ISA
Institut für
SoftwareArchitektur

TECHNISCHE HOCHSCHULE MITTELHESSEN

Compilerbau CS1019

Th. Letschert

TH Mittelhessen Gießen

University of Applied Sciences

Struktur und Entwicklung eines Compilers

- Konkrete und abstrakte Syntax (AST)
- Vom Interpreter zum Compiler
- Phasenstruktur eines Compilers
- Analyse und Entwurf eines Compilers

Interpreter und Compiler: Aufgaben

Interpreter:

- Syntaxanalyse: Der Parser erzeugt einen (vereinfachten) Ableitungsbaum: den AST
- Auswertung: Der AST wird ausgewertet

Compiler:

- Syntaxanalyse: Der Parser erzeugt einen AST
- Die Auswertung wird ersetzt durch
  - eine Analyse und Transformation des AST sowie
  - die Codegenerierung

Konkrete und abstrakte Syntax

Syntax

Programm- und natürlich-sprachliche Texte sind lineare Folgen von Zeichen (Strings)

Ihre primäre Struktur ist darum immer: Folge von Zeichen

In diese primäre Struktur eingebettet ist stets eine „tiefere“ Struktur:

```
Zeichen ~> Worte ~> Nominalphrase, Verbalphrase ~> Hauptsatz, Nebensatz ~> Satz
```

Der Parser hat die Aufgabe die tiefere Struktur zu erkennen und zu dokumentieren

Redundanz der Syntax

(Programm-) Texte enthalten in der Regel Bestandteile ohne „eigene Sinn“

Bestandteile, die nur dazu diesen, die Struktur des Textes erkennbar zu machen

Beispiel: Klammern

Abstrakte Syntax

Ein abstrakter Syntax Baum (*abstract syntax tree)* kurz AST

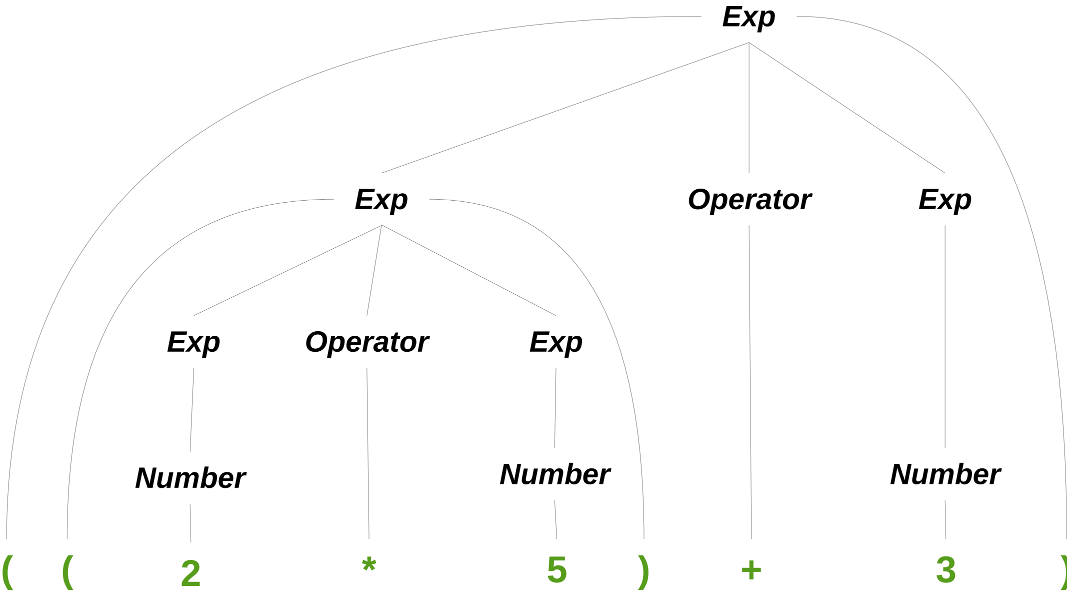
ist ein Ableitungsbaum in dem alle redundanten Elemente entfernt wurden.

Konkrete Syntax

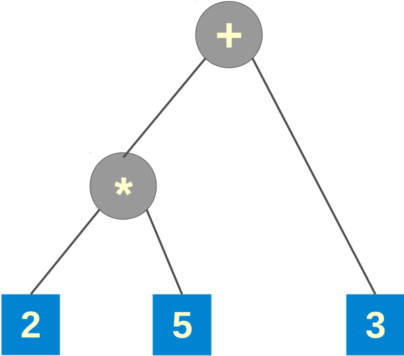
Die Syntax der die Text folgen wird zur Abgrenzung oft konkrete Syntax genannt

(Konkrete) Syntax und Abstrakte Syntax

Beispiel



Konkrete Syntax: Ableitungsbaum, enthält Elemente die nur zum Erkennen der Struktur notwendig sind



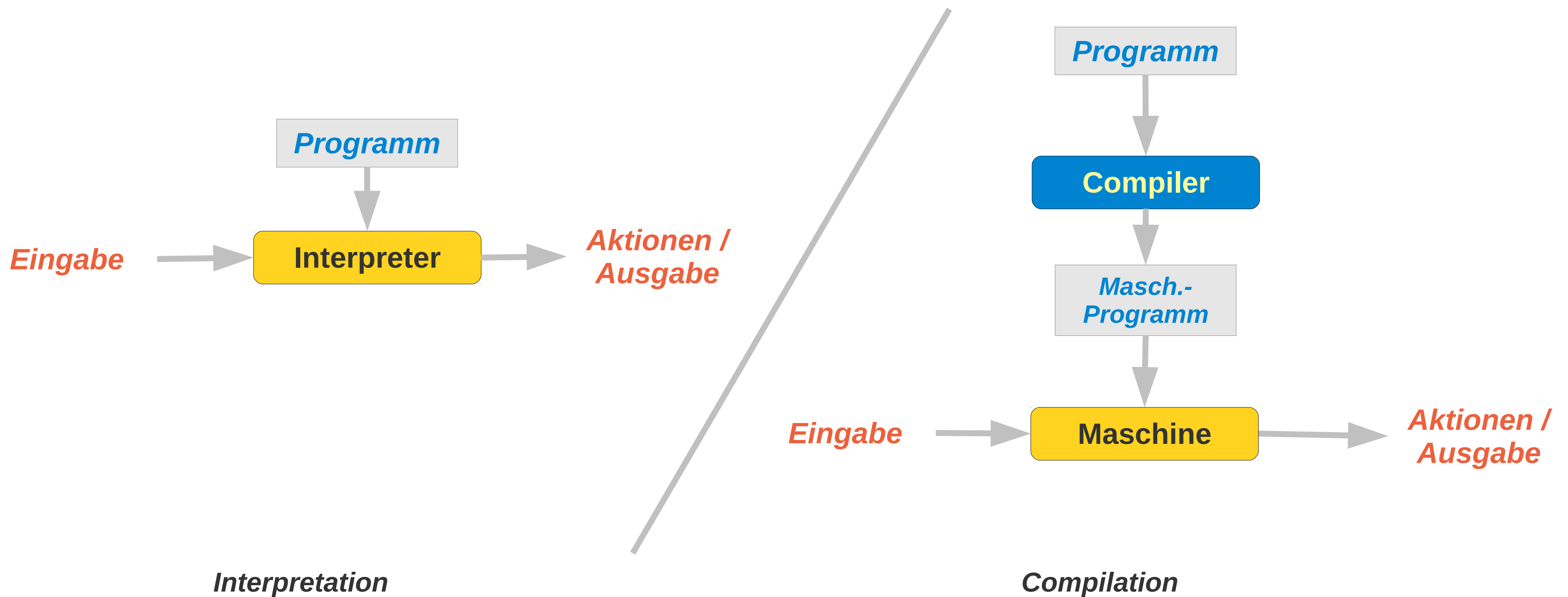
Abstrakte Syntax: Ableitungsbaum mit Abstraktion von, für die Semantik, unwesentlichen Elementen

Interpreter und Compiler

Interpreter: Realisiert ein zum Quellprogramm äquivalentes Verhalten in der Metasprache

Compiler: Erzeugt
- in der Metasprache
- ein Zielprogramm in der Zielsprache
- mit zum Quellprogramm äquivalenten Verhalten

Metasprache: Sprache in der die Verarbeiter der Quellsprache, Interpreter und Compiler, verfasst sind.

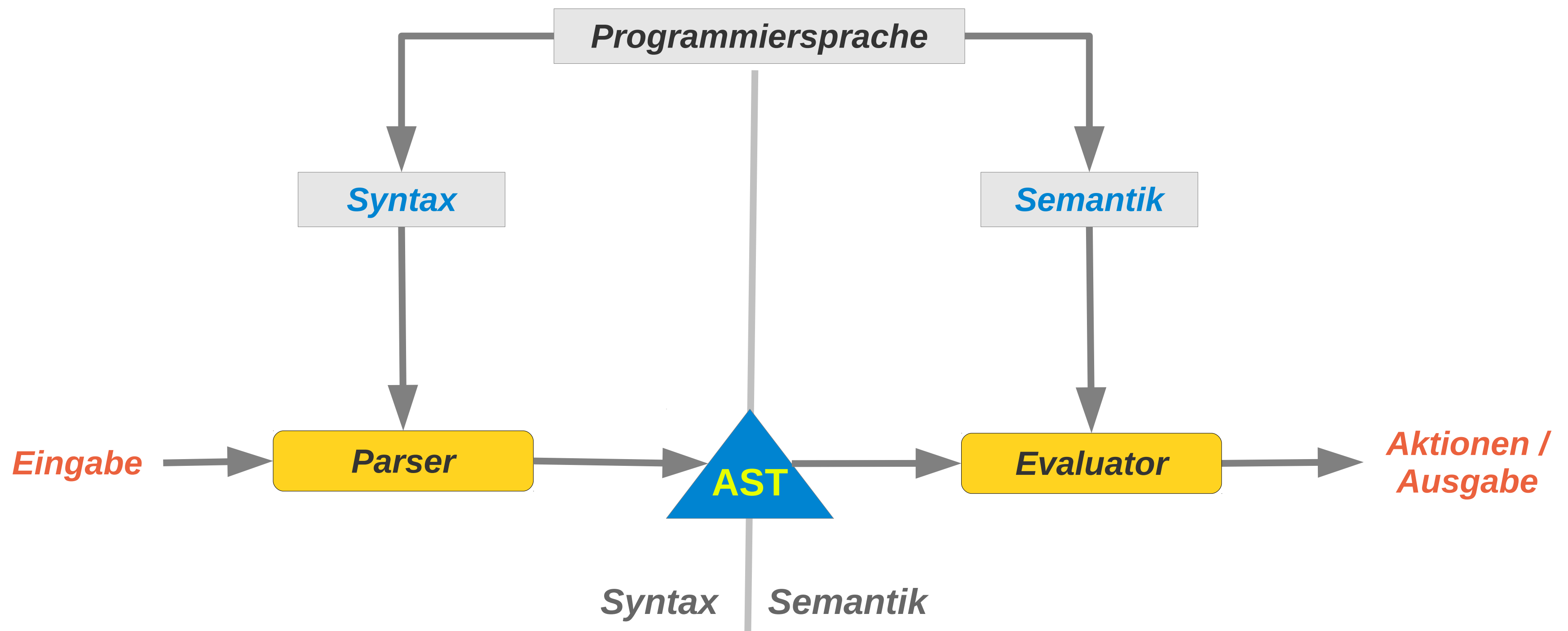


Interpreter: Definition von Syntax und Semantik

Interpreter: Realisiert das Verhalten des Quellprogramms

Interpreter: kann als **Definition** einer Programmiersprache angesehenen werden

- Die **Syntax** wird durch Grammatik definiert
- Die **Semantik** wird durch die Funktion definiert, die den abstrakten Syntaxbaum auswertet. (Das Programm bedeutet, das was der Interpreter macht, wenn er es liest.)
- AST, der abstrakte Syntaxbaum, bildet die Schnittstelle zwischen der Verarbeitung von Syntax und Semantik



Definitorischer Interpreter

Definition einer Programmiersprache

Interpreter als Sprachdefinition: Definitorischer Interpreter

Ein Interpreter der

- möglichst einfach und
- möglichst klar

definiert ist, kann als Definition einer Programmiersprache verwendet werden.

Da mit Grammatiken ein klarer und einfacher Formalismus zur Definition der konkreten Syntax zur Verfügung steht, definiert man eine Sprache über Syntax, abstrakte Syntax und Evaluator

Sprachdefinition: Syntax, AST, Evaluator

Eine Sprachdefinition besteht aus folgenden Komponenten:

- Definition der konkreten Syntax in Form einer Grammatik
- Definition der abstrakten Syntax (AST) in Form eines algebraischen Datentyps (ADT) *
- Definition der Semantik in Form einer Auswertungsfunktion (Evaluator)

* ADT, Thema von *Algorithmen und Datenstrukturen*, siehe etwa: https://homepages.thm.de/~hg51/Veranstaltungen/A_D/Folien/ad-03.pdf

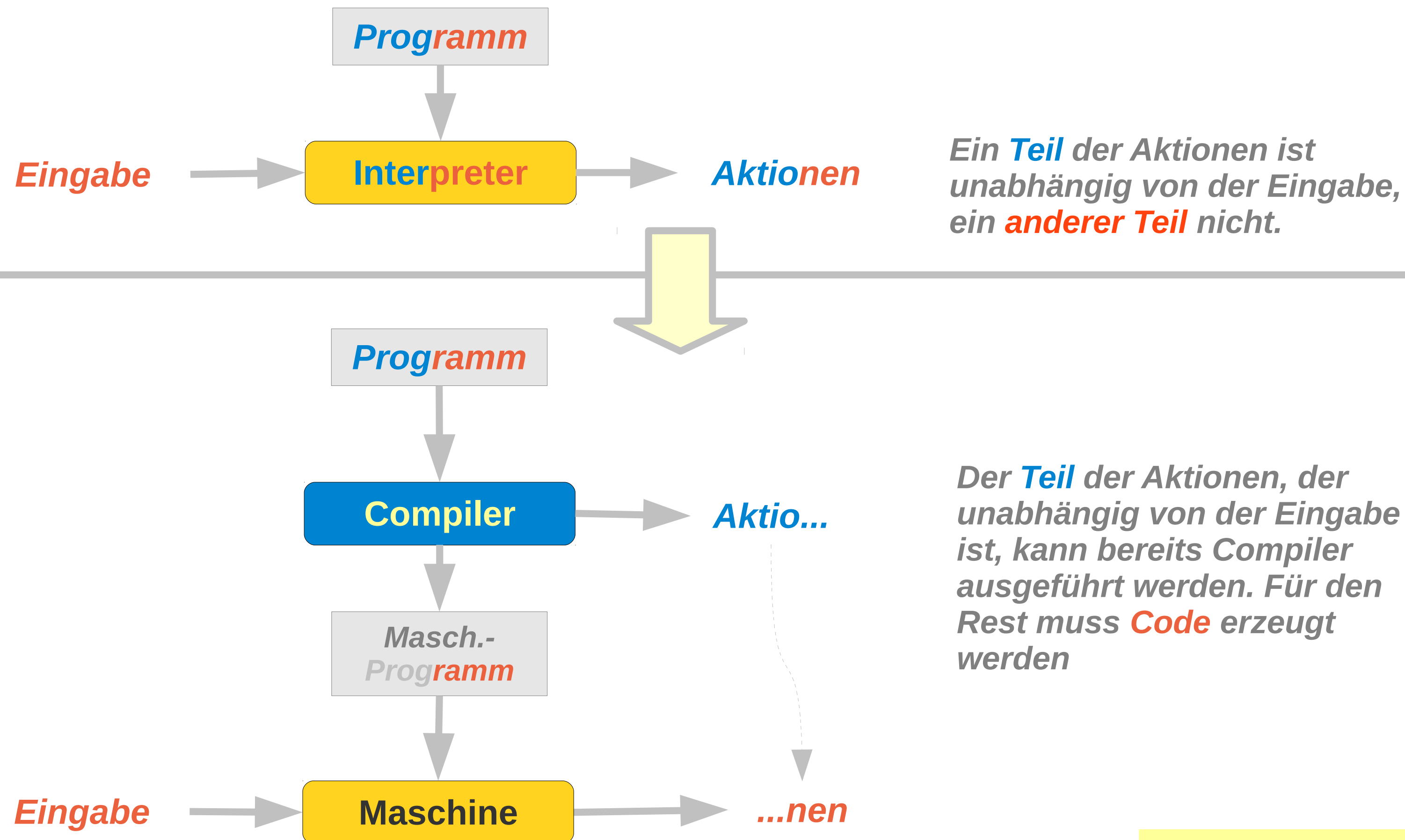
Interpreter und Compiler

Die zentrale Fragestellung des Compiler-Bauers ist: „Was ist statisch und was ist dynamisch?“

Vom Interpreter zum Compiler

Ein Compiler ist ein Interpreter, der

- Möglichst viel „interpretiert“
- und für das, was er nicht „interpretieren“ kann, Maschinencode erzeugt



Ein Teil der Aktionen ist unabhängig von der Eingabe, ein anderer Teil nicht.

Der Teil der Aktionen, der unabhängig von der Eingabe ist, kann bereits Compiler ausgeführt werden. Für den Rest muss Code erzeugt werden

… oder:„Was kann zur Compile-Zeit, was muss zur Laufzeit gemacht werden?

Interpreter und Compiler

Vom Interpreter zum Compiler

Statik und Dynamik

- **Statisch**: alles was (Eingabe-unabhängig ist und darum) zur Übersetzungszeit bearbeitet wird (oder werden kann), wird statisch genannt
- **Dynamisch**: alles was (Eingabe-abhängig ist und darum) zur Laufzeitzeit bearbeitet wird (oder werden muss), wird dynamisch genannt

Vom Interpreter zum Compiler

- **Konzeption**: Statische- und dynamische Berechnungen trennen
- Statische Berechnungen zur Compile-Zeit ausführen
- **Codegenerierung**: Für dynamische Berechnungen Code generieren.

Vom Interpreter zum Compiler

Beispiel-Sprache

Vollständig geklammerte arithmetische Ausdrücke mit READ-Anweisungen

z.B.:

```
((2 + R) * 5)
```

Bei der Auswertung werden Leseanweisungen (`'R'`) ausgeführt.

Die Auswertung wird damit eingabeabhängig.

Vom Interpreter zum Compiler

Beispiel-Sprache: Abstrakte Syntax *(vergl. Foliensatz 1)*

```
object AST {
  sealed abstract class ExpTree
  case class  Number(v: Int) extends ExpTree
  case object ReadExp extends ExpTree
  case class  Operation(exp1: ExpTree, op: Char, exp2: ExpTree) extends ExpTree
}
```

Abstrakte Syntax der Ausdrücke mit eingestreuten READ-Anweisungen

Vom Interpreter zum Compiler

Beispiel-Sprache: Konkrete Syntax *(vergl. Foliensatz 1)*

```
object Parser {

  def parse(text: String): ExpTree = {
    var pos: Int = 0

    def parseExp: ExpTree = text(pos) match {
      case '1' => parseNumber
      ...
      case '0' => parseNumber
      case '(' => parseOperation
      case 'R' => parseRead
      case _ => throw new IllegalArgumentException
    }

    def parseOperation: Operation = {
      ...
    }

    def parseNumber: Number = text(pos) match {
      ...
    }

    def parseRead() : ExpTree = text(pos) match {
      case 'R' => { pos = pos+1; ReadExp }
      case _ => throw new IllegalArgumentException
    }

    parseExp
  }

}
```

Ein 'R' symbolisiert die Eingabeaufforderung

Vom Interpreter zum Compiler

Beispiel-Sprache: Semantik als Evaluator *(vergl. Foliensatz 1)*

```
object Evaluator {

  def eval(tree: ExpTree): Int = tree match {
      case Number(x) => x
      case ReadExp => scala.io.StdIn.readInt()
      case Operation(e1, op, e2) =>
        op match {
          case '+' => eval(e1) + eval(e2)
          case '-' => eval(e1) - eval(e2)
          case '*' => eval(e1) * eval(e2)
          case '/' => eval(e1) / eval(e2)
          case _ => throw new IllegalArgumentException
        }
  }

}
```

Evaluator.eval verarbeitet

- *das Programm als **AST** und*
- *die **Eingabe***

Vom Interpreter zum Compiler

Statische und dynamische Berechnungen

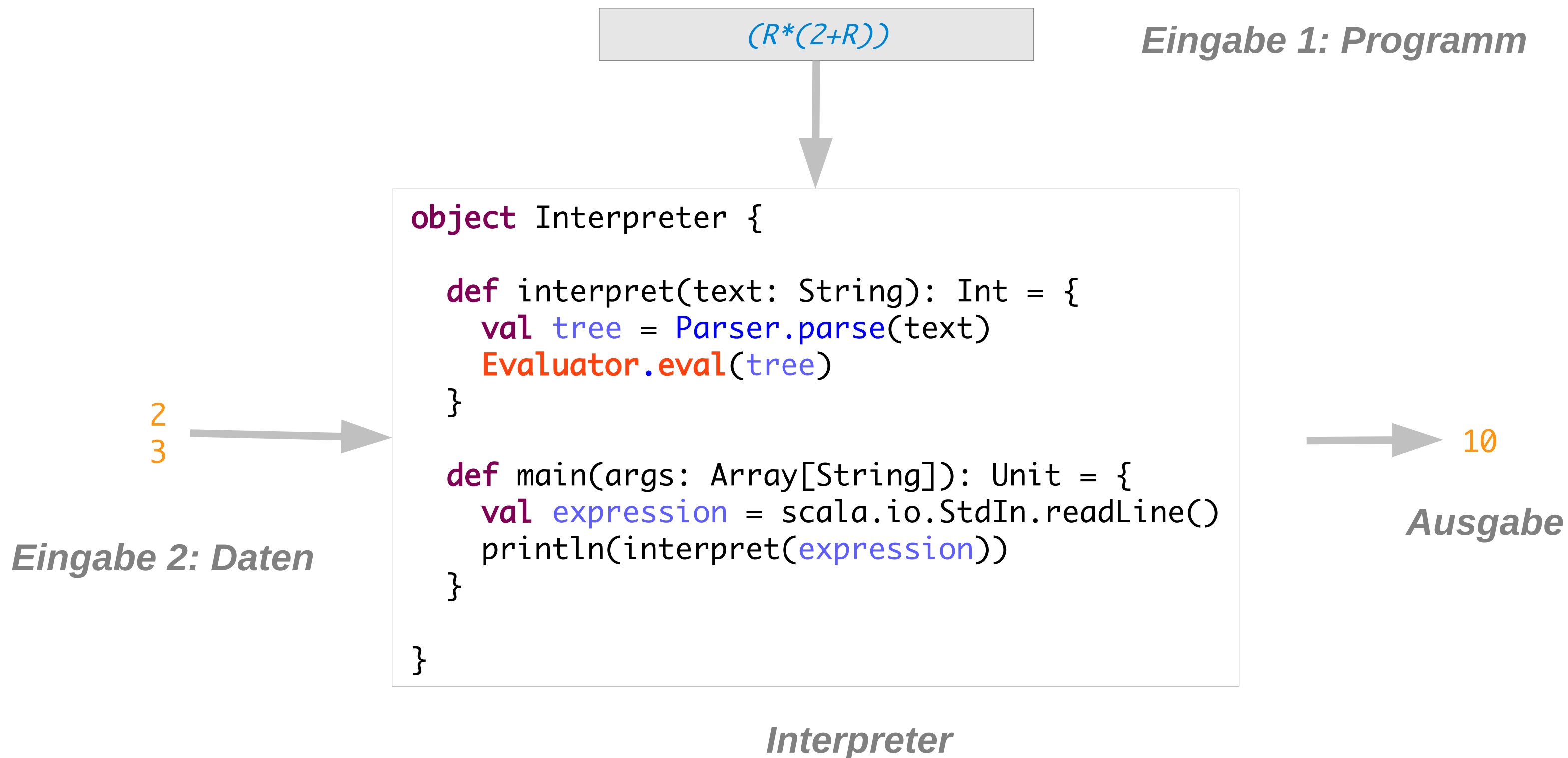
Alles ist statisch, ausser den Einlese-Operationen und den von ihnen abhängigen Berechnungen.

Statisch ist also

- **Parser.parse** **komplett**
- **Evaluator.eval** **teilweise**

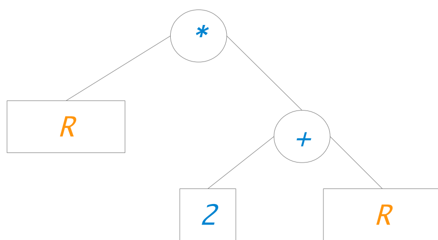
Vom Interpreter zum Compiler

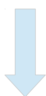
Statische und dynamische Berechnungen



Der Parser agiert komplett statisch und muss im Folgenden nicht weiter betrachtet werden.

Vom Interpreter zum Compiler Beispiel Ausdruckssprache





```
def eval(tree: ExpTree): Int = tree match {
    case Number(x) => x
    case ReadExp => scala.io.StdIn.readInt()
    case Operation(e1, op, e2) =>
      op match {
        case '+' => eval(e1) + eval(e2)
        case '-' => eval(e1) - eval(e2)
        case '*' => eval(e1) * eval(e2)
        case '/' => eval(e1) / eval(e2)
        case _ => throw new IllegalArgumentException
      }
}
```

rein statische Aktion: AST durchlaufen, Knoten analysieren

rein dynamische Aktion: Wert einlesen

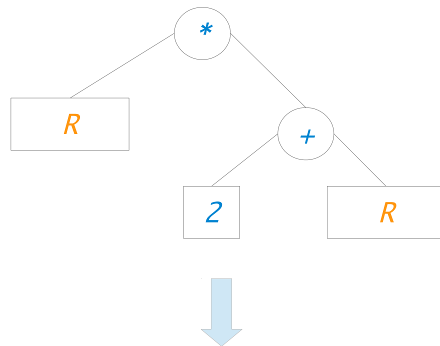
statisch und dynamisch gemischt. Manche Operanden sind statisch, andere dynamisch

2
3 → 10

Partielle Interpretation

Partielle Interpretation

Die Funktion eval wird mit einem Teil der Eingabe (dem AST) ausgeführt und liefert eine Funktion, die den Rest (die Eingabedaten) verarbeitet




```
(x: Int, y:Int) => x*(2+y)
```

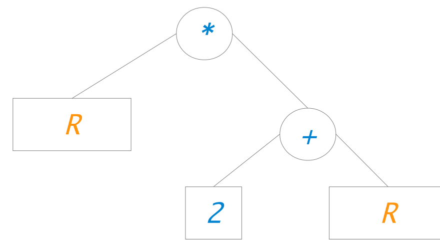
Die partielle Auswertung liefert eine Funktion die die restlichen, dynamischen Berechnungen ausführt.

Vom Interpreter zum Compiler

Compilation

Compiler

Die Funktion wird mit einem Teil der Eingabe (den Ausdruck) ausgeführt
und liefert eine Code, mit dessen Ausführung der Rest (die Daten) verarbeitet wird



```
Push 2
Read
Add
Read
Mult
```

Die Compilation liefert Code der die restlichen, dynamischen Berechnungen ausführt.

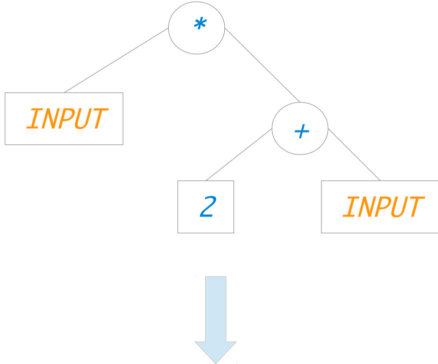
2
3

Stackmaschine

10

Vom Interpreter zum Compiler

Analyse Beispiel Ausdruckssprache



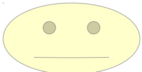

```
def eval(tree: ExpTree): Int = tree match {
    case Number(x) => x
    case ReadExp => scala.io.StdIn.readInt()
    case Operation(e1, op, e2) =>
      op match {
        case '+' => eval(e1) + eval(e2)
        case '-' => eval(e1) - eval(e2)
        case '*' => eval(e1) * eval(e2)
        case '/' => eval(e1) / eval(e2)
        case _ => throw new IllegalArgumentException
      }
}
```

Post-Order-Traversierung des AST (statisch)

Code zum Einlesen eines Int-Werts (dynamisch)

Gemischte „Rechnungen“:

Zahl , Zahl ~> Zahl
Zahl , Code ~> Code
Code , Code ~> Code
Code , Code ~> Code



eval kann Code oder eine Zahl liefern: Implementierung?

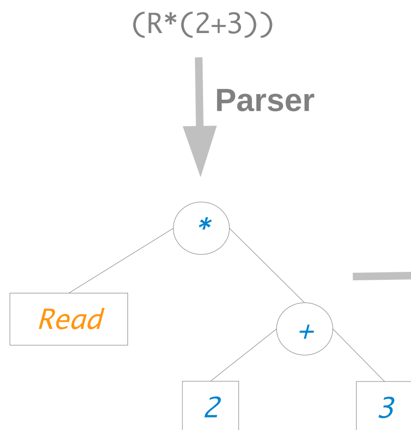
Vom Interpreter zum Compiler Beispiel Ausdruckssprache

Transformation: eval ~> genCode

Die Funktion eval wird mit einem Teil der Eingabe (dem AST) ausgeführt und liefert Code, der den Rest verarbeitet

Auswertbare Teilausdrücke werden zur Compilezeit berechnet.

Entweder kann der Ausdruck zur Compilezeit ausgewertet werden – compile liefert dann einen Int-Wert, oder
der Ausdruck kann nicht statisch ausgewertet werden – compile liefert dann Code der den Wert zur Laufzeit berechnet.




```
def genCode(e: Exp) : Either[Int, List[Instruction]] = e match {
  ...
}
```

```
List(
  READ,
  PUSH 5,
  MULT
)
```

Eval ~> genCode

```
object Codegenerator {

  private def f(op: Char) = (v1:Int, v2:Int) => op match {
    case '+' => v1 + v2
    case '-' => v1 - v2
    case '*' => v1 * v2
    case '/' => v1 / v2
    case _ => throw new IllegalArgumentException
  }

  private def c(op: Char) = op match {
    case '+' => Add
    case '-' => Sub
    case '*' => Mult
    case '/' => Div
    case _ => throw new IllegalArgumentException
  }

  def genCode(tree: ExpTree): Either[Int, List[Instruction]] = tree match {
      case Number(x) => Left(x)
      case ReadExp => Right(List(Rdint))
      case Operation(e1, op, e2) =>
        (genCode(e1), genCode(e2)) match {
          case (Left(v1), Left(v2))           => Left(f(op)(v1, v2))
          case (Right(instr1), Right(instr2)) => Right(instr1 ::: instr2 ::: List(c(op)))
          case (Left(v1), Right(instr2))      => Right(Pushc(v1) :: instr2 ::: List(c(op)))
          case (Right(instr1), Left(v2))      => Right(instr1 ::: List(Pushc(v2), c(op)))
        }
  }
}
```

f nimmt einen Operator (als Char) und liefert die entsprechende Operation (als Funktion Int x Int => Int).

Der Codegenerator wertet aus soweit es möglich ist, ansonsten generiert er Code.

Interpreter ~> Compiler

```
object Compiler {

  def compile(prog: String) : List[Instruction] = Codegenerator.genCode(Parser.parse(prog)) match {
    case Left(v) => List(Pushc(v), Wrint)
    case Right(instr) => instr ::: List(Wrint)
  }
}
```

```
object IDE {

  def main(args: Array[String]): Unit = {
    val expression = scala.io.StdIn.readLine()
    val machineCode = Compiler.compile(expression)
    StackMachine.reset
    StackMachine.load(machineCode)
    StackMachine.run
  }

}
```

Kontextanalyse: Konstantenfaltung

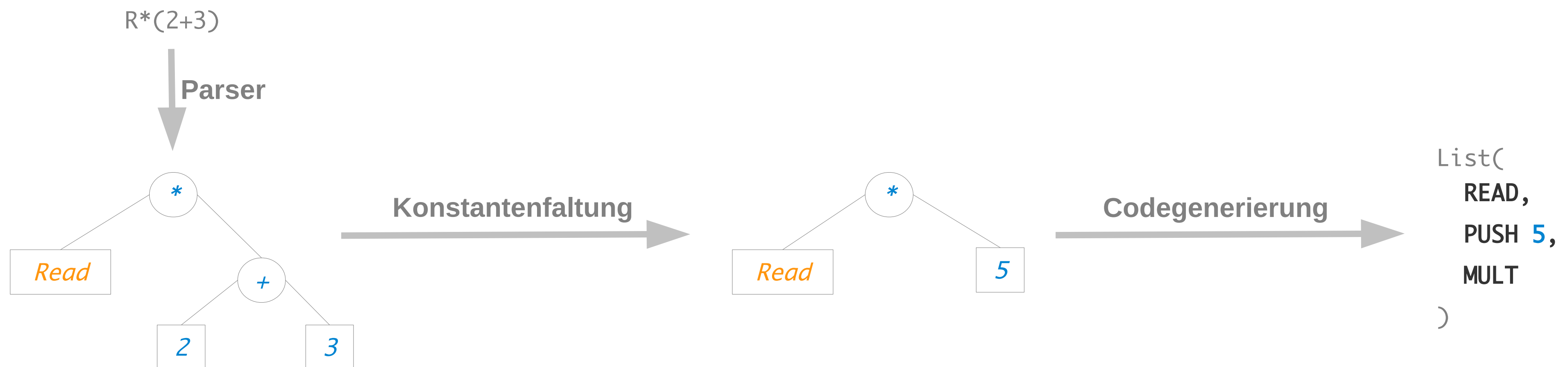
genCode (oben)

Auswertbare Teilausdrücke werden zur Compilezeit berechnet.

Das macht genCode komplizierter, da ein Teilausdruck einen Wert oder eine Anweisungsliste liefern kann

Konstantenfaltung / constant folding

Vereinfachung: Auswertungen die zur Compilezeit möglich sind, werden als Transformation des AST gespeichert.



Kontextanalyse / semantische Analyse

Konstantenfaltung ist nur ein Beispiel für eine Modifikation des AST zur Compilezeit. Solche Modifkationen werden semantische Analyse oder Kontextanalyse genannt.

Kontextanalyse: Konstantenfaltung

```
object ContextAnalyser {

  private def f(op: Char) = (v1:Int, v2:Int) => op match {
    case '+' => v1 + v2
    case '-' => v1 - v2
    case '*' => v1 * v2
    case '/' => v1 / v2
    case _ => throw new IllegalArgumentException
  }

  def constfold(ast: ExpTree): ExpTree = ast match {
    case Number(v) => Number(v)
    case ReadExp => ReadExp
      case Operation(e1, op, e2) =>
          val e1Folded = fold(e1)
          val e2Folded = fold(e2)
          (e1Folded, e2Folded) match {
            case (Number(v1), Number(v2)) => Number(f(op)(v1, v2))
            case _ =>  Operation(e1Folded, op, e2Folded)
          }
  }

}
```

Baumtransformation mit Konstantenfaltung

Kontextanalyse: Konstantenfaltung

Die Konstantenfaltung gehört zur Kontextanalyse

Bei der Kontextanlyse wird der AST analysiert und transformiert

Ziel: Ermöglichen / Vereinfachen der Codegenerierung

```
object Compiler {

  def compile(prog: String) : List[Instruction] =
    Codegenerator_II.genCode(
           ContextAnalyser.constfold(
                Parser.parse(prog))) ::: List(Wrint) ::: List(Halt)

}
```

Übersetzen:
- *Parsen*
- *Analysieren*
- *Code generieren*

```
object Codegenerator {

  private def c(op: Char) = op match {
    case '+' => Add
    case '-' => Sub
    case '*' => Mult
    case '/' => Div
    case _ => throw new IllegalArgumentException
  }

  def genCode(tree: ExpTree): List[Instruction] = tree match {
      case Number(x) => List(Pushc(x))
      case ReadExp => List(Rdint)
      case Operation(e1, op, e2) =>
          genCode(e1) ::: genCode(e2) ::: List(c(op))
  }

}
```

Der Codegenerator wertet nichts aus, er generiert nur noch Code.

Vom Interpreter zum Compiler

Kontextanalyse

Zur Kontextanalyse gehört viel mehr als nur Konstantenfaltung

Bei der Kontextanlyse wird der AST analysiert und transformiert

Ziel: Ermöglichen / Vereinfachen der Codegenerierung

Vorgehen: Verarbeiten von „kontext-sensitiven“ Informationen:

- Informationen, die bei der Syntaxanalyse nicht verarbeitet werden können weil sie „aus dem „Kontext“ eines Konstrukts kommen
- Beispiel: In einem Ausdruck kommt eine Name vor,
  dieser ist im Kontext („an völlig anderer Stelle“) definiert,
  wir brauchen aber Info über die Definition um Code generieren zu können
  - Intra-UE-Info: Info aus der gleichen Übersetzungseinheit
  - Extra-UE-Info: Info aus einer anderen Übersetzungseinheit

Vom Interpreter zum Compiler

Vom Interpreter zum Compiler

Prinzip: Statische- und dynamische Berechnungen trennen

Kontext-Analyse: Statische Berechnungen als Transformation des AST realisieren

Codegenerierung: Für dynamische Berechnungen Code generieren.

Compilation:

Parsen konkrete Syntax ~> abstrakte Syntax (AST)

Kontextanalyse Ast analysieren und transformieren

Codegenerierung AST ~> Maschinencode

Entwicklung eines Compilers

Analyse

A-1. Definiere die zu übersetzende Sprache

- **abstrakte Syntax** und informale Semantik
  Welche Konstrukte hat die Sprache, was bedeuten (bewirken) sie
  Ausdrücke,
  Anweisungen,
  Typen (statisch / dynamisch),
  Übersetzungseinheiten
  ausführbare Einheiten
  ...
- **konkrete Syntax**
  Wie solle die Programmtexte aussehen,
  mit welchem Verfahren sollen sie geparst (analysiert) werden
  Wie sieht die konkrete Syntax aus
- **Semantik**
  Beschreibe die Semantik exakt durch
  - eine Sprachdefinition, und / oder
  - einen (definitorischen) Interpreter

Entwicklung eines Compilers

Analyse

A-2. Analysiere die zu übersetzende Sprache

- Trenne statische und dynamische Aspekte:
  - Was kann zur Übersetzungszeit,
  - was kann zur Bindezeit,
  - was muss zur Laufzeit

  berechnet werden
- Welche Informationen müssen wann zur Verfügung stehen (üblicherweise getrennte Übersetzung !)
  - Bei einem Compilerlauf
  - Beim Binden von Objektmodulen
  - Beim Ausführen einer ausführbaren Einheite

Entwicklung eines Compilers

Analyse

A-3. Analysiere die Ziel-Sprache / das Zielsystem

Welche Möglichkeiten bietet die Zielsprache:
Architektur (Stack oder Register-Maschine),
Adressierungsarten,
...

Entwurf

E-1. Definiere Entwicklungs-Phasen

- Übersetzen (Compilieren)
- Binden
- Ausführen

Entwicklung eines Compilers

Entwurf

E-2. Definiere Compiler-Phasen

Trenne den Compiler in Phasen mit definierten Schnittstellen:

- **Syntaxanalyse**
  analysierte die syntaktische Struktur der Eingabe
  endet mit einem AST
- **Semantische Analyse / Kontextanalyse**
  analysiert die Eingabe unter Verwendung von Kontextinformationen:
  - **Inter-UE Analyse**
    Info aus anderen Übersetzungseinheiten (UEs)
    z.B. importierte Definitionen
  - **Intra-UE Analyse**
    Info aus der gleichen Übersetzungseinheit (UE)
    z.B. lokale Definitionen

  endet mit einem modifizierten AST und statischer UE-Info (eventuell Teil des Objektcodes)
- **Codegenerierung**
  endet in Objektmodulen oder Assemblercode

Implementierung

Implementiere Datenstrukturen und alle Phasen:

- **Definiere Datenstruktur: Abstrakte Syntax**
- **Implementiere Syntaxanalyse**
- **Definiere Datenstruktur: Modifizierte Abstrakte Syntax**
- **Definiere Datenstruktur: Speicherformat übersetzter Einheiten**
- **Implementiere Kontextanalyse**
- **Definiere Datenstruktur: Speicherformat von Objektmodulen bzw. Assemblercode**
- **Implementiere Codegenerierung**

Entwicklung eines Compilers