

# Inhaltsverzeichnis

<b>1 Fragen</b>	<b>2</b>
1.1 Design by Contract . . . . .	2
1.1.1 JML . . . . .	2
1.2 Compiler . . . . .	2
1.2.1 First und Follow Mengen . . . . .	2
1.2.2 Java-Bytecode . . . . .	2
<b>2 Theoretische Grundlagen</b>	<b>3</b>
2.1 Äquivalenz . . . . .	3
2.1.1 $\alpha$ -Äquivalenz (Folie: 20_(166)) . . . . .	3
2.1.2 $\eta$ -Äquivalenz (Folie: 20_(167)) . . . . .	3
2.1.3 Church-Zahlen (Folie: 20_(179ff)) . . . . .	3
2.1.4 Rekursionsoperator Y (Folie: 20_(184)) . . . . .	4
2.2 Typinferenz . . . . .	4
<b>3 Prolog</b>	<b>5</b>
<b>4 Parallelprogrammierung</b>	<b>6</b>
4.1 Grundlagen . . . . .	6
4.1.1 Coffman-Bedingungen (Deadlock-Bedingungen) (Folie: 54_(47)) . . . . .	6
4.1.2 Flynn's Taxonomy (Folie: 51_(13)) (Grafiken von SS 14, Nr. 6) . . . . .	6
4.1.3 Beschleunigung: Amdahl's Law (Folie: 51_(17)) . . . . .	7
4.2 MPI . . . . .	7
4.3 Java . . . . .	8
4.3.1 Runnable . . . . .	8
4.3.2 Callable + Feature . . . . .	9
4.3.3 RecursiveAction / ForkJoinPool . . . . .	10
4.3.4 RecursiveTask / ForkJoinPool . . . . .	11
4.3.5 AkkaActor tell() . . . . .	12
4.3.6 AkkaActor ask() . . . . .	13
<b>5 Design by Contract</b>	<b>14</b>
5.1 JML . . . . .	14
5.1.1 Basic Syntax . . . . .	14
<b>6 Compiler</b>	<b>15</b>
6.1 Java-Bytecode . . . . .	15
6.1.1 Präfixe / Suffixe . . . . .	15
6.1.2 Lesen / Schreiben von lokalen Variablen . . . . .	15
6.1.3 Lesen / Schreiben von Feldern . . . . .	15
6.1.4 Sprungbefehle . . . . .	15
6.1.5 Methodenaufrufe . . . . .	15
6.1.6 Objekterzeugung . . . . .	15
6.1.7 Arithmetische Berechnungen . . . . .	15

# Kapitel 1

## Fragen

### 1.1 Design by Contract

#### 1.1.1 JML

- Objekte non-Null Default: Reference
- muss die Precondition Availability rule (Folie: 60\_(25)) erfüllt / eingehalten werden? Oder nur Best Practice?

### 1.2 Compiler

#### 1.2.1 First und Follow Mengen

- $Follow_k(x)$  auch wie folgt definierbar?:

$$\begin{aligned} Follow_k(x) &= \{\cup First_k(y) | \exists m, y \in (V \cup \Sigma)^* : S \Rightarrow^* mxy\} \\ &= \{u \in \Sigma^k | \exists m, y \in (V \cup \Sigma)^* : \left( S \Rightarrow^* mxy \right) \wedge \left( u \in First_k(y) \right)\} \text{ (große Klammern hinzugefügt)} \\ &= \{u \in \Sigma^k | \exists m, y \in (V \cup \Sigma)^* : S \Rightarrow^* mxy \wedge u \in First_k(y)\} \text{ (Folie: 71_(379))} \end{aligned}$$

- wie ist  $Follow_k(x)$  definiert, wenn

$$S \not\Rightarrow mxy$$

d.h. wenn „x“ nie von S aus abgebildet wird? (bspw.  $Follow_k(S)$ , d.h. S ist nur Startzustand, wird aber durch keine Produktion / Abbildung erreicht?)

**Annahme:**  $Follow_k(x) = \{\#\}$

- gilt  $\# \in Follow_k(S)$  immer?

#### 1.2.2 Java-Bytecode

- was passiert wenn Typ der Stackvariablen nicht mit dem erwarteten Operationstyp übereinstimmt?  
z.B: oberste Variable auf Stack ist „3.5“ (Double) und istore\_3 wird ausgeführt  
 $\Rightarrow$  Exception?

# Kapitel 2

## Theoretische Grundlagen

### 2.1 Äquivalenz

#### 2.1.1 $\alpha$ -Äquivalenz (Folie: 20\_(166))

gleicher Ausdruck / Funktion, nur andere Namen  $\Rightarrow$  durch Umbenennung Transformation von  $t_1$  zu  $t_2$  möglich

#### 2.1.2 $\eta$ -Äquivalenz (Folie: 20\_(167))

Zwei Funktionen sind gleich, falls Ergebnis gleich für alle Argumente

#### 2.1.3 Church-Zahlen (Folie: 20\_(179ff))

- Zahlen:

$$\begin{aligned}c_0 &= \lambda s. \lambda z. z \\c_1 &= \lambda s. \lambda z. s z \\c_2 &= \lambda s. \lambda z. s (s z) \\&\vdots \\c_n &= \lambda s. \lambda z. s^n z\end{aligned}$$

- Operationen

- Nachfolgerfunktion  $succ = \lambda n. \lambda s. \lambda z. s (n s z)$
- Addition  $plus = \lambda m. \lambda n. \lambda s. \lambda z. m s (n s z)$
- Multiplikation  $times = \lambda m. \lambda n. \lambda s. n (m s) \stackrel{\eta}{=} \lambda m. \lambda n. \lambda s. \lambda z. n (m s) z$
- Potenzieren  $exp = \lambda m. \lambda n. n m \stackrel{\eta}{=} \lambda m. \lambda n. \lambda s. \lambda z. n m s z$
- Subtraktion  $sub = \lambda m. \lambda n. n pred m$  (ÜB 5, Nr. 4)
  - \*  $pair = \lambda a. \lambda b. \lambda f. f a b$
  - \*  $fst = \lambda p. p(\lambda a. \lambda b. a)$
  - \*  $snd = \lambda p. p(\lambda a. \lambda b. b)$
  - \*  $next = \lambda p. pair (snd p) (succ (snd p))$
  - \*  $pred = \lambda n. fst (n next (pair c_0 c_0))$
- Vergleichsoperation „lessEq  $m \leq n$ “  $lessEq = \lambda m. \lambda n. isZero (sub m n)$  (ÜB 6, Nr. 2)
- Vergleichsoperation „greaterEq  $m \geq n$ “  $greaterEq = \lambda m. \lambda n. isZero (sub n m)$
- Vergleichsoperation „eq  $m == n$ “  $eq = \lambda m. \lambda n. (\lambda a. a) (lessEq m n) (greaterEq m n) c_{false}$
- $isZero = \lambda n. n (\lambda p. c_{false}) c_{true}$  (SS 14)

- boolesche Werte

- True  $c_{true} = \lambda t. \lambda f. t$
- False  $c_{false} = \lambda t. \lambda f. f$

### 2.1.4 Rekursionsoperator Y (Folie: 20\_(184))

$$Y = \lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$$

- Rekursionsoperator Y ist nicht typisierbar (Folie: 21\_(205))

## 2.2 Typinferenz

Bei Let-Polymorphismus angepasste Regeln von VAR und ABS (Folie: 22\_(211))

Regel	Constraint	Bsp.
$\text{CONST} \frac{c \in \text{Const}}{\Gamma \vdash c : \tau_c}$	$\text{CONST} \frac{c \in \text{Const}}{\Gamma \vdash c : \alpha_1} : \alpha_1 = \text{Typ}(c)$	$\text{CONST} \frac{1 \in \text{Const}}{\Gamma \vdash 1 : \alpha_1} : \alpha_1 = \text{Typ}(1) = \text{Int}$
$\text{VAR} \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$	$\text{VAR} \frac{(x : \alpha_1)(x) = \alpha_2}{(x : \alpha_1) \vdash x : \alpha_2} : \alpha_1 = \alpha_2$	
$\text{ABS} \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \rightarrow \tau_2}$	$\text{ABS} \frac{\Gamma, x : \alpha_2 \vdash t : \alpha_3}{\Gamma \vdash \lambda x.t : \alpha_1} : \alpha_1 = \alpha_2 \rightarrow \alpha_3$	
$\text{APP} \frac{\Gamma \vdash t_1 : \tau_2 \rightarrow \tau \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1\ t_2 : \tau}$	$\text{APP} \frac{\Gamma \vdash t_1 : \alpha_2 \quad \Gamma \vdash t_2 : \alpha_3}{\Gamma \vdash t_1\ t_2 : \alpha_1} : \alpha_2 = \alpha_3 \rightarrow \alpha_1$	
$\text{LET} \frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma, x : ta(\tau_1, \Gamma) \vdash t_2 : \tau_2}{\Gamma \vdash \text{let } x=t_1 \text{ in } t_2 : \tau_2}$	$\text{LET} \frac{\Gamma \vdash t_1 : \alpha_2 \quad \Gamma, x : ta(\alpha_2, \Gamma) \vdash t_2 : \alpha_3}{\Gamma \vdash \text{let } x=t_1 \text{ in } t_2 : \alpha_1} : \alpha_2 = \alpha_3$	

# Kapitel 3

## Prolog

- Vergleich arithmetischer Ausdrücke
  - gleich: „ $=$ “
  - ungleich: „ $\neq$ “
  - kleiner: „ $<$ “
  - kleiner-gleich : „ $\leq$ “
  - größer: „ $>$ “
  - größer-gleich: „ $\geq$ “

# Kapitel 4

## Parallelprogrammierung

### 4.1 Grundlagen

#### 4.1.1 Coffman-Bedingungen (Deadlock-Bedingungen) (Folie: 54\_(47))

Wenn alle vier der folgenden Bedingungen zutreffen, liegt ein Deadlock vor. Deadlocks können verhindert werden, indem immer mindestens eine Bedingung nicht erfüllt ist, d.h. nicht alle auf einmal erfüllt sein können.

1. **Mutual exclusion**
  - beschränkter Zugriff auf eine Ressource
  - Ressource kann nur mit einer beschränkten Anzahl von Nutzern geteilt werden
2. **Hold and wait**: Warten auf alle benötigten Ressourcen, während die Kontrolle über bisher zugesprochene (mind. eine) Ressourcen behalten wird.
3. **No preemption**: Zugewiesene Ressourcen können nur freiwillig zurückgegeben werden, die Rückgabe kann nicht erzwungen werden.
4. **Circular Wait**: Möglichkeit von Kreisen in Ressourcen-Anfragen Graph:  
Zyklische Kette von Prozessen, die bereits Ressourcen (mind. eine) erhalten haben und gleichzeitig auf weitere Ressourcen warten, welche jeweils dem nächsten Prozess in der zirkulären Kette zugesprochen wurden.

#### 4.1.2 Flynn's Taxonomy (Folie: 51\_(13)) (Grafiken von SS 14, Nr. 6)

Taxonomie		Graphische Repräsentation
SISD (Single Instruction x Single Data)	von Neumann Architektur: eine Anweisungsreihenfolge auf einem Speicher ausgeführt	
SIMD (Single Instruction x Multiple Data)	Eine Anweisung auf homogene Daten angewendet (z.B. Array) z.B. Vektor Prozessoren von frühen Supercomputern	
MIMD (Multiple Instruction x Multiple Data)	versch. Prozessoren operieren auf versch. Daten z.B. aktuelle Mehrkernprozessoren	
MISD (Multiple Instruction x Single Data)	Mehrere Anweisungen gleichzeitig auf gleichen Daten ausgeführt z.B. redundante Architekturen oder Pipelines in modernen Prozessoren	

#### 4.1.3 Beschleunigung: Amdahl's Law (Folie: 51\_(17))

Beschleunigung eines Algo durch die Verwendung von n Prozessoren:

$$S(n) = \frac{T(1)}{T(n)} = \frac{\text{Ausführungszeit mit einem Prozessor}}{\text{Ausführungszeit mit n Prozessoren}}$$

maximale Beschleunigung durch parallele Ausführung mit n Prozessoren:

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}}$$

p: parallelisierbarer prozentualer Anteil des Programms

## 4.2 MPI

- `MPI_Comm_WORLD`: Standard Communicator
- `MPI_Init(&argc, &args)`: initialisiere MPI
- `MPI_Finalize()`: Clean-Up nach Ausführung von MPI
- `MPI_Comm_rank(MPI_Comm_WORLD, &my_rank)`: `my_rank` enthält den Rank des aktuellen Prozesses
- `MPI_Comm_size(MPI_Comm_WORLD, &size)`: `size` enthält Gesamtanzahl von Prozessen
- `MPI_Barrier(MPI_Comm_WORLD)`: Barriere, blockiert bis alle Prozesse die Barriere aufgerufen haben
- `MPI_Send(void* buffer, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
- `MPI_Recv(void* buffer, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status* status)`

## 4.3 Java

### 4.3.1 Runnable

```
public class DemoRunnable {
    public static void main(String[] args) {
        Thread thread0 = new Thread(new MyRunnable(0));
        Thread thread1 = new Thread(new MyRunnable(1));

        thread0.start();
        thread1.start();
    }

    public static class MyRunnable implements Runnable {
        private int id;

        public MyRunnable(int id) {
            this.id = id;
        }

        public void run() {
            System.out.println("Running: " + this.id);
        }
    }
}

/* Ausgabe:
Running: 0
Running: 1
*/
```



### 4.3.2 Callable + Feature

```

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.Callable;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;

public class DemoCallableFuture {
    public static void main(String[] args) {
        ExecutorService executorService = Executors.newFixedThreadPool(4);
        List<Future<String>> futures = new ArrayList<Future<String>>();

        for (int i = 0; i < 2; i++) {
            futures.add(executorService.submit(new MyCallable(i)));
        }

        for (Future<String> future : futures) {
            try {
                String result = future.get();
                System.out.println(result);
            } catch (InterruptedException e) {
                //TODO: handle exception
            } catch (ExecutionException e){
                //TODO: handle exception
            }
        }

        executorService.shutdown();
    }

    public static class MyCallable implements Callable<String> {
        private int id;

        public MyCallable(int id) {
            this.id = id;
        }

        public String call() {
            return "Running:␣" + this.id;
        }
    }
}

/* Ausgabe:
Running: 0
Running: 1
*/

```

### 4.3.3 RecursiveAction / ForkJoinPool

```

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveAction;

public class DemoRecursiveAction {
    public static void main(String[] args) {
        MyRecursiveAction action = new MyRecursiveAction(4, "none");
        ForkJoinPool pool = new ForkJoinPool();
        pool.invoke(action);
    }

    public static class MyRecursiveAction extends RecursiveAction {
        private static final long serialVersionUID = 1L;
        private int workload;
        private String side;

        public MyRecursiveAction(int workload, String side) {
            this.workload = workload;
            this.side = side;
        }

        @Override
        protected void compute() {
            if (this.workload > 1) {
                System.out.println("Workload splitted (" + this.side + "): " + this.workload);

                int half = (int) this.workload / 2;
                MyRecursiveAction left = new MyRecursiveAction(half, "left");
                MyRecursiveAction right = new MyRecursiveAction(half, "right");

                left.fork();
                right.compute();
                left.join();
            } else {
                System.out.println("Workload not splitted (" + this.side + "): " + this.workload);
            }
        }
    }
}

/* Ausgabe:
Workload splitted (none): 4
Workload splitted (right): 2
Workload not splitted (right): 1
Workload not splitted (left): 1
Workload splitted (left): 2
Workload not splitted (right): 1
Workload not splitted (left): 1
*/

```

## 4.3.4 RecursiveTask / ForkJoinPool

```

import java.util.ArrayList;
import java.util.List;
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.RecursiveTask;

public class DemoRecursiveTask {
    public static void main(String[] args) {
        MyRecursiveTask task = new MyRecursiveTask(4, "None");
        ForkJoinPool pool = new ForkJoinPool();
        Long result = pool.invoke(task);

        System.out.println("Final result: " + result);
    }

    public static class MyRecursiveTask extends RecursiveTask<Long> {
        private static final long serialVersionUID = 1L;
        private long workload;
        private String side;

        public MyRecursiveTask(long workload, String side) {
            this.workload = workload;
            this.side = side;
        }

        @Override
        protected Long compute() {
            if (this.workload > 1) {
                int half = (int) this.workload / 2;
                MyRecursiveTask left = new MyRecursiveTask(half, "Left");
                MyRecursiveTask right = new MyRecursiveTask(half, "Right");

                left.fork();
                Long rightResult = right.compute();
                Long leftResult = left.join();
                Long result = leftResult + rightResult;
                System.out.println(this.side + " result (splitted): " + result);
                return result;
            } else {
                System.out.println(this.side + " result (not splitted): " + this.workload);
                return this.workload;
            }
        }
    }
}

/* Ausgabe:
Right result (not splitted): 1
Left result (not splitted): 1
Right result (splitted): 2
Right result (not splitted): 1
Left result (not splitted): 1
Left result (splitted): 2
None result (splitted): 4
Final result: 4
*/

```

### 4.3.5 AkkaActor tell()

```

package DemoAkkaActor;

import akka.actor.*;

public class DemoAkkaActorTell {
    public static void main(String[] args) {
        ActorSystem actorSystem = ActorSystem.create("Demo");
        ActorRef zero = actorSystem.actorOf(Props.create(MyActor.class, 0));
        ActorRef one = actorSystem.actorOf(Props.create(MyActor.class, 1));

        zero.tell("Hello", one);
        one.tell("Hello", zero);
        actorSystem.terminate();
    }

    public static class MyActor extends AbstractActor {
        private int id;

        public MyActor(int id) {
            this.id = id;
        }

        @Override
        public Receive createReceive() {
            return receiveBuilder()
                .match(String.class, this::handleStringMessage)
                .matchAny(message -> unhandled(message))
                .build();
        }

        private void handleStringMessage(String message) {
            System.out.println("Id:␣" + this.id + "␣message:␣" + message);
            getSender().tell("Hi", ActorRef.noSender());
        }
    }
}

/* Ausgabe
Id: 1 message: Hello
Id: 0 message: Hello
Id: 0 message: Hi
Id: 1 message: Hi
*/

```

## 4.3.6 AkkaActor ask()

```

package DemoAkkaActor;

import akka.actor.*;
import akka.pattern.Patterns;
import akka.util.Timeout;
import scala.concurrent.Await;
import scala.concurrent.Future;

import java.util.concurrent.TimeUnit;

public class DemoAkkaActorAsk{
    public static void main(String [] args) {
        ActorSystem actorSystem = ActorSystem.create("Demo");
        ActorRef actor = actorSystem.actorOf(Props.create(MyActor.class));

        Integer [] values = new Integer [] {1,2};
        Timeout timeout = new Timeout(1, TimeUnit.SECONDS);
        Future<Object> future = Patterns.ask(actor, values, timeout);

        try {
            int result = (Integer) Await.result(future, timeout.duration());
            System.out.println("Result: " + result);
        } catch (Exception e) {
            e.printStackTrace();
        }

        actorSystem.terminate();
    }

    public static class MyActor extends AbstractActor {
        private int id;

        @Override
        public Receive createReceive() {
            return receiveBuilder()
                .match(Integer [].class, this::handleIntegerMessage)
                .matchAny(message -> unhandled(message))
                .build();
        }

        private void handleIntegerMessage(Integer [] message) {
            int sum = 0;
            for (int val : message){
                sum+=val;
            }
            getSender().tell(sum, getSelf());
        }
    }
}

/* Ausgabe:
Result: 3
*/

```

# Kapitel 5

## Design by Contract

### 5.1 JML

#### 5.1.1 Basic Syntax

`\forall`

`\exists`

Syntax	Bedeutung
$a ==> b$	a impliziert b
$a <==>$	a und b äquivalent
$a <!=>$	a und b <b>nicht</b> äquivalent ( $a \nleftrightarrow b$ )
<code>\result</code>	Ergebnis der Methode
<code>\old(E)</code>	Wert von E, bevor die Methode ausgeführt wurde

# Kapitel 6

## Compiler

### 6.1 Java-Bytecode

#### 6.1.1 Präfixe / Suffixe

Präfix / Suffix	Operand Typ
i	integer
l	long
s	short
b	byte
c	character
f	float
d	double
a	reference

#### 6.1.2 Lesen / Schreiben von lokalen Variablen

Vergleich „`iload_x`“ vs. „`iload x`“

- „`iload_x`“ (mit Unterstrich):  $x \in \{0, 1, 2, 3\}$  ist ein (einziger) Befehl, ohne Parameter. „`x`“ ist schon im Opcode enthalten. Der Befehl besteht aus 1 Byte.
- „`iload x`“ (ohne Unterstrich):  $0 \leq x \leq 255$  ist ein Befehl, mit Parameter „`x`“. `x` ist nicht im Opcode enthalten. „`iload x`“ funktioniert mit allen Zahlen `x`, die in ein Byte passen.

Da die Befehle mit Unterstrich Platz sparen, werden sie von realen Compilern bevorzugt; vorausgesetzt `x` ist klein genug.

#### 6.1.3 Lesen / Schreiben von Feldern

#### 6.1.4 Sprungbefehle

#### 6.1.5 Methodenaufrufe

#### 6.1.6 Objekterzeugung

#### 6.1.7 Arithmetische Berechnungen

Befehl	Parameter	Beschreibung	Beispiel
<code>iconst_x</code>	$x \in \{0, 1, 2, 3, 4, 5, m1\}$	lädt die int-Konstante x m1 steht für Konstante „-1“	<code>iconst_1</code> lädt den int-Wert „1“ auf den Stack
<b>TYPE</b> <code>load_x</code>	x: Index der lokalen Variable $x \in \{0, 1, 2, 3\}$ (x gehört zum Befehl, es gibt pro x ein Befehl; d.h. x ist keine Variable)	lädt den Wert der Variable mit Typ „TYPE“ mit Index x auf den Stack	<code>iload_2</code> lade Wert von der Variable mit Index 2 und dem Typ „Integer“
<b>TYPE</b> <code>store_x</code>	x: Index der Variable $x \in \{0, 1, 2, 3\}$ (x gehört zum Befehl, es gibt pro x ein Befehl; d.h. x ist keine Variable)	speichert den obersten Wert auf dem Stack mit Typ „TYPE“ in Variable mit Index x	<code>istore_2</code> speichere den obersten Wert auf dem Stack vom Typ „Integer“ in Variable 2
<b>TYPE</b> <code>load x</code>	x: Index der lokalen Variable $0 \leq x \leq 255$ (x mit 1 Byte darstellbar)	lädt den Wert der Variable mit Typ „TYPE“ mit Index x auf den Stack	<code>iload 7</code> lade Wert von der Variable mit Index 7 und dem Typ „Integer“
<b>TYPE</b> <code>store x</code>	x: Index der lokalen Variable $0 \leq x \leq 255$ (x mit 1 Byte darstellbar)	speichert den obersten Wert auf dem Stack mit Typ „TYPE“ in Variable mit Index x	<code>istore 7</code> speichere den obersten Wert auf dem Stack vom Typ „Integer“ in Variable 7

Befehl	Parameter	Beschreibung	Beispiel
<b>TYPE</b> <code>mul</code>	-	multipliziert zwei Werte vom Typ „TYPE“ und lädt das Ergebnis als obersten Wert auf den Stack	<code>imul</code>
<b>TYPE</b> <code>div</code>	-	dividiert zwei Werte vom Typ „TYPE“ und lädt das Ergebnis als obersten Wert auf den Stack	<code>idiv</code>
<b>TYPE</b> <code>add</code>	-	addiert zwei Werte vom Typ „TYPE“ und lädt das Ergebnis als obersten Wert auf den Stack	<code>iadd</code>
<b>TYPE</b> <code>sub</code>	-	subtrahiert zwei Werte vom Typ „TYPE“ und lädt das Ergebnis als obersten Wert auf den Stack	<code>iadd</code>
<b>TYPE</b> <code>neg</code>	-	negiert einen Wert vom Typ „TYPE“ und lädt das Ergebnis als obersten Wert auf den Stack	<code>ineg</code>