprADT] with {
  def const(number: Int): ExprADT =
    Const(number)
  def variable(name: String): ExprADT =
    Variable(name)
  def plus(left: ExprADT, right: ExprADT): ExprADT =
    Plus(left, right)
  def minus(left: ExprADT, right: ExprADT): ExprADT =
    Minus(left, right)
}
```

(1)

```scala
given ToTree : Expr[Tree] with {
  def const(number: Int): Tree =
    Node("const", List[Tree](Leaf(number.toString)))
  def variable(name: String): Tree =
    Node("var", List[Tree](Leaf(name)))
  def plus(left: Tree, right: Tree): Tree =
    Node("plus", List[Tree](left, right))
  def minus(left: Tree, right: Tree): Tree =
    Node("minus", List[Tree](left, right))
}
```

```scala
val adtFromGenExpr: ExprADT = genExpr
val adtFromTree: ExprADT = treeToADT(treeFromGenExpr)
```

Serialisierung

treeToADT

Deserialisie-rung

(2)

```scala
val treeFromGenExpr: Tree = genExpr
```

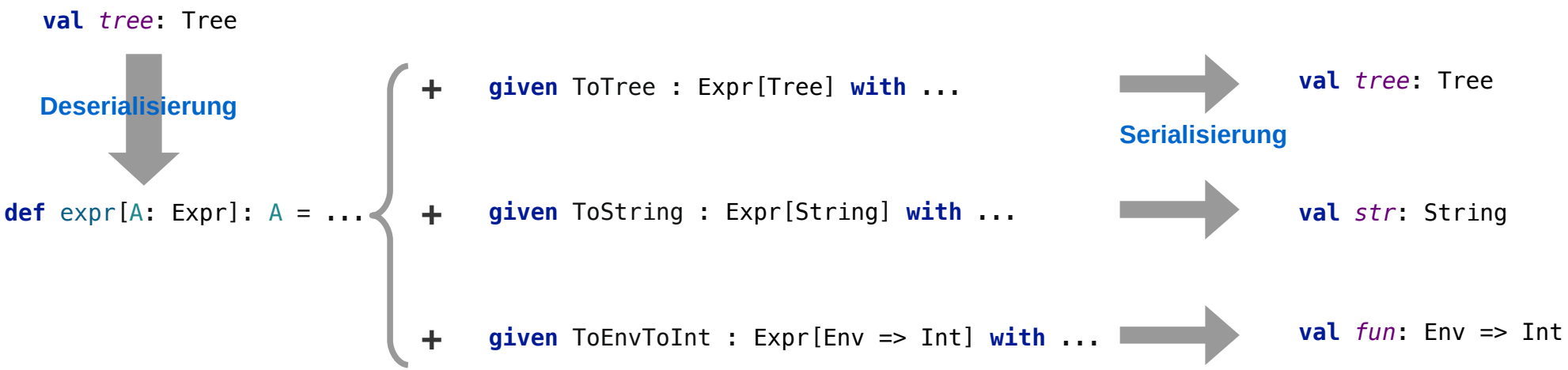
Das Deserialisierungsproblem

Deserialisierung

Aus einem Tree-Wert soll ein final codierter also generischer Ausdruck werden.

Ein generischer Ausdruck ist eine „Typ-Funktion“, die

- **einen Typ T annimmt**
- **und daraus einen Wert vom Typ Expr[T] konstruiert**



Das Deserialisierungsproblem

Deserialisierung

Rang-2 Polymorphimus

Wenn generische Werte, also Werte die einen Typ parametrisiert sind, unbeschränkt erzeugbar und verwendbar sind, dann spricht man von Rank-2 Polymorphismus.

Scala 3 unterstützt Rang-2 Polymorphismus*. Der final codierte Ausdruck, ein generischer Wert kann damit definiert werden als:

```scala
def fromTree(tree: Tree): [A] => Expr[A] => A =
  [A] => (eRep: Expr[A]) => tree match {
  case Node(tag, nodes) =>
    tag match {
      case "const" =>
        nodes match {
          case Leaf(str) :: Nil => eRep.const(str.toInt)
        }
      case "var" =>
        nodes match {
          case Leaf(str) :: Nil => eRep.variable(str)
        }
      case "plus" =>
        nodes match {
          case node0 :: node1 :: Nil =>
            eRep.plus(fromTree(node0)(eRep), fromTree(node1)(eRep))
        }
      case "minus" =>
        nodes match {
          case node0 :: node1 :: Nil =>
            eRep.minus(fromTree(node0)(eRep), fromTree(node1)(eRep))
        }
    }
}
```

```scala
val genExpr: [A] => Expr[A] => A = fromTree(tree)
```

***fromTree* erzeugt einen Wert (!), dessen Typ generisch ist.**

Das ist Rang-2 Polymorphismus!

*siehe http://dotty.epfl.ch/docs/reference/new-types/polymorphic-function-types.html

Das Deserialisierungsproblem

Deserialisierung

Finale codierte Ausdrücke können erzeugt werden!

```
val tree: Tree =
  Node("minus",
    List(
      Node("plus",
        List(
          Node("var",
            List(Leaf("five"))),
          Node("const",
            List(Leaf("45"))))),
      Node("plus",
        List(
          Node("var", List(Leaf("five"))),
          Node("var",List(Leaf("three")))))))
```

```
val genExpr: [A] => Expr[A] => A = fromTree(tree)

def deserialize[A: Expr](tree: Tree) =
  fromTree(tree)(summon[Expr[A]])
```

```
object SerializeToStr {
  given StrExpr : Expr[String] with {
    def const(number: Int): String = number.toString
    def variable(name: String): String = name
    def plus(left: String, right: String): String =
      s"($left + $right)"
    def minus(left: String, right: String): String =
      s"($left - $right)"
  }

  val res = deserialize(tree)////((five + 45) - (five + three))
}
```

Das Deserialisierungsproblem

Deserialisierung

Ohne Kontext Funktionen: keine implizite Übergabe der Typklassen-Instanz

```
val genExpr: [A] => Expr[A] => A = fromTree(tree)

def deserialize[A: Expr](tree: Tree) =
  fromTree(tree)(summon[Expr[A]])
```

Hier wird die Typklassen-Instanz als expliziter Parameter übergeben.

Die die, die das nicht mögen, können mit Kontext-Funktionen* arbeiten.

*** http://dotty.epfl.ch/docs/reference/contextual/context-functions.html**

Das Deserialisierungsproblem

Deserialisierung

Mit Kontext Funktionen: implizite Übergabe der Typklassen-Instanz

Kontext-Funktion

```
def fromTree(tree: Tree): [A] => Expr[A] ?=> A =
  [A] => (using eRep: Expr[A]) => tree match {
  case Node(tag, nodes) =>
    tag match {
      case "const" =>
        nodes match {
          case Leaf(str) :: Nil => summon[Expr[A]].const(str.toInt)
        }
      case "var" =>
        nodes match {
          case Leaf(str) :: Nil => summon[Expr[A]].variable(str)
        }
      case "plus" =>
        nodes match {
          case node0 :: node1 :: Nil =>
            eRep.plus(fromTree(node0)(using summon[Expr[A]]), fromTree(node1)(using summon[Expr[A]]))
        }
      case "minus" =>
        nodes match {
          case node0 :: node1 :: Nil =>
            eRep.plus(fromTree(node0)(using summon[Expr[A]]), fromTree(node1)(using summon[Expr[A]]))
        }
    }
}

val genExpr: [A] => Expr[A] ?=> A = fromTree(tree)
```

Das Deserialisierungsproblem

Deserialisierung

Mit Kontext Funktionen: implizite Übergabe der Typklassen-Instanz

```scala
given StrExpr : Expr[String] with {
  def const(number: Int): String = number.toString
  def variable(name: String): String = name
  def plus(left: String, right: String): String =
    s"($left + $right)"
  def minus(left: String, right: String): String =
    s"($left - $right)"
}

val res: String = genExpr[String] ////((five + 45) - (five + three))
```

Der Final Ansatz und das Deserialisierungsproblem

Texte können zu generischen (final codierten)
Ausdruckswerten deserialisiert werden.