Compilerbau CS1019

Th. Letschert

TH Mittelhessen Gießen

University of Applied Sciences

Compiler und Interpreter

- Übersetzen vs. Interpretieren
- Syntax / Semantik
- Quellsprache, Hochsprache / Zielsprache, Maschinensprache

Was ist ein Compiler

Compiler

- **Englisch**

  to compile **= zusammenstellen, sammeln, erstellen, aufarbeiten**

- **Informatik**

  Übersetzer von einer Programmiersprache in eine andere

  In der Regel von

  - **einer Hochsprache (C, Java, …)**

    also einer Sprache, deren Programme von Anwendungsentwicklern erstellt werden
  - **in eine Maschinen-nahe Sprache (Assembler, Maschinencode, …)**

    also einer Sprache, deren Programme auf einer Maschine ablauffähig sind

  Allgemein: Ein Compiler übersetzt

  - **Programme der Quellsprache / Quellprogramme in**
  - **Programme der Zielsprache / Zielprogramme**

Compiler – Themengebiete

Quellsprache / Zielsprache

Definition: Syntax und Semantik

„abstrakte“ sprachtheoretische Thematik mit Bezug zur Linguistik

Maschinen-nahe Zielsprache

Assembler / Maschinencode

Maschinen-Modell, Register, Adressierung, Speicherverwaltung, ...

Binder, Lader, Bibliotheken

„systemnahe“ Thematik mit Bezug zu Systemprogrammierung und Betriebssysteme

Code des Compilers

Compiler sind komplexe Software

Themen aus dem Bereich Algorithmen und Datenstrukturen sowie Software-Engineering

Mein erster Compiler

Mini-Compiler

Übersetzt einfache vollständig geklammerte arithmetische Ausdrücke

Quellsprache

Einfache vollständig geklammerte arithmetische Ausdrücke

Zielsprache

Maschinen-Sprache einer (hypothetischen) Mini-CPU

Quellsprache – Syntax

Eingabe – auch: Programm, Satz

- Die Eingabe für einen Compiler besteht aus Text (einem String)
- Nur bestimmte Texte sind erlaubte Eingaben
- Die Texte haben eine Struktur: ihre syntaktische Struktur
- Grammatik: Die Syntax, also die
  - erlaubten Texte und
  - ihre Struktur

  werden durch eine Grammatik definiert

Syntax der Quellsprache, definiert durch eine Grammatik:

Expression → *Number* | *Operation*
Number → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
Operation → (*Expression Operator Expression*)
Operator → + | – | * | /

} ***Produktionen***

Startsymbol: *Expression*
Nichtterminale: *Expression Number Operator Operation*
Terminale: 1 2 3 4 5 6 7 8 9 0 + – * / ( )

Beispiel

((2 * 5) + 3) korrekt
2 * 5 + 3 nicht korrekt

Syntax – Korrektheit

Korrektheit der Eingabe

- **Eine Grammatik beschreibt Produktionsregeln:**

  Wie *kann* ein korrekter Satz / ein korrektes Programm / eine korrekte Eingabe generiert werden
- **Jeder Text,**
  - **der durch die Anwendung der Produktionsregeln**
  - **ausgehend vom Startsymbol erzeugt werden kann**

  ist korrekt.

Beispiel

((2 * 5) + 3) korrekt.

Beweis durch Herleitung:

Expression
→ (*Expression Operator Expression*)
→ (*Expression* + *Expression*)
→ (*Expression* + *Number*)
→ (*Expression* + 3)
→ ((*Expression Operator Expression*) + 3)
→ ((*Expression* * *Expression*) + 3)
→ ((*Number* * *Expression*) + 3)
→ ((2 * *Expression*) + 3)
→ ((2 * *Number*) + 3)
→ ((2 * 5) + 3)

Ein Text / eine Eingabe ist ein (korrekter) Satz, wenn er mit der Grammatik produziert (hergeleitet) werden kann

Die Korrektheit wird durch die Angabe einer Ableitung bewiesen.

Syntax – Struktur

Struktur der Eingabe

- Die Anwendung der Produktionen definiert eine **Struktur** der Eingabe
- **Ableitungsbaum**: Darstellung der Anwendung der Produktionsregeln
- Die Produktion kann, muss aber nicht eindeutig sein

Beispiel

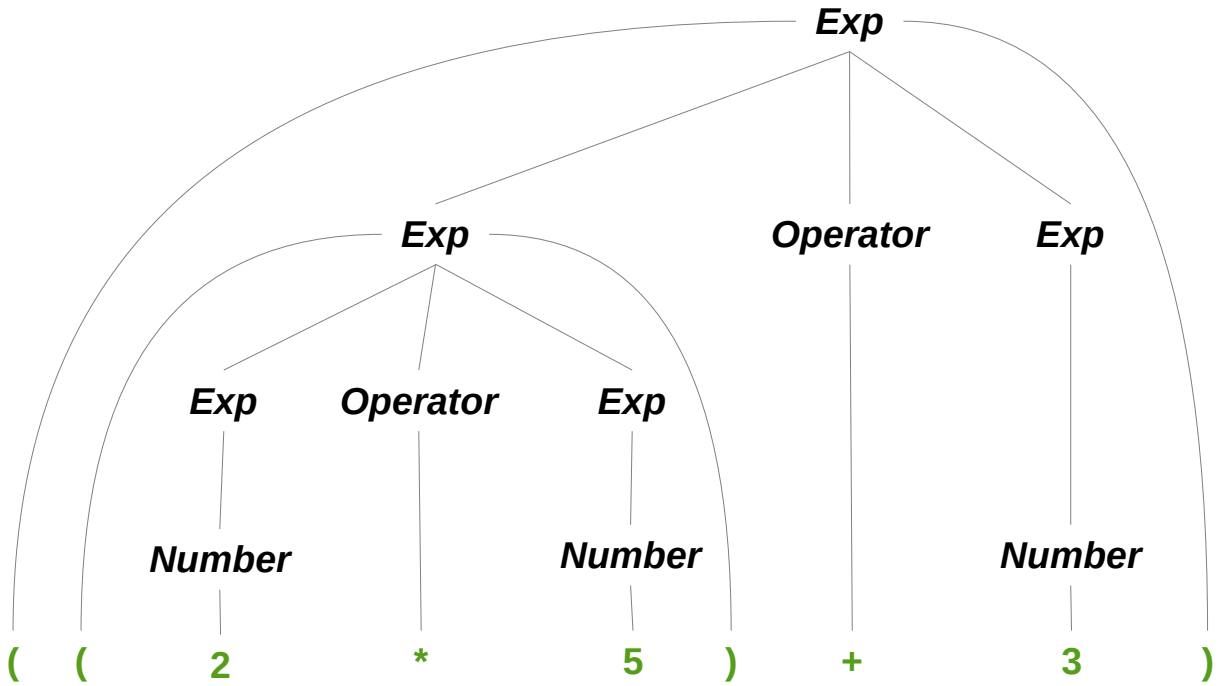
((2 * 5) + 3) ist ein korrekter Satz entsprechend dieser Syntax.

Beweis:

Exp
→ (*Exp Operator Exp*)
→ (*Exp* + *Exp*)
→ (*Exp* + *Number*)
→ (*Exp* + 3)
→ ((*Exp Operator Exp*) + 3)
→ ((*Exp* * *Exp*) + 3)
→ ((*Number* * *Exp*) + 3)
→ ((2 * *Exp*) + 3)
→ ((2 * *Number*) + 3)
→ ((2 * 5) + 3)

Ableitung

Expression hier abgekürzt zu Exp



Ableitungsbaum: graphische Darstellung der Ableitung

Syntax – Parsing / Syntaxanalyse

Parser

Programm-Modul mit den Aufgaben

- **Prüfung der Korrektheit der Eingabe :**

  Gibt es eine Folge von Produktionen, mit deren Anwendung der vorgelegte Text aus dem Startsymbol produziert werden kann ?

- **Feststellen der Struktur :**

  Gib die Produktionen an (etwa als Ableitungsbaum), mit denen der Text aus dem Startsymbol erzeugt werden kann.

Parser-Spezifikation

- **Eingabe: Text (String)**
- **Ausgabe: Ableitungsbaum der die syntaktische Struktur der Eingabe repräsentiert**

```
sealed abstract class ExpTree
case class Number(v: Int) extends ExpTree
case class Operation(exp1: ExpTree, op: Char, exp2: ExpTree) extends ExpTree
```

Definition von* ExpTree, *des Typs eines Ableitungsbaums:
Ein Ableitungsbaum ist entweder
– vom Typ* Number *und enthält dann eine ganze Zahl v oder er
– ist vom Typ* Operation *und enthält dann zwei* ExpTrees *exp1 und exp2 sowie einen Operator op

Syntax – Parsing / Syntaxanalyse

Parser-Funktion

Die rekursive Struktur der Eingabe führt fast automatisch zu einer rekursiven Parsing-Funktion.

Die Parsing-Funktion verarbeitet den Text ab einer bestimmten Position, bis zum Ende des dort beginnenden Ausdrucks.

Die Fehlerbehandlung und das Überlesen von Leerzeichen wurde der Einfachheit halber weg gelassen.

Exp → *Number* | *Operation*
Number → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
Operation → (*Exp Operator Exp*)
Operator → + | – | * | /

Die Grammatik und ihr Parser

```
object ExpParser {
  def parse(text: String): ExpTree = {
    var pos: Int = 0
    def parseFromPos: ExpTree = text(pos) match {
      case '1' => { pos = pos+1; Number(1) }
      case '2' => { pos = pos+1; Number(2) }
      case '3' => { pos = pos+1; Number(3) }
      case '4' => { pos = pos+1; Number(4) }
      case '5' => { pos = pos+1; Number(5) }
      case '6' => { pos = pos+1; Number(6) }
      case '7' => { pos = pos+1; Number(7) }
      case '8' => { pos = pos+1; Number(8) }
      case '9' => { pos = pos+1; Number(9) }
      case '0' => { pos = pos+1; Number(0) }
      case '(' => {
        pos = pos+1 // skip '('
        val exp1 = parseFromPos
        val op = text(pos)
        pos = pos+1 // skip opSign
        val exp2 = parseFromPos
        pos = pos+1 // skip ')'
        Operation(exp1, op, exp2)
      }
      case _ => throw new IllegalArgumentException
    }
    parseFromPos
  }
}
```

Parsing – Rekursiver Abstieg

Recursive Decent / Rekursiver Abstieg

Die vorgestellt Parsing-Funktion arbeitet nach dem Prinzip des rekursiven Abstiegs:

Der Parser verarbeitet die Eingabe von links nach rechts, Zeichen für Zeichen.

An Hand des „aktuellen“ Zeichens wird erkannt,

- **welche Produktion den Text erzeugt hat,**
- **der an dieser Position beginnt.**

Meist strukturiert man den rekursiven Abstieg folgendermaßen:

- **Eine Funktion pro Nonterminal mit**
- **einer Fallunterscheidung pro Produktion für dieses Nonterminal**

Parsing – Rekursiver Abstieg

Ordentlich strukturiert

```
object Parser {

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

    def parseExp: ExpTree = text(pos) match {
      ...
    }

    def parseOperation: Operation = {
      ...
    }

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

    parseExp
  }

}
```

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

```
def parseOperation: Operation = {
  pos = pos+1 // skip '('
  val exp1 = parseExp
  val opSymbol = text(pos)
  pos = pos+1 // skip opSign
  val exp2 = parseExp
  pos = pos+1 // skip ')'
  opSymbol match {
    case '+' => Operation(exp1, opSymbol, exp2)
    case '-' => Operation(exp1, opSymbol, exp2)
    case '*' => Operation(exp1, opSymbol, exp2)
    case '/' => Operation(exp1, opSymbol, exp2)
    case _ => throw new IllegalArgumentException
  }
}
```

```
def parseNumber: Number = text(pos) match {
  case '1' => { pos = pos+1; Number(1) }
  ...
  case '0' => { pos = pos+1; Number(0) }
  case _ => throw new IllegalArgumentException
}
```

Exp → *Number* | *Operation*
Number → 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0
Operation → (*Exp Operator Exp*)
Operator → + | – | * | /

Das geht alles kompakter, doch dazu kommen wir später.

Quellsprache – Semantik

Eingabe – auch: Programm, Satz

- Die Eingabe für einen Compiler besteht aus Text (einem String)
- Dieser Text bedeutet etwas
- Die Bedeutung nennt man Semantik
- Jeder syntaktisch korrekte Satz / jedes korrekte Programm hat eine Bedeutung
- Arithmetische Ausdrücke „bedeuten“ Zahlen

Beispiel

`( ( 2 * 5 ) + 3 )` ist korrekt und bedeutet 13

Semantik – rekursiv definiert

Die Bedeutung eines Textes (Satz, Programm, …) ist meist rekursiv über dessen Struktur definiert.

Beispiel

((2 * 5) + 3) ist korrekt und bedeutet 13

denn

((2 * 5) + 3) ist entstanden aus (2 * 5), +, und 3

und (2 * 5) bedeutet 10,

3 bedeutet 3,

+ bedeutet + (addiere)

Semantik – Interpreter / 1 mit Baumaufbau

Ein Interpreter bestimmt / berechnet die Semantik eines Satzes / Programms direkt

```
object Interpreter {

  def interpret(text: String): Int = {
    val tree = parse(text)
    eval(tree)
  }

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


  def parse(text: String): ExpTree = {
    . . .
  }

  def main(args: Array[String]): Unit = {
    val expression = scala.io.StdIn.readLine()
    println(interpret(expression))
  }

}
```

Der „Programm“-Text wird analysiert, seine Struktur in einem Ableitungsbaum dokumentiert und dieser dann ausgewertet.

Semantik – Interpreter / 2 ohne Baumaufbau

Ein Interpreter kann auch ohne den Zwischenschritt der Baumkonstruktion arbeiten

Er wertet das Programm während der Analyse des Textes „im Fluge“ aus.

```
object Interpreter {

  def interpret(text: String): Int = eval(text)

  def eval(text: String): Int = {

    var pos: Int = 0

    def evalExp: Int = text(pos) match {
      . . .
    }

    def evalOperation: Int = {
        . . .
    }

    evalExp
  }

}
```

```
    def evalExp: Int = text(pos) match {
      case '1' => 1
      case '2' => 2
      . . .
      case '0' => 0
      case '(' => evalOperation
      case _ => throw new IllegalArgumentException
    }
```

```
    def evalOperation: Int = {
        pos = pos+1 // skip '('
        val exp1 = evalExp
        val opSymbol = text(pos)
        pos = pos+1 // skip opSign
        val exp2 = evalExp
        pos = pos+1 // skip ')'
        opSymbol match {
          case '+' => exp1 + exp2
          case '-' => exp1 - exp2
          case '*' => exp1 * exp2
          case '/' => exp1 / exp2
          case _ => throw new IllegalArgumentException
        }
    }
```

Zielsprache

Zielsprache ist eine Maschinensprache

Maschinensprache

- Verarbeitet Anweisungen
- Durch Hardware-Interpreter
  Anweisungen werden durch Hardware interpretiert,
  oder *könnten* durch Hardware interpretiert werden
- Anweisungen modifizieren den aktuellen Zustand der Machine

Hardware- vs Software-Interpreter

- Software-Interpreter
  beliebige Funktionalität, die durch ein Programm realisiert werden kann
- Hardware-Interpreter: Zustands-Maschine
  Funktionalität, die durch Hardware realisiert werden kann

  Maschinen-Zustand x Anweisung => neuer Maschinen-Zustand

Hardware vs Software

Rekursion in Hardware nicht möglich (Maschinen können sich (noch) nicht klonen)
Rekursion kann nur simuliert werden

Virtuellen Maschinen

Übersetzung in 2 Phasen

- **Phase I (Frontend des Compilers):**
  Übersetzung in Zwischendarstellung
  Diese Phase ist nur abhängig von der Quellsprache
- **Phase II (Backend des Compilers):**
  Übersetzung der Zwischendarstellung in Maschinencode
  Diese Phase ist nur abhängig vom Zielsystem

Idee der virtuellen Maschine

Die Zwischendarstellung ist der Code einer fiktiven (virtuellen) Maschine

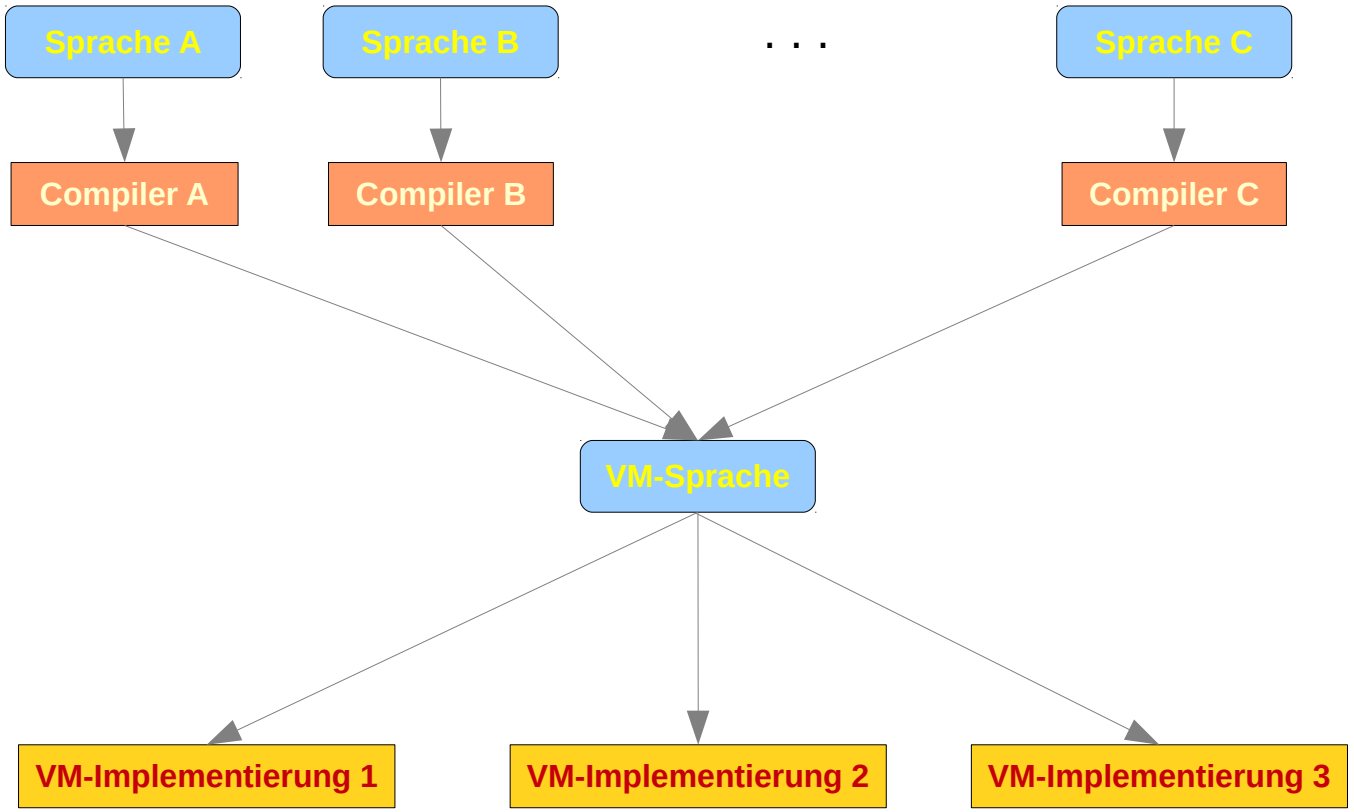
Vorteil:

- **Ein Frontend für diverse Zielsysteme**
- **Ein Backend für diverse Quellsprachen**
- **Programme unterschiedlicher Quellsprachen kooperieren auf einfache Art**
- **Fortschritte in der Hardwaretechnik leichter zugänglich für Quellsprachen**

Historie

- **P-Code für Pascal-Compiler (1970er)**
- **VM für Java (1990er)**
- **CLR für .NET (200er)**
- **….**

Konzept der virtuellen Maschinen



Implementierung von virtuellen Maschinen

Emulation

Die virtuelle Maschine wird durch ein Programm emuliert

Die Befehle der VM-Sprache werden durch ein Emulator-Programm ausgeführt

Übersetzung

Die Befehle der VM-Sprache werden in den Code einer realen Maschine übersetzt

JIT-Compiler

Mischform von Emulation und Übersetzung

Erzeugung von „echtem“ Maschinencode während der Laufzeit des Programms

Zielsystem: Stack- und Register-Maschinen

Maschinentypen

Maschinen, d.h. CPUs, (real oder virtuell) können über unterschiedliche interne Speicherplätze verfügen

Register-Maschinen

verfügen über einen mehr oder weniger großen Satz an Registern

Register: interner Speicherplatz der direkt über einen Namen angesprochen werden kann

Reale Maschinen sind in der Regel Register-Maschinen

Stack-Maschinen

verfügen über interne Speicherplätze die als Stack organisiert sind und nur über Stack-Operationen angesprochen werden können.

Stack-Maschinen sind in der Handhabung etwas einfacher als Register-Maschinen

Zielsystem – Stack-Maschine

Zielsprache ist die Maschinensprache einer Stack-Maschine

Stack-Maschine

- Hat einen Stack
- **Kann Werte auf dem Stack durch arithmetische Operationen verknüpfen**

Zustand der Maschine

- Aktuelle Werte auf dem Stack

Anweisungen

- PUSHC x — *lege x auf den Stack (x ist eine ganze einstellige Zahl)*
- ADD — *Entferne die beiden obersten Werte vom Stack, Addiere sie und lege das Ergebnis auf dem Stack ab*
- SUB — *Entsprechend*
- MULT — *Entsprechend*
- DIV — *Entsprechend*
- RDINT — *Lies einen Int-Wert*
- WRInt — *Schreibe einen Int-Wert*
- HALT — *Stoppe die Maschine*

Zielsprache – Syntax

Zielsprache ist die Maschinensprache einer Stack-Maschine

Maschinen-Sprache

- **Wird durch Hardware analysiert**
- **Hardware versteht nur Bits und Bytes**
- **Syntax der Zielsprache muss auf Byte-Ebene definiert werden**

Instruktionen für die Stack-Maschine:

```
abstract class Instruction(val opCode: Byte, val arg: Int)

case object Halt          extends Instruction(0, 0)
case class  Pushc(v: Int) extends Instruction(1, v)
case object Add           extends Instruction(2, 0)
case object Sub           extends Instruction(3, 0)
case object Mult          extends Instruction(4, 0)
case object Div           extends Instruction(5, 0)
case object Mod           extends Instruction(6, 0)
case object Rdint         extends Instruction(7, 0)
case object Wrint         extends Instruction(8, 0)
case object Noop          extends Instruction(-1, 0)
```

Jede Instruktion besteht aus einem
– Op-Code: Was ist zu tun und einem
– Immediate-Wert: einem optionalen Argument der Operation

Ziel-Maschine

Speicher: Stack und Programmspeicher

```
val stackSize = 10
val memSize   = 100
val stack : Array[Int] = new Array(stackSize)
var stackTop: Int = -1
val program_memory: Array[Instruction] = Array.tabulate(memSize)(i => Noop)
```

Steuerwerk: Instruktionen laden und ausführen

```
var IR : Instruction = Noop
var running = false

def load(prog: List[Instruction]) : Unit = {
  var i = 0
  prog.foreach {
    instr => program_memory(i) = instr
    i = i+1
  }
}

def run: Unit = {
  var PC = 0
  running = true
  while (PC < memSize && running) {
    IR = program_memory(PC)
    PC = PC+1
    exec(IR)
  }
}
```

Ziel-Maschine

Eine Instruktion ausführen

```
def exec(instruction: Instruction) : Unit = instruction match {
  case Pushc(v: Int) =>
    stackTop = stackTop +1
    stack(stackTop) = v
  case Add =>
    val v1 = stack(stackTop); stackTop = stackTop-1
    val v2 = stack(stackTop); stackTop = stackTop-1
    stackTop = stackTop+1
    stack(stackTop) = v2 + v1
  case Sub =>
    val v1 = stack(stackTop); stackTop = stackTop-1
    val v2 = stack(stackTop); stackTop = stackTop-1
    stackTop = stackTop+1
    stack(stackTop) = v2 - v1
  case Mult =>
    val v1 = stack(stackTop); stackTop = stackTop-1
    val v2 = stack(stackTop); stackTop = stackTop-1
    stackTop = stackTop+1
    stack(stackTop) = v2 * v1
  case Div =>
    val v1 = stack(stackTop); stackTop = stackTop-1
    val v2 = stack(stackTop); stackTop = stackTop-1
    stackTop = stackTop+1
    stack(stackTop) = v2 / v1
  case Halt =>
    running = false
  case Rdint =>
    val v = scala.io.StdIn.readInt()
    stackTop = stackTop +1
    stack(stackTop) = v
  case Wrint =>
    val v = stack(stackTop)
    stackTop = stackTop-1
    println(v)
  case Noop =>
}
```

Ziel-Maschine

Zurück setzen

```
def reset: Unit = {
  stackTop = -1
  running = false
  IR = Noop
  (0 until memSize) foreach( i => program_memory(i) = Noop )
}
```

Compiler

Aufgabe

Übersetzen eines Programms der Quellsprache

In ein äquivalentes Programm der Zielsprache

äquivalente Programme

Ein Programm *p* der Quellsprache ist äquivalent

zu einem Programm *p'* der Zielsprache

wenn

- Der Interpreter für *p* und
- Eine Ausführung von *p'* durch die Maschine

bei gleicher Eingabe das gleiche Ergebnis liefern

Compiler – Parser

```
obect Parser {

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

    def parseExp: ExpTree = text(pos) match {
      case '1' => parseNumber
      case '2' => parseNumber
      . . .
      case '0' => parseNumber
      case '(' => paserOperation
      case _ => throw new IllegalArgumentException
    }


    def parseNumber: Number = text(pos) match {
      case '1' => { pos = pos+1; Number(1) }
      case '2' => { pos = pos+1; Number(2) }
      . . .
      case '9' => { pos = pos+1; Number(9) }
      case '0' => { pos = pos+1; Number(0) }
      case _ => throw new IllegalArgumentException
    }

    def paserOperation: Operation = { . . . }

    parseExp
  }

}
```

Ein Compiler benötigt einen Parser der den Quelltext analysiert und den Baum erstellt.
Der Parser des Interpreters kann wiederverwendet werden.

```
def paserOperation: Operation = {
  pos = pos+1 // skip '('
  val exp1 = parseExp
  val opSymbol = text(pos)
  pos = pos+1 // skip opSign
  val exp2 = parseExp
  pos = pos+1 // skip ')'
  opSymbol match {
    case '+' => Operation(exp1, Operator(opSymbol), exp2)
    case '-' => Operation(exp1, Operator(opSymbol), exp2)
    case '*' => Operation(exp1, Operator(opSymbol), exp2)
    case '/' => Operation(exp1, Operator(opSymbol), exp2)
    case _ => throw new IllegalArgumentException
  }
}
```

Compiler – Codegenerator

Aus dem vom Parser erzeugten Baum wird Maschinencode generiert

```
object Codegenerator {

  def genCode(tree: ExpTree): List[Instruction] = tree match {
    case Number(x) => List(Pushc(x))
    case Operation(exp1, op, exp2) =>
      val l1 = genCode(exp1)
      val l2 = genCode(exp2)
      l1 ::: l2 ::: (op match {
        case '+' => List(Add)
        case '-' => List(Sub)
        case '*' => List(Mult)
        case '/' => List(Div)
      })
  }

}
```

Compiler – Parser + Codegenerator

```
object Compiler {

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

}
```

erzeugt Maschinencode für die Stackmaschine

IDE – Compiler + Maschine + Benutzerschnittstelle

```
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
  }

}
```

erzeugt Maschinencode für die Stackmaschine und lässt ihn durch die Stackmaschine ausführen

Compilerbau – Themengebiete

Höhere Programmiersprachen

Syntax Grammatik: Definition von Textstrukturen
Parser: Analyse von Texten / Korrektheit / Struktur

Semantik Programmiersprachliche Konstrukte und ihre Bedeutung

Maschinennahe Programmiersprachen

Maschinen und Maschinenprogramme

Ausführung von Maschinenprogrammen

Binder: Zusammenbau von Maschinenprogrammen aus Komponenten

Lader: Maschinenprogramme zur Ausführung bringen

Datenstrukturen und Algorithmen

Konstruktion von Bäumen

Algorithmen auf Bäumen

Codegenerierung

Compilerbau – Übersicht

Phasen

Die Übersetzung eines Programms erfolgt (real, gelegentlich nur konzeptionell) in Phasen:

- **Phase I / Frontend des Compilers: Analyse**
  Das Quellprogramm wird analysiert und in Zwischendarstellung transformiert
  Z.B. Bytecode bei Java-Compilern
  Diese Phase hängt im Wesentlichen nur von der Quellsprache ab
- **Phase II / Backend des Compilers: Synthese**
  Aus der Zwischendarstellung wird Maschinencode erzeugt
  Die Phase hängt im Wesentlichen nur von der Maschine und ihrer Sprache ab

Compilerbau – Phasen

Die beiden Hauptphasen werden meist weiter unterteilt

Phase I / Frontend des Compilers: Analyse

- **Syntaxanalyse**

  Prüfung der syntaktischen Korrektheit des Programms

  feststellen seiner syntaktischen Struktur mit den Unterphase:

  - **Lexikalische Analyse**

    Aus welchen Tokens („Worten“) ist das Programm aufgebaut
  - **Syntaxanalyse**

    Welche Strukturen („Sätze“) bilden die Tokens.

- **Semantische Analyse**

  Analyse der Inhalte eines Programms, die nicht syntaktisch ausgedrückt werden können:

  - **Definition und Verwendung von Konstanten, Variablen, Typen**
  - **Korrektheit und Konsistenz der Typen in Ausdrücken**
  - **etc.**
- **Optimierung**

  Die Aktionen des Quellprogramms können oft optimiert werden um so die Laufzeit und Speichereffizienz zu verbessern.

Compilerbau – Phasen

Phase II / Backend des Compilers: Synthese

Die Phaseneinteilung des Backends ist weniger standardisiert als die des Frontends. Folgende Hauptphasen sind meist zu finden:

- **Assembler**

  Der Compiler kann als Zwischendarstellung des Quellprogramms Code einer Assemblersprache generieren.

  Assemblercode: Maschinennaher symbolischer Code

  Assembler: Übersetzt symbolischen Code in verschiebbaren Maschinencode

  Verschiebbarer Maschinencode: Maschinencode mit symbolischen Adressen kann an beliebige „echte Adressen geschoben werden“.

- **VM-Code**

  Alternativ zum Assembler-Code kann auch Code einer virtuellen Maschine erzeugt werden.

- **Binder und Lader**

  Der Binder (*Linker*) macht aus verschiebbarem Maschinencode des Programms und Bibliotheksfunktionen Maschinencode der vom Lader geladen und dann ausgeführt werden kann.

Literaturhinweise

- Michael Jäger: *Compilerbau – eine Einführung SS 2015*
  https://homepages.thm.de/~hg52/lv/compiler/skripten/compilerskript/pdf/compilerskript.pdf
- H. Geisse, Boris Budweg: *Compilerbau*,
  https://homepages.thm.de/~hg53/cb-ws1415/Compilerbau_v1.2.pdf
- K.D. Cooper, L. Torczon: *Engineering a Compiler*,
  2te Auflage, Morgan Kaufmann 2012
- M.L. Scott: *Programming Language Pragmatics*
  Academic Press 2000