Compilerbau CS1019

Th. Letschert

TH Mittelhessen Gießen

University of Applied Sciences

Softwaretechnik im Compilerbau

– Phasenaufteilung

– Vom Zwischencode zum Zielcode

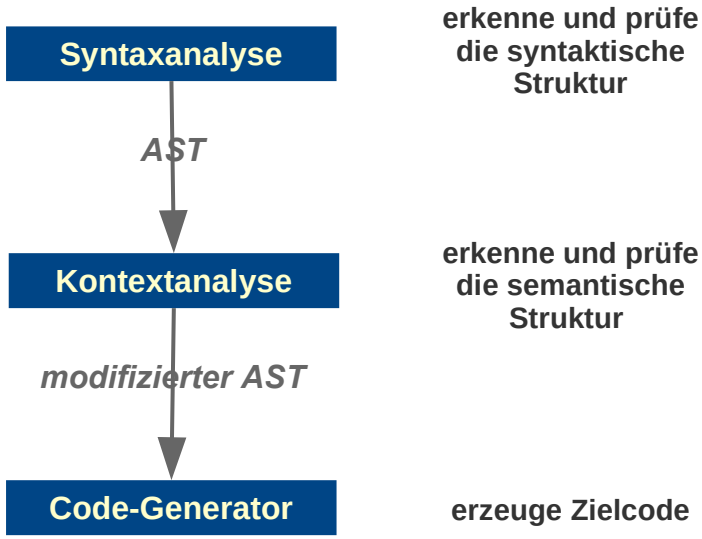
Phasen: Die Struktur eines Compilers

Compiler transformiert Quellcode in Zielcode

Die Basis-Struktur des Compiler:

- **Unterteilung dieser Transformation in Schritte, genannt Phasen**
- **Es gibt keine Vorgabe welche Phasen sinnvoll oder notwendig sind.**

Übliche Struktur:



Diese Phasen können weiter unterteilt werden.

Bestimmte Aufgaben können unterschiedlichen Phasen zugeteilt werden.

Die Phasen können in unterschiedlicher Art zusammenarbeiten.

Phasen und ihre Aufgaben

Die Aufgaben können den Phasen recht frei zugeordnet werden:

Beispiel Bezeichner-Identifikation / Symbolerzeugung

Die Identifikation der Bezeichner und die Erzeugung eines Symbols für jeden definierten Namen ist keine syntaktische Aufgabenstellung

Es ist trotzdem üblich sie in der Syntaxanalyse zu erledigen.

Beispiel Berechnung von Distanzadressen

Relativen Adressen (relativ zum Frame-Pointer) sind maschinen-abhängig und gehören „logisch“ in den Bereich der Codegenerierung.

Es ist trotzdem üblich ihre Berechnung der Kontextanalyse zuzuordnen

Beispiel abstrakte Syntax

Die konkrete Syntax ist das, was der Parser parst. Die abstrakte syntax ist die datenstruktur, die der Parser erzeugt. Die abstrakte Syntax ist „offiziell“ eine auf das wesentliche reduzierte Version der konkreten Syntax.

Es ist gelegentlich sinnvoll, den Parser Strukturen erzeugen zu lassen, die in der konkreten Syntax nicht (in dieser Form) zu finden sind.

Konversions-Operationen können beispielsweise bereits vom Parser in den AST eingefügt werden, obwohl sie keine Entsprechung im Quellcode haben (sie sind dazu gedachte Operationen)

Phasen und Unterphasen

Beispiel Syntaxanalyse

Kann unterteilt werden in

- Scanner / lexikalische Analyse und
- Parser / syntaktische Analyse

Die Unterphasen können in unterschiedlicher Art zusammenarbeiten

- Scanner erzeugt Tokenstrom
- Scanner produziert einzelne Tokens auf Anforderung des Parsers

Beispiel Codegenerierung

Kann unterteilt werden in

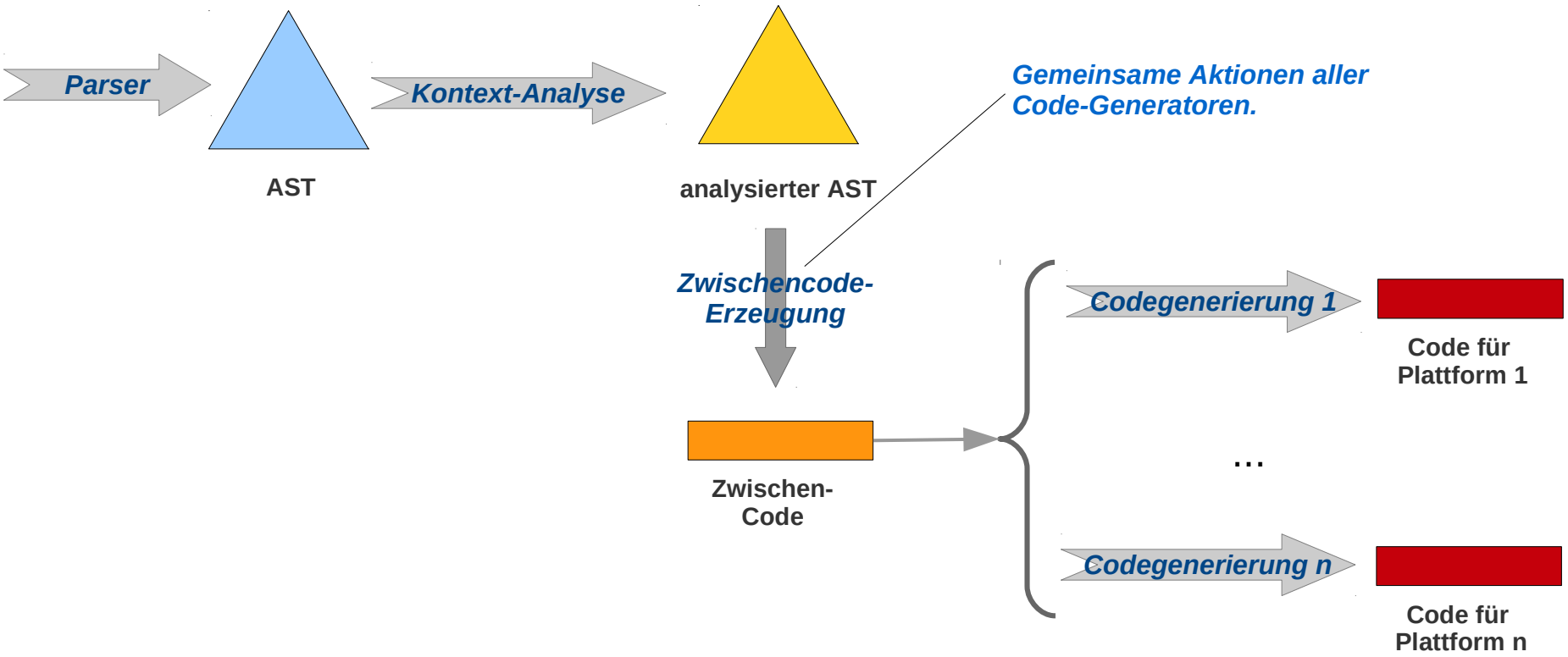
- Generierung von Zwischencode
- Generierung von Zielcode

Die Unterphasen können in unterschiedlicher Art zusammenarbeiten

- Zwischencode-Modul erzeugt Strom von Zwischencode-Anweisungen
  die dann in Zielcode umgewandelt werden.
- Zwischencode-Modul erzeugt eventuell rein virtuelle Zwischencode-Anweisungen
  die sofort / im Fluge in Zielcode umgewandelt werden.

Warum Zwischencode I: Kompakter Compiler

Die Erzeugung von Zwischencode ist vor allem dann sinnvoll, wenn der Compiler mehrere Zielplattformen hat:



Warum Zwischencode II: Strukturierung des Compilers

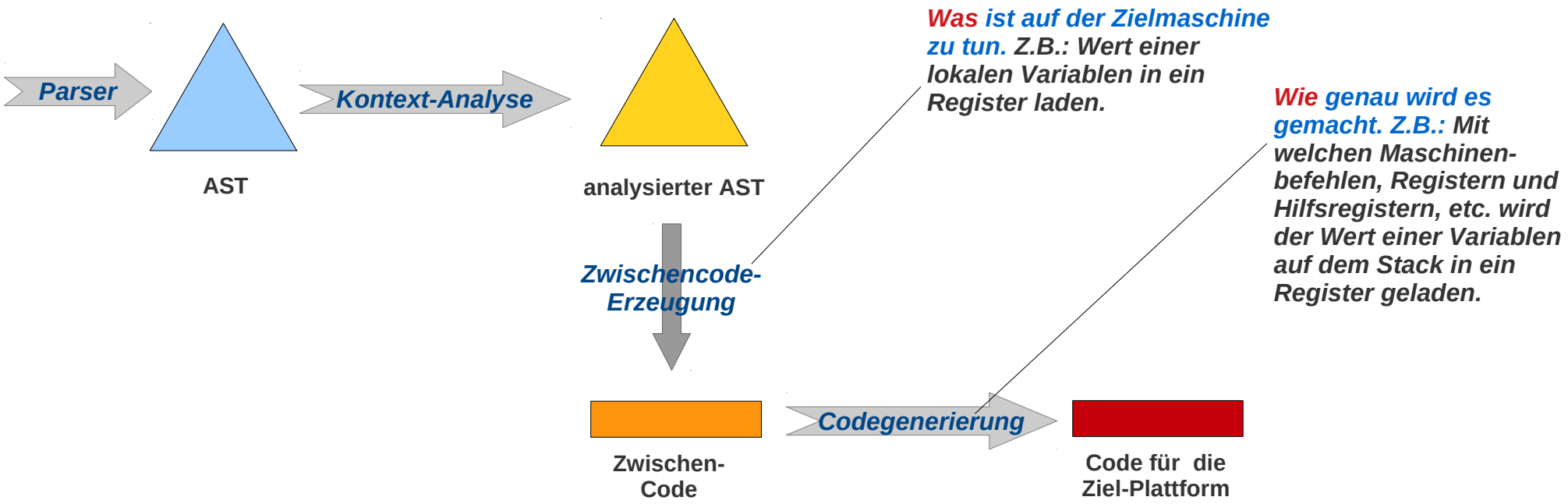
Die Erzeugung von Zwischencode ist aber auch dann sinnvoll, wenn der Compiler nur eine Zielplattformen hat

Die Aufteilung in Phasen ist das wichtigste SW-technische Mittel zu Strukturierung eines Compiler.

Eine zusätzliche Phase bringt eine zusätzliche Struktur:

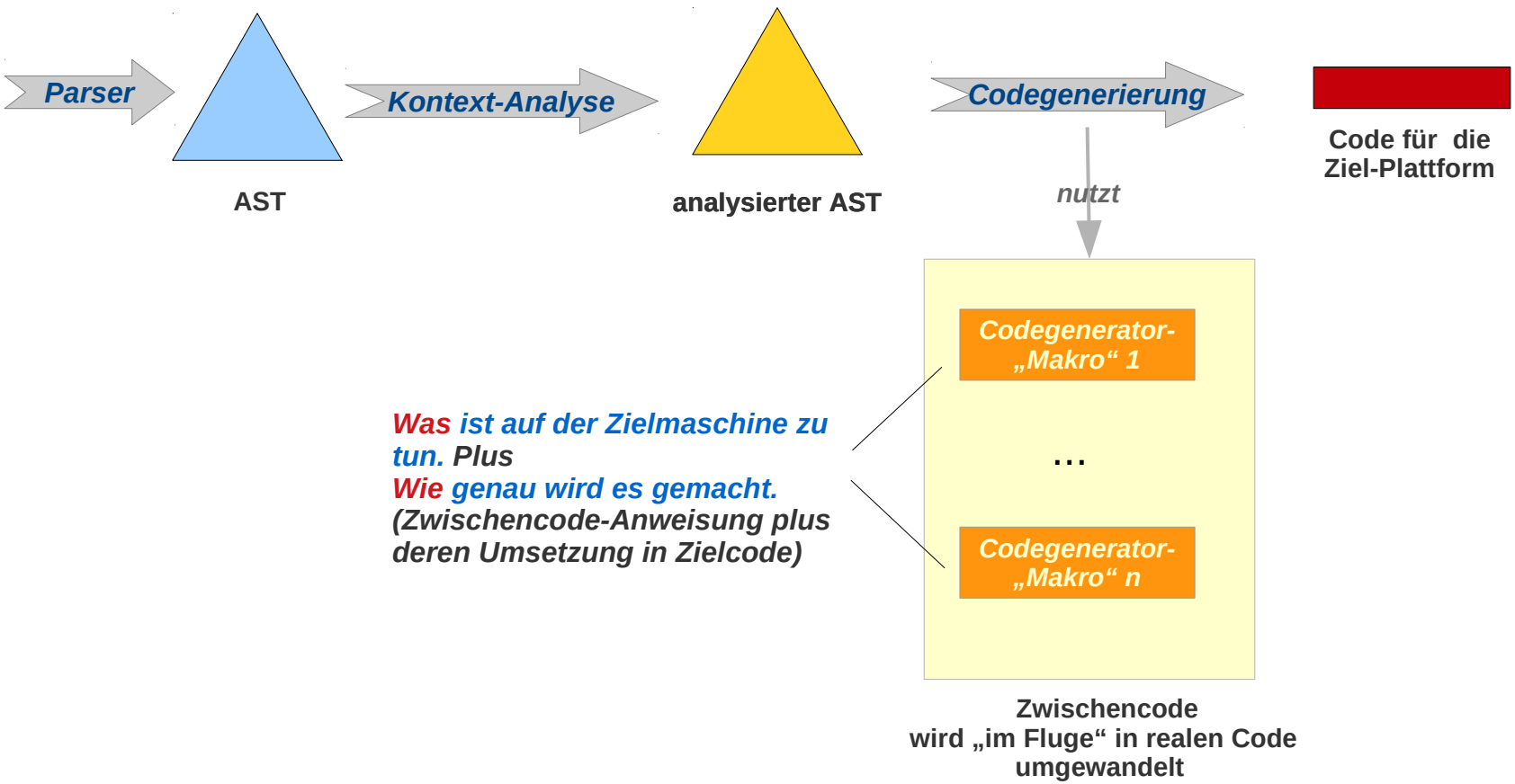
Konzeptionelle / grobkörnige Operationen auf der Zielplattform

- **werden identifiziert und**
- **dann im Detail umgesetzt**



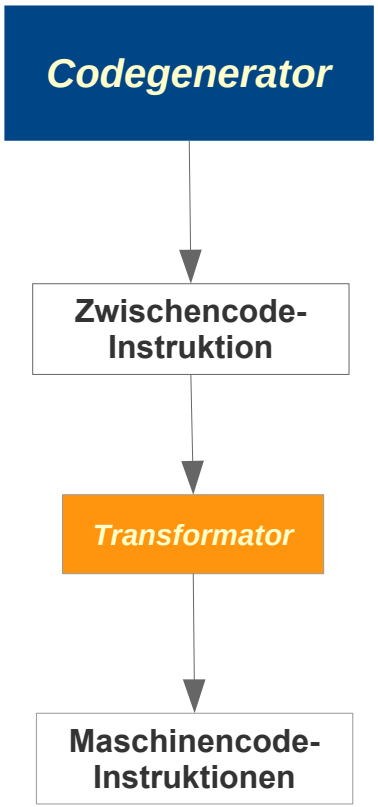
Zwischencode als virtuelle oder reale Datenstruktur

Der Zwischencode muss nicht zwingend real als Datenstruktur erzeugt werden.
Die gewünschte Strukturierung des Compilers kann auch erreicht werden, wenn der Zwischencode als Strukturierung des realen Codes erzeugt wird:

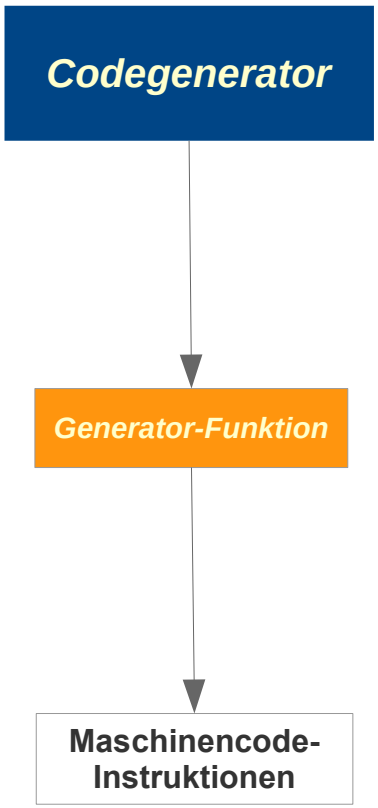


Zwischencode als virtuelle oder reale Datenstruktur

Mögliche SW-technische Realisation:



Jede Instruktion des Zwischencodes real als Datenstruktur erzeugt und dann im Fluge in realen Code transformiert



Jede Instruktion des Zwischencodes wird als Funktion implementiert

Zwischencode als virtuelle oder reale Datenstruktur

Mögliche SW-technische Realisation, Beispiel (1):

```
/**
  * a code builder accepts a sequence of intermediate code
  * instructions and transforms them to some "real" code.
  */
trait CodeBuilder {

  /**
    * Attaches instructions of the target language that correspond to an
    * intermediate code instruction to the code buffer.
    * @param instr the intermediate code instruction that is to be translated to real code
    */
  def += (instr: IntermediateInstr): Unit

  /**
    * Get the generated code as string.
    * @return the code generated up to the moment of calling this method.
    */
  def getCode: String

}
```

Zwischencode als virtuelle oder reale Datenstruktur

Mögliche SW-technische Realisation, Beispiel (2):

```
class C_CodeBuilder extends CodeBuilder {

  private val RR: String = "RR" // name of return register
  private val SP: String = "SP" // name of stack pointer register
  private val FP: String = "FP" // name of frame pointer register
  private val SDA: String = "SDA" // name of the register thar contains the address of static data area


  // the code buffer that is filled with target code.
  private val strBuf: StringBuilder = new StringBuilder


  // Format an instruction of the intermediate language as
  // instructions of the C language.
  def += (instr: IntermediateInstr): Unit = instr match {

    case AssignInstr(d, Some(op1), Some(op), Some(op2)) =>
      strBuf ++= c"\t$d = $op1 $op $op2;\n"

    ...

  }

  ...

}
```

angepasste String-Interpolation

```
// define string interpolation for C-formatting
implicit private class CHelper(val sc: StringContext) {
  def c(args: Any*): String = {
    ...
  }
}
```

Zwischencode als virtuelle oder reale Datenstruktur

Mögliche SW-technische Realisation, Beispiel (3):

```
// this method translates values and location of the intermediate code
// to values and location of the C language.
private def toC(arg: Any): String = arg match {
  case MIntProgLoc(locInfo) =>
    if (locInfo.nesting > 0) s"($FP.cellRef + ${locInfo.offset})->value"
    else s"($SDA.cellRef + ${locInfo.offset})->value"

  case MIntImmediateValue(d:Int) =>
    d.toString

  case TempMIntLoc(nr) =>
    s"T_$nr.value"

  case TempMAddressLoc(nr) =>
    s"(A_$nr.cellRef)"

  case MAddressProgLoc(locInfo) =>
    if (locInfo.nesting > 0) s"($FP.cellRef + ${locInfo.offset})->cellRef"
    else s"($SDA.cellRef + ${locInfo.offset})->cellRef"

  case DeRef(addrLoc) =>
    toC(addrLoc)+"->value"

  case MkRef(mIntLoc) =>
    mIntLoc match {
      case MIntProgLoc(locInfo: RTLocInfo) =>
        if (locInfo.nesting > 0) s"$FP.cellRef + ${locInfo.offset}"
        else s"$SDA.cellRef + ${locInfo.offset}"

      case DeRef(x) => toC(x)  // &*x  & and * cancel each other

    }

  case x => x.toString
}
```

Zwischencode als virtuelle oder reale Datenstruktur

Mögliche SW-technische Realisation, Beispiel (4):

```
// define string interpolation named c interpolation
implicit private class CHelper(val sc: StringContext) {
  def c(args: Any*): String = {
    val pi = sc.parts.iterator
    val ai = args.iterator
    val bldr = new java.lang.StringBuilder(scala.StringContext.treatEscapes(pi.next()))
    while(pi.hasNext) {
      val part = scala.StringContext.treatEscapes(pi.next)
      val arg = toC(ai.next)
      bldr append arg
      bldr append part
    }
    bldr.toString
  }
}
```