# Compilerbau CS1019


**Th. Letschert**

**TH Mittelhessen Gießen**

***University of Applied Sciences***


## Code für Ausdrücke und Anweisungen

- Codegenerierung und Kontextanalyse
- Zwischencode: Drei-Adress-Code
- Übersetzung von Ausdrücken
- Laufzeit-Typen
- Übersetzung von Speicherzugriffen und Kontrollstrukturen

# Codegenerierung im Kontext des Compilers

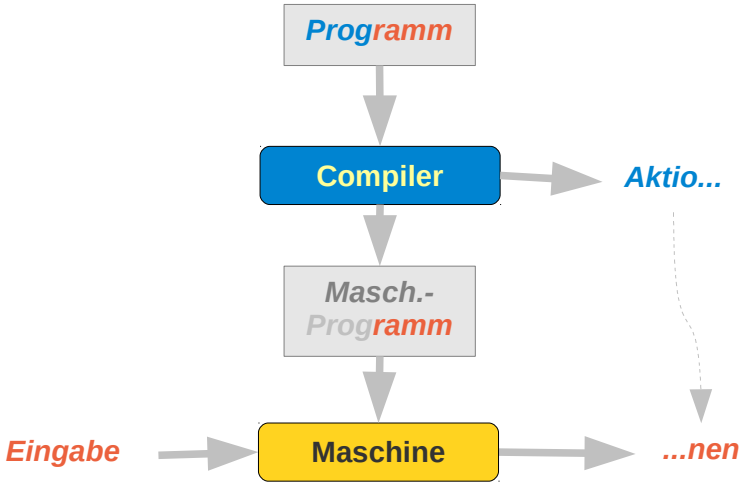
*Die zentrale Fragestellung des Compiler-Bauers ist: „Was ist statisch und was ist dynamisch?“*

## Vom Interpreter zum Compiler

**Ein Compiler ist ein Interpreter, der**

- **möglichst viel zur Übersetzungszeit erledigt**
- **und für das, was er nicht erledigen kann, Maschinencode erzeugen.**

**Der Compiler kann das nicht erledigen, das direkt oder indirekt von der Programm-Eingabe abhängt**



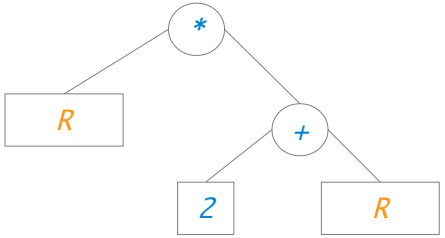

*Der Teil der Aktionen, der unabhängig von der Eingabe ist, kann bereits Compiler ausgeführt werden.*

*Für den Rest muss Code erzeugt werden*

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

## Vom Interpreter zum Compiler

**Der erzeugte Code muss den Fähigkeiten der Zielmaschine angepasst sein**




*Ein Interpreter in einer Hochsprache könnte dies problemlos mit rekursiven Funktionen auswerten*

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

*Rekursion wird von der HW nicht unterstützt. Darum Code für eine Stack-Maschine mit deren beschränkte Möglichkeiten. Die Rekursion muss vom Compiler „im Voraus“ abgewickelt werden.*

# Codegenerierung im Kontext des Compilers

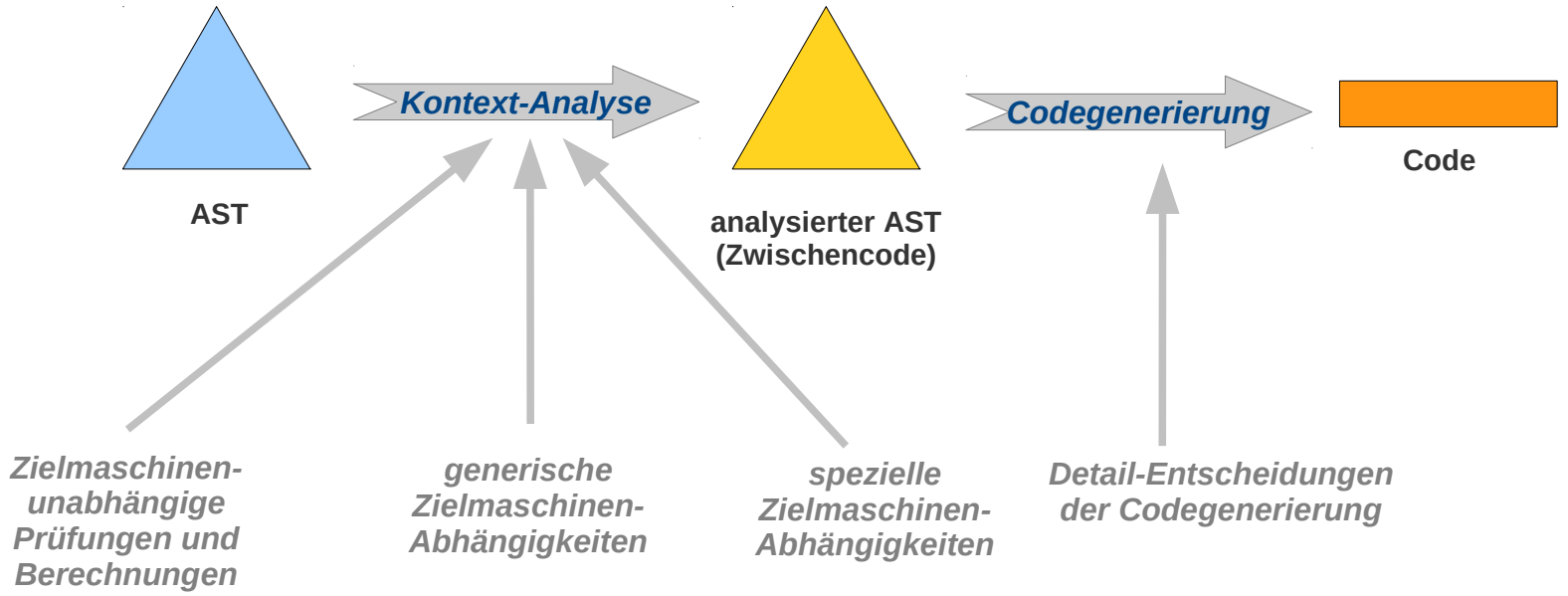
## Kontextanalyse und Codegenerierung

**Der Compiler muss bei der Codegenerierung die Fähigkeiten der Zielmaschine beachten**

**Die Möglichkeiten**

- **eines Interpreters (in einer Hochsprache) müssen in die Möglichkeiten**
- **der Zielsprache übersetzt werden**

**In der Kontextanlyse wird dies vorbereitet.**

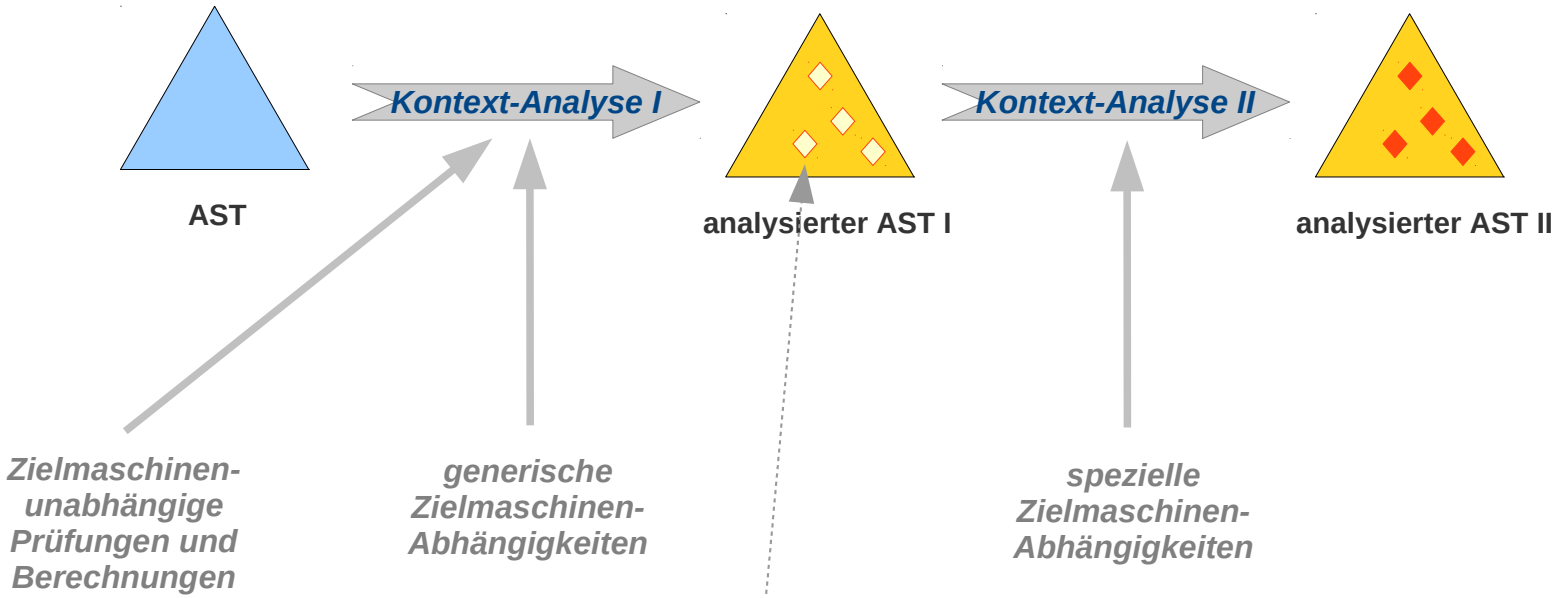


## Kontextanalyse und Codegenerierung

### Organisation der Kontextanalyse

**Aufteilung in Phasen**

**AST enthält „Löcher“ die in späteren Phase gefüllt werden**




```
class VarSymbol(
   ...
   val staticTyp: Option[TypeInfo] = = None,
   var runtimetInfo: Option[LocationInfo] = None
)
```

# Codegenerierung: Für welche Zielmaschine

## Code für hypothetische, virtuelle oder reale Maschine

### Code-Erzeugung für reale Maschine

- Aus dem (analysierten) AST wird Code für eine reale Maschine generiert

### Code-Erzeugung für virtuelle Maschine

- Aus dem (analysierten) AST wird Code für eine virtuelle Maschine generiert
- Virtuelle Maschine: Software-Implementierung einer Maschine (Interpreter / Emulator)

### Code-Erzeugung für hypothetische Maschine

- Aus den (analysierten) AST wird Code für eine hypothetische Maschine generiert
- Hypothetische Maschine: Maschine die
  - weder in Hardware noch
  - als Software-Implementierung

  existiert.

  Der Zwischencode wird ausschließlich als Zwischenstufe vor der eigentlichen Codegenerierung erzeugt.

# Codegenerierung: Direkt oder via Zwischencode

## Code für hypothetische, virtuelle oder reale Maschinen

### Code-Erzeugung

**Komplexe Thematik, beeinflusst von**

- **Ausdrucksmöglichkeiten der Quellsprache**
- **Ausdrucksmöglichkeiten der Zielsprache**
- **Entwurfs- und Codierungsentscheidungen**

# Codegenerierung: Direkt oder via Zwischencode

## Code für hypothetische, virtuelle oder reale Maschinen

### Code-Erzeugung direkt oder via Zwischencode

Die Codeerzeugung für die reale Maschine kann

- direkt aus dem analysierten AST erfolgen, oder
- über einen Zwischencode (Code für eine virtuelle / hypothetische Maschine) gehen

Vorteile der Erzeugung von Zwischencode für eine virtuelle / hypothetische Maschine

- Der Zwischencode kann alternativ:
  - in Code einer realen Maschine übersetzt, oder
  - von einer virtuellen Maschine interpretiert werden

  Ansatz der VM von Java
- Die Codeerzeugung
  - kann leicht an eine andere reale Maschine angepasst werden, oder
  - für viele unterschiedliche Zielsysteme angeboten werden

  Ansatz des Gnu-Compilers gcc
- Die Codeerzeugung wird in zwei Phasen mit jeweils geringerer Komplexität aufgeteilt

Nachteile der Erzeugung von Zwischencode für eine hypothetische Maschine

- weniger effizient

## Zwischencode: Varianten

### AST (Baum oder baumartige Datenstruktur)

- Der (analysierte) AST kann als Zwischencode betrachtet werden
- Nachteil: sehr weit weg vom realen Code

### Linearer Zwischencode

- Der Zwischencode besteht aus Anweisungen für eine virtuelle / hypothetische Maschine

# Zwischencode: Varianten

## Varianten von virtuellen / hypothetischen Maschinen

- **Stack-Maschine**

  Auswertungen von Ausdrücken finden auf einem (Operations-/Rechen-) Stack statt

- **Register-Maschine**

  Auswertungen von Ausdrücken finden auf Registern statt

  Register: benannte Speicherplätze der CPU

- **Register- oder Stack-Maschine mit Funktionsunterstützung**

  Neben dem Stack (bzw. den Registern) für Ausdrucksauswertungen Ausdrucksauswertung bietet die Maschine einen Stack für die Auswertung von Funktions- / Prozedur-aufrufen (*Callstack* oder einfach *Stack*) der mit speziellen Befehlen genutzt werden kann, z.B.:

```
call functionName
...
return
```

# Drei-Adress-Code: Arithmetische Operationen

## Drei-Adress-Code

Der Code besteht aus einer Folge von Operationen mit maximal drei Operanden

Geeignet als Zwischencode für Registermaschinen als Zielmaschine

## Drei-Adress-Code für arithmetische Operationen

Format einer Anweisung

```
x := y op z
```

`op`: eine Operation

`x,y,z`: Operanden:

- Variablen (Speicherstellen, entsprechen Variablen / Parametern der Quellsprache), oder
- temporäre Variablen (Register, Hilfsvariablen der Codegenerierung) oder
- direkte Werte

  meist darf nur ein Operand ein direkter Wert sein

## Beispiel

```
t1 := 5
t2 := x
t2 := t2 * t1
t3 := t2 + 2
```

*`x` ist eine Variable der Quellsprache `t1` – `t3` sind Hilfsvariablen der Codegenerierung die eventuell als Register realisiert werden.*

# Erzeugung von Drei-Adress-Code für arithmetische Ausdrücke

## Quellsprache

Arithmetische Ausdrücke

## Zielsprache

Instruktionen im 3-Adress-Format

## Wichtige Aufgabe: Verwaltung von Hilfsvariablen

Hilfsvariablen (möglichst als Register realisiert) sind ein knappes Gut (im Zielcode)

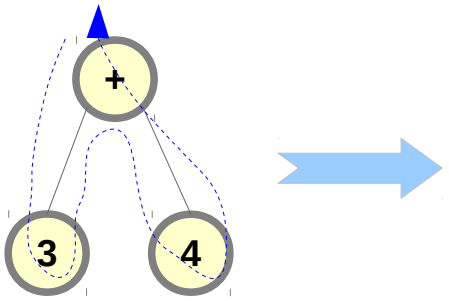
Die Verwaltung der Hilfsvariablen kann der Struktur der Ausdrücke folgen:

Für jeden Unterbaum im Ausdrucksbaum

- werden Hilfsvariablen allokiert
- diese können nach dem Abarbeiten des Unterbaums wieder frei gegeben werden

## Erzeugung von Drei-Adress-Code für Ausdrücke

### Arithmetische Operationen ~> Drei-Adress-Code




```
Load 3
Load 4
ADD
```

*Stack-Code*

```
t1 := 3
t2 := 4
t3 := t1 + t2
```

*Drei-Adress-Code*

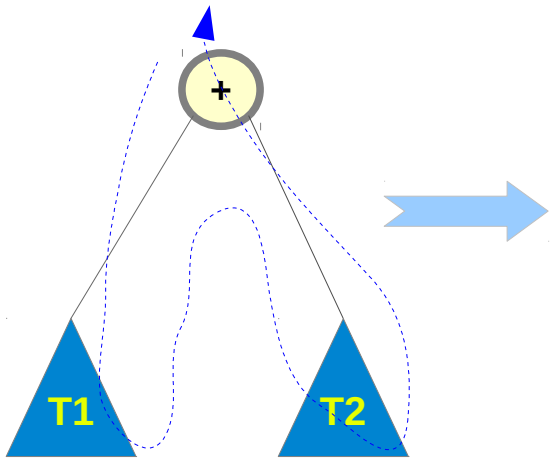
```
t1 := 3
t2 := 4
t1 := t1 + t2
```

*Drei-Adress-Code, optimiert*

*Post-oder-Traversierung auch bei der Erzeugung von drei-Adress-Code*

## Erzeugung von Drei-Adress-Code für Ausdrücke

### Arithmetische Operationen ~> Drei-Adress-Code




*Code für T1*
*Code für T2*
ADD

*Stack-Code*

*Code für T1 ~> t1*
*Code für T2 ~> t2*
`t3 := t1 + t2`

*Drei-Adress-Code*

*Code für T1 ~> t1*
*Code für T2 ~> t2*
`t1 := t1 + t2`

*Drei-Adress-Code, optimiert*

*Post-oder-Traversierung auch bei der Erzeugung von drei-Adress-Code*

## Drei-Adress-Code für Ausdrücke

Ein Ausdruck hat einen Wert, der Wert muss irgendwo gespeichert werden.

**Wer entscheidet wo der Wert eines Ausdrucks gespeichert wird:**

- **Auftragnehmer / Bottom-Up**: Der Verarbeiter eines **Unter-Ausdrucks** entscheidet wo der Wert eines Unterausdrucks zu finden ist

```
def genCodeExp(exp: ExpTree): Location = ...
```

*erzeugt Code der den Wert des Ausdrucks hier speichert*

- **Auftraggeber / Top-Down**: Der Verarbeiter eines **Ausdrucks** entscheidet, wo die Werte der Unterausdrücke zu platzieren sind.

```
def genCodeExp(exp: ExpTree, target: Location): Unit = ...
```

Beide Strategien sind weitgehend äquivalent.

Meist wird die Top-Down-Strategie verwendet:

Bei einer Zuweisung kann die Zieladresse direkt übergeben werden.

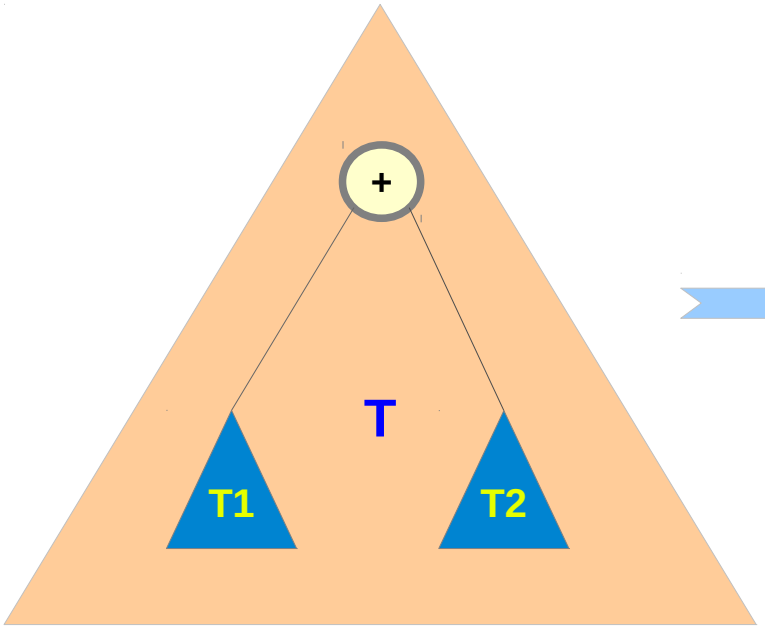
Die Methode zur Erzeugung des (Zwischen-) Codes hat dann die Zieladresse als Parameter.

Beide Methoden können gemischt verwendet werden, z.B.:

- R-Ausdrücke: top-down
- L-Ausdrücke: bottom-up

## Erzeugung von Drei-Adress-Code für Ausdrücke

### Arithmetische Operationen ~> Drei-Adress-Code




```
allocate t1
genCode(T, t1) = {
      genCode(T1, t1)
      allocate t2
      genCode(T2, t2)
      t1 := t1 + t2
      free t2
}
```

*Erzeugung von Code der den Wert von T in t1 legt*

*Codeerzeugung mit vorgegebenem Zuweisungsziel (Top-down)*

# Drei-Adress-Code für arithmetische Ausdrücke

## Code für einfache Ausdrücke / Beispiel

```
val e: Exp = Mul( Add( Number(1),
                       Number(2)
                     ),
                  Number(3)
                )

genCodeValExp(e) foreach( println(_) )
```

mit ein paar geeigneten toString-Methoden

```
t_1 = 1
t_2 = 2
t_0 = t_1 + t_2
t_1 = 3
t_0 = t_0 * t_1
```

# Drei-Adress-Code für arithmetische Ausdrücke / Beispiel

## Quellsprache (AST)

### Arithmetische Ausdrücke (einfaches Beispiel)

```
case class Number(d: Int)         extends Exp
case class Add(e1: Exp, e2: Exp) extends Exp
case class Sub(e1: Exp, e2: Exp) extends Exp
case class Div(e1: Exp, e2: Exp) extends Exp
case class Mul(e1: Exp, e2: Exp) extends Exp
```

# Drei-Adress-Code für arithmetische Ausdrücke / Beispiel

## Zielsprache (Zwischencode)

### Anweisungen für arithmetische Operationen (1)

```
// Location or value on the (virtual) target machine
sealed abstract class LocOrValue

// Location
sealed abstract class Location extends LocOrValue

// Numeric value
case class ImmediateValue(x: Int) extends LocOrValue

// Temporary location (created by code generation)
case class TempLoc(nr: Int) extends Location

// Operations on the (virtual) target machine
sealed abstract class MOp
case object AddOp extends MOp
case object SubOp extends MOp
case object MultOp extends MOp
case object DivOp extends MOp

// instruction on the (virtual) target machine
sealed abstract class Instr

// Binary operation: x := y op z
case class AssignInstr(
  dest:     Location,
  operand1: Option[Location],
  op:       Option[MOp],
  operand2: Option[LocOrValue]
) extends Instr
```

Eine `AssignInstr` kann einen oder zwei Operanden haben. Das wird hier durch Option zum Ausdruck gebracht.

# Drei-Adress-Code für arithmetische Ausdrücke / Beispiel

## Zielsprache (Zwischencode)

### Anweisungen für arithmetische Operationen (2)

Mit diesen apply-Funktionen können `AssignInstr`s leichter erzeugt werden: der Umgang mit Some und None wird „automatisiert“.

```
// facilitate generation of assign instructions
object AssignInstr {

  def apply(dest: Location, operand1: Location, op: MOp, operand2: LocOrValue) : AssignInstr =
    AssignInstr(dest, Some(operand1), Some(op), Some(operand2))

  def apply(dest: Location, operand1: Location) : AssignInstr =
    AssignInstr(dest, Some(operand1), None, None)

  def apply(dest: Location, operand2: ImmediateValue) : AssignInstr =
    AssignInstr(dest, None, None, Some(operand2))
}
```

# Drei-Adress-Code für arithmetische Ausdrücke / Beispiel

## Code für einfache Ausdrücke

```
/** Generate code for an expression.
  * @param exp  the expression to be translated
  * @return     intermediate code for exp
  */
def genCode(exp: Exp) : List[Instr] = {


  def genBinOp(l: Exp, op:MOp, r: Exp, target: MIntLoc): Unit = {
    val t1 = acquireMIntTemp()
    genCodeValExp(l, t1)
    val t2 = acquireMIntTemp()
    genCodeValExp(r, t2)
    codeBuf += AssignInstr(target, t1, op, t2)
    releaseMIntTemp(t1)
    releaseMIntTemp(t2)
  }


  def genCodeValExp(exp: Exp, target: Location): Unit = exp r
    case Add(l, r) => genBinOp(l, AddOp, r, target)
    case Sub(l, r) => genBinOp(l, SubOp, r, target)
    case Mul(l, r) => genBinOp(l, MultOp, r, target)
    case Div(l, r) => genBinOp(l, DivOp, r, target)
    case Number(v) =>
      codeBuf += AssignInstr(target, ImmediateValue(v))
  }

  var t = acquireTemp()
  genCodeExp(exp, t)
  codeBuf.toList
 }
```

Verwaltung der Hilfs-Speicherplätze;
Puffer für den erzeugten Code

```
// create temporary locations
var temps: List[Int] = (0 to tempsCount).toList

// get next unused temporary
def acquireTemp(): TempLoc = {
  val res = TempLoc(temps.head)
  temps = temps.tail
  res
}

def releaseTemp(t: TempLoc) : Unit = {
  temps = t.nr :: temps
}

val codeBuf : ListBuffer[Instr] = new ListBuffer()
```

# Drei-Adress-Code: Logische Operationen

## Code für logische Operationen

- Codierung der booleschen Werte als numerische Werte

  0 ~ true, 1 ~ false

- Prozedurale Codierung:

  Alle booleschen Operationen werden über Sprung-Anweisungen realisiert

  Einfach wenn die Sprache boolesche Werte nicht direkt, sondern nur als Ergebnis von Vergleichen in Bedingungen verwendet.

## Code für bedingte und unbedingte Sprünge

- Reale Maschinen unterstützen in der Regel

  bedingte und unbedingte Sprünge als Maschinen-Instruktionen

- Format im 3-Adress-Code:

  if ( <conditition> ) goto <label>

  goto <label>

# Übersetzung von bedingten Ausdrücken

**Bedingte Ausdrücke können auf zwei Arten behandelt werden:**

## Emulation durch numerische Werte

**Boolesche Werte werden im Zielcode durch numerische Werte repräsentiert**

**z.B. 0 ~ false / 1 ~ true**

**und boolesche Operationen werten durch numerische Operationen nachgebildet**

**z.B. not ~ *-1**

## Emulation durch Sprünge

**Boolesche Werte werden im Zielcode nicht direkt dargestellt sondern komplett in Sprunganweisungen übersetzt.**

**Dies ist nur möglich, wenn boolesche Ausdrücke nur in Bedingungen auftreten können.**

# Kontrollstrukturen im Zwischencode

## Bedingte und unbedingte Sprünge

**Kontrollstrukturen (und eventuell auch boolesche Ausdrücke) werden in bedingte und unbedingte Sprünge der Zielmaschine übersetzt**

**Dazu müssen Anweisungen auch mit Sprung-Marken (Labels) versehen werden können.**

### Beispiel:

```
WHILE a < b DO
  a := a + b;
OD
```

```
L_1: NOOP
     t_0 = a
     t_1 = b
     IF(t_0 < t_1) GOTO L_2
     GOTO L_3
L_2: NOOP
     t_1 = a
     t_0 = b
     a = t_1 + t_0
     GOTO L_1
L_3: NOOP
```

# Drei-Adress-Code: Kontrollstrukturen im Zwischencode

## Zwischencode / Beispiel:

```
// Conditional jump
// if (operator1 op operator2) goto jumpTo
case class IfInstr(
     operand1: MIntLocOrValue,
     op:       MRelOp,
     operand2: MIntLocOrValue,
     jumpTo: String
) extends Instr

// Unconditional Jump: goto label
case class JumpInstr(label: String) extends Instr

// label: Noop
case class LabeledInstr(label: String) extends Instr


// Compare operations
abstract class MRelOp
case object EqOp extends MRelOp
case object NeOp extends MRelOp
case object LsOp extends MRelOp
case object GtOp extends MRelOp
case object LeOp extends MRelOp
case object GeOp extends MRelOp
```

# Drei-Adress-Code: Kontrollstrukturen übersetzen

## Erzeugung von Zwischencode / Beispiel:

```
private def genCodeCmd(cmd: Cmd): Unit = cmd match {

  ...

  case While(cond, body) =>
    val startLabel = newLabel
    val continueLabel = newLabel
    val endLabel = newLabel
    codeBuf += LabeledInstr(startLabel)
    genCodeBe(cond, continueLabel)
    codeBuf += JumpInstr(endLabel)
    codeBuf += LabeledInstr(continueLabel)
    body.foreach { cmd => genCodeCmd(cmd) }
    codeBuf += JumpInstr(startLabel)
    codeBuf += LabeledInstr(endLabel)

  ...

}
```

```
// Create labels
var labelCount = 0
def newLabel : String = {
  labelCount = labelCount + 1
  "L_"+labelCount
}
```

# Übersetzung von bedingten Ausdrücken

## Erzeugung von Zwischencode / Beispiel:

```
/* Generates code that jumps to a given label if the boolean expression evaluates to true.
 * Parameters:
 *   - bExp: the boolean expression, i.e. a comparison
 *   - trueLabel: the label to jump to if the comparison evaluates to true
 */
def genCodeBe(bExp: BoolExp, trueLabel: String): Unit = {
  def genCondJump(l: ValueExp, r: ValueExp, compOp: RelOperation) : Unit = {
    val t1 = acquireTemp
    genCodeValueExp(l, t1)
    val t2 = acquireTemp
    genCodeValueExp(r, t2)
    codeBuf += IfInstr(t1, compOp, t2, trueLabel)
    releaseTemp(t1)
    releaseTemp(t2)
  }
  bExp match {
    case Less(l: ValueExp, r: ValueExp) =>
        genCondJump(l, r, LsOp)
    case Greater(l: ValueExp, r: ValueExp) =>
        genCondJump(l, r, GtOp)
    case Equal(l: ValueExp, r: ValueExp) =>
        genCondJump(l, r, EqOp)
    case LessEqual(l: ValueExp, r: ValueExp) =>
        genCondJump(l, r, LeOp)
    case GreaterEqual(l: ValueExp, r: ValueExp) =>
        genCondJump(l, r, GeOp)
    case NotEqual(l: ValueExp, r: ValueExp) =>
        genCondJump(l, r, NeOp)
  }
}
```

# Übersetzung von bedingten Anweisungen

## Erzeugung von Zwischencode / Beispiel:

```
def genCodeCmd(cmd: Cmd): Unit = cmd match {

  ...

  case If(cond, thenCmds, Nil) =>
    val thenLabel = newLabel
    val exitLabel = newLabel
    genCodeBe(cond, thenLabel)
    codeBuf += JumpInstr(exitLabel)
    codeBuf += LabeledInstr(thenLabel)
    thenCmds.foreach { cmd => genCodeCmd(cmd) }
    codeBuf += LabeledInstr(exitLabel)

  case If(cond, thenCmds, elseCmds) =>
    val thenLabel = newLabel
    val elseLabel = newLabel
    val exitLabel = newLabel
    genCodeBe(cond, thenLabel)
    elseCmds.foreach { cmd => genCodeCmd(cmd) }
    codeBuf += JumpInstr(exitLabel)
    codeBuf += LabeledInstr(thenLabel)
    thenCmds.foreach { cmd => genCodeCmd(cmd) }
    codeBuf += LabeledInstr(exitLabel)

  ...

}
```

***einarmiges If***

***zweiarmiges If***

## Speicherzugriffe im Zwischencode

Der Compiler erzeugt zunächst Zwischencode der dann in den „echten“ Zielcode übersetzt wird.
Der Zwischencode sollte unabhängig von einer konkreten Zielmaschine sein,
aber aber die wesentlichen Merkmale der möglichen Zielmaschine(n) haben

### Speicherverwaltung der hypothetischen Maschine

Der Speicher der Zielmaschine besteht hier aus drei Sektionen:

- Code
- statische Daten
- dynamische Daten

#### Beispiel Minisprache

Der statische Datenbereich wird für globale Variablen und Konstanten verwendet
Der dynamischen Datenbereich wird für Prozeduren und eventuell für temporäre Variablen benötigt

# Typen in der Quell- Zwischen- und Zielsprache

**Die Quellsprache hat in der Regel ein mehr oder weniger komplexes Typsystem.
Aber auch die Zielsprache ist nicht komplett typfrei. Auch wenn stets Bitmuster manipuliert werden, ist es alles andere als gleichgültig, welche Interpretation (~Typ) diese haben.**

## Typen und Maschinencode

- Im Maschinencode werden Bitmuster manipuliert.
- Die Interpretation der Bitmuster hängt von den verwendeten Operationen ab
- Die Größe der Speicherbereiche für Werte / Variablen hängt von deren Typ ab

## Primitive und strukturierte Typen im Maschinencode

- Maschinen unterstützen i.d.R. nur primitive Typen:
  - Ganzzahlen mit oder ohne Vorzeichen verschiedener Länge
  - Gebrochene Zahlen verschiedener Länge
  - Pointer / Addressen
- Strukturierte Typen der Quellsprache werden als Sequenzen von primitiven Typen codiert
- Operationen auf strukturierten Typen werden in Operationen auf primitiven Typen übersetzt

# Typen in der Quell- Zwischen- und Zielsprache

## Typen im Zwischencode

- Die Verwendung von Typen im Zwischencode ist eine im Prinzip freie Entwurfsentscheidung
- Pragmatische Entscheidung, nahe an der Zielsprache, aber ohne spezielle Eigenheiten anzunehmen:
  - Strukturierte Typen sind im Zwischencode auf primitive Typen reduziert
  - Die Typen im Zwischencode sind die primitiven Typen der Quellsprache.
    Über deren Umsetzung in Maschinensprache entscheidet die Codegenerierung

Beispiel:

- Zwischencode kennt Int (=signed Int) und Adressen
- Alle Speicherstellen sind typisiert (mit den Typen den Zwischencodes)
- 3-Address-Operationen berücksichtigen die Typen

# Werte und Speicherstellen im Zwischencode

## Int-Werte und -Speicherstellen in der Zwischensprache

```
//-----------------------------------------------------
// int values and locations

// int value or a location with an int value
sealed abstract class MIntLocOrValue

// a location with an int value
sealed abstract class MIntLoc extends MIntLocOrValue

// a location in an stack frame with an int value
case class MIntFrameLoc(locInfo: RTLocInfo) extends MIntLoc

// immediate int value
case class MIntImmediateValue(d:Int) extends MIntLocOrValue

// Temporary MInt location (created by code generation)
case class TempMIntLoc(nr: Int) extends MIntLoc
```

***Speicherstellen und Werte der Zwischensprache haben einen (primitiven / Zielsprachen-orientierten) Typ***

# Werte und Speicherstellen im Zwischencode

## Adress-Werte und -Speicherstellen in der Zwischensprache

*Die Zwischensprache kennt Adressen: Referenzen sind Adress-Werte*

```
//-----------------------------------------------------
// address values and locations

// Locations that contain an address
sealed abstract class MAddressLoc

// a location in an stack frame with an address value
case class MAddressFrameLoc(locInfo: RTLocInfo) extends MAddressLoc

// Temporary location that contains an address (created by code generation)
case class TempMAddressLoc(nr: Int) extends MAddressLoc
```

# Drei-Adress-Code für Speicherzugriffe

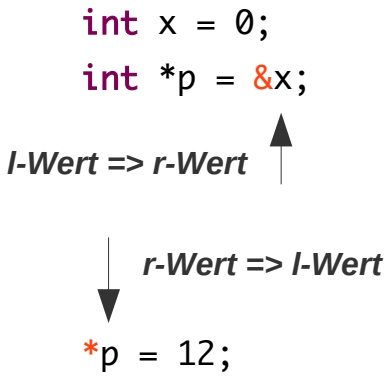
## Werte und Speicherstellen im Zwischencode

### l-Wert <=> r-Wert

**Adressberechnungen erfordern ein Konstrukt mit dem ein l-Wert (Adresse) in einen r-Wert transformiert wird. In C ist das der Adress-Operator &**

### r-Wert => l-Wert

**Soll eine berechneter / gespeicherter Wert als l-Wert genutzt werden, dann erfordert dies ein Konstrukt, das einen r-Wert in einen l-Wert transformiert. In C ist dies der Dereferenzierungs-Operator ***




### Konversionen: L-Wert (Speicherstelle) <~> R-Wert (Adresse) im Zwischencode:

```
//-----------------------------------------------
// Conversions between address and locations

// dereference the address found at an MAddressLoc: convert an r-value of type MAddress
// to an l-value of type MInt
case class DeRef(addrLoc: MAddressLoc) extends MIntLoc

// compute the address of a MIntLoc: convert an l-value of type MInt
// to an r-value of type MAddress
case class MkRef(mIntLoc: MIntLoc) extends MAddressLoc
```

***Die Zwischensprache kennt die l-Wert <~> r-Wert Konversionen:***

- **** : Adress-Wert ~> Speicherstelle***
- ***& : Speicherstelle ~> Adress-Wert***

# Zwischencode-Operationen – typisiert

## Typisierte Zuweisung – Int-Variante

```
// Binary operation: x := y op z on MInts
// dest location x is taken as l-value,
// operand1, y and operand2, z if it is a location, are taken as r-value
case class AssignInstr(
                        dest:     MIntLoc,
                        operand1: Option[MIntLoc],
                        op:       Option[MOp],
                        operand2: Option[MIntLocOrValue]
                      ) extends Instr

// facilitate generation of assign instructions
object AssignInstr {

  def apply(dest: MIntLoc, operand1: MIntLoc, op: MOp, operand2: MIntLocOrValue): AssignInstr =
    AssignInstr(dest, Some(operand1), Some(op), Some(operand2))

  def apply(dest: MIntLoc, operand1: MIntLoc): AssignInstr =
    AssignInstr(dest, Some(operand1), None, None)

  def apply(dest: MIntLoc, operand2: MIntImmediateValue): AssignInstr =
    AssignInstr(dest, None, None, Some(operand2))
}
```

```
sealed abstract class MOp
case object AddOp extends MOp
case object SubOp extends MOp
case object MultOp extends MOp
case object DivOp extends MOp

abstract class MRelOp
case object EqOp extends MRelOp
case object NeOp extends MRelOp
case object LsOp extends MRelOp
case object GtOp extends MRelOp
case object LeOp extends MRelOp
case object GeOp extends MRelOp
```

# Zwischencode-Operationen – typisiert

## Typisierte Zuweisung – Adress-Variante

```
// Assignment of address values
case class AssignAddrInstr(dest: MAddressLoc, source: MAddressLoc) extends Instr
```

***Bei Bedarf erweiterbar***

# Übersetzung von Zuweisungen

## Zuweisungen in der Quellsprache (AST)

```
case class Assign(var left: LocExp, right: Exp) extends Cmd
```

*Zuweisung: Die linke Seite repräsentiert eine Speicherstelle, die rechte einen Wert*

```
// Expression that denote (int) values
sealed abstract class Exp extends Positional {
  var staticType: Option[TypeInfo] = None // will be set by typifier
}
case class Number(d: Int)        extends Exp
case class Add(e1: Exp, e2: Exp) extends Exp
case class Sub(e1: Exp, e2: Exp) extends Exp
case class Div(e1: Exp, e2: Exp) extends Exp
case class Mul(e1: Exp, e2: Exp) extends Exp
case class LocAccess(var locExp: LocExp)   extends Exp
```

*r-Wert-Ausdrücke*

```
// Expressions that denote storage locations
sealed abstract class LocExp extends Positional {
  var staticType: Option[TypeInfo] = None // will be set by typifier
}
case class DirectLoc(symb: LocSymbol) extends LocExp // elementary storage location

// virtual dereferencing operation (*-operation)
case class StarConv(locExp: LocExp) extends LocExp
```

*l-Wert-Ausdrücke*

# Übersetzung von Zuweisungen

## Übersetzung von Zuweisungen

```
private def genCodeCmd(cmd: Cmd): Unit = cmd match {

  case Assign(left, right) =>
    val target = genCodeIntLocExp(left) // target location now contains the the destination loc
    genCodeValExp(right, target)        // generate code that puts value of right to target

  ...

}
```

*x* := *e*; 

***Erzeuge Code der die linke Seite x (zur Laufzeit) berechnet. Das Ergebnis ist eine Location (l-Value der Zwischensprache).***

`target` ***ist der zur Compile-Zeit bekannte Platz an dem die Ziel-Location zu finden ist.***

# Übersetzung von Zuweisungen

## Beispiele

```
a := a + b;
```

*a und b sind globale Variablen*

```
t_0 = INT[0, 0]
t_1 = INT[0, -1]
INT[0, 0] = t_0 + t_1
```

`INT[0, 0]` : *Speicherstelle vom (Maschinen-) Typ Int an Position*
- *Verschachtelungstiefe: 0*
- *Offset: 0*

`t_0` : *Hilfsvariable vom Typ Int*

```
PROC p(REF x: INT, y: INT)
BEGIN
    x := x + y;
END
```

```
a_0 = ADDR[1, 2]
a_1 = ADDR[1, 2]
t_0 = *[a_1]
t_1 = INT[1, 3]
*[a_0] = t_0 + t_1
```

`ADDR[1, 2]` : *Speicherstelle vom (Maschinen-) Typ Addr an Position*
- *Verschachtelungstiefe: 1*
- *Offset: 2*

`a_0` : *Hilfsvariable vom Typ Addr*

`*[a_1]` : *Dereferenzierungs-Operation: Speicherstelle auf die der Inhalt von zeigt;*
*Bei Auftreten auf der rechten Seite: plus implizierter Konversion l-Wert ~> r-Wert der Zwischencode-Instruktion: Nimm den Inhalt der Speicherstelle).*