Programmierparadigmen

1 Haskell

Basics

`++` concatenates two lists or string together

`:` adds element on the right to the list on the left

`\x -> x` basic lambda expression

`f (g x) = (f . g) x` Dot operator is composition

`($) :: (a -> b) -> a -> b` Funktionsanwendung rechtsassoziativ. `f (k a b) = f $ k a b`

Case Of und If-Then-Else

```
foo a xs = case xs of
    [] -> False
    x:xs' -> if (x == a) then True else False
```

List Comprehension

`[e|q_1, ...., q_n]` q_i sind Generatoren, binden Elemente an Namen

Bsp:
`squaredEvens l = [ x*x | x <- l, x ‘mod‘ 2 == 0]`
`squaredEvens [0..10] => [0,4,16,36,46,100]`

Datatypes und Typsynonyme

`type Student = String`

```
data BinaryTree =  leaf Int
                | node Tree Tree
```

Functions

- `abs :: Num a => a -> a`
  absolute value of the number
- `all :: (a -> Bool) -> [a] -> Bool`
  true, if all elements satisfy the predicate
- `any :: (a -> Bool) -> [a] -> Bool`
  true, if any element satisfy the predicate
- `concat :: [[a]] -> [a]`
  accepts a list of lists and concatenates them
- `concatMap :: (a -> [b]) -> [a] -> [b]`
  Map a function over all the elements of a container and concatenate the resulting lists.

- `div :: Integral a => a -> a -> a`
  returns how many times the first number can be divided by the second one.
- `drop :: Int -> [a] -> [a]`
  creates a list without the given number of items from the beginning of the second argument.
- `dropWhile :: (a -> Bool) -> [a] -> [a]`
  drops all elements until predicate is false for the first time
- `elem :: Eq a => a -> [a] -> Bool`
  true if the list contains the first argument
- `filter :: (a -> Bool) -> [a] -> [a]`
  returns a list of all elements which fulfill the predicate
- `fst :: (a,b) -> a`
  Extract the first component of a pair.
- `foldl :: (b -> a -> b) -> b -> [a] -> b`
  Left-associative fold of a structure, lazy in the accumulator. This is rarely what you want, but can work well for structures with efficient right-to-left sequencing and an operator that is lazy in its left argument.

  (((a + b) + c) + d)
- `foldr :: (a -> b -> b) -> b -> [a] -> b`
  Right-associative fold of a structure, lazy in the accumulator.
  a + (b + (c + (d)))
- `fromInteger :: Num a => Integer -> a`
  convert from Integer to Num instance
- `head :: [a] -> a`
  returns the first item of a list
- `iterate :: (a -> a) -> a -> [a]`
  returns an infinite list of repeated applications of `f` to `x`:
- `last :: [a] -> a`
  returns the last item of a list
- `length :: [a] -> Int`
  returns the number of items in a list
- `map :: (a -> b) -> [a] -> [b]`
  returns the list obtained by applying the function to each element of the list
- `max :: Ord a => a -> a -> a`
  returns the larger of its two arguments
- `maximum :: Ord a => [a] -> a`
  returns the maximum value of the list
- `min :: Ord a => a -> a -> a`
  returns the smaller of its two arguments
- `minimum :: Ord a => [a] -> a`
  returns the minimum value of the list
- `mod :: Integral a => a -> a -> a`
  return the modulus of two arguments
- `negate :: Num a => a -> a`
  change the sign of the number

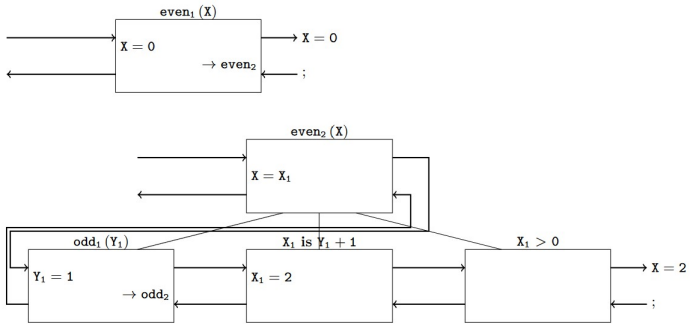
- `null :: [a] -> Bool`
  returns True if a list is empty, otherwise False
- `pred :: Enum a => a -> a`
  return predecessor
- `repeat :: a -> [a]`
  creates infinite list where all items are the first argument
- `reverse :: [a] -> [a]`
  creates new list from original with items in reverse order
- `snd :: (a,b) -> b`
  Extract the second component of a pair.
- `succ :: Enum a => a -> a`
  return successor
- `sum :: Num a => [a] -> a`
  computes the sum of a finite list of numbers.
- `tail :: [a] -> [a]`
  it accepts a list and returns the list without its first item
- `take :: Int -> [a] -> [a]`
  returns the prefix of `xs` of length `n`, or `xs` itself if `n >= length xs`.
- `takeWhile :: (a -> Bool) -> [a] -> [a]`
  returns the longest prefix (possibly empty) of elements that satisfy the predicate
- `zip :: [a] -> [b] -> [(a,b)]`
  joins two lists to a list of pairs
- `zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]`
  output elements are calculated from function and elements of input lists occuring at same position
- `(!!) :: [a] -> Int -> a`
  list index operator
- Example of instance: `instance Show a => Show (Exp a) where`

Breitensuche:

```
bfs :: Tree a -> [a]
bfs t = go1 [t] []
    where
        go1 (x:xs) result = go1 (go2 x xs) (res x result)
        go1 [] result = result
        go2 Leaf xs = xs
        go2 (Node a x b) xs = xs ++ [a,b]
        res Leaf xs = xs
        res (Node a x b) xs = xs ++ [x]
```

2 Prolog

Beispiel Ausführungsbaum für Generatorfunktion even(X) mit X = 0, X = 2



3 Lambda

- Church-Zahlen: $c_0 = \lambda s.\, \lambda z.\, z,\ c_1 = \lambda s.\, \lambda z.\, s\ z,\ c_2 = \lambda s.\, \lambda z.\, s\ (s\ z), \ldots$
- $isZero$ = $\lambda n.\, n\ (\lambda x.\, c_{false})\ c_{true}$
- Nachfolgerfunktion: $succ$ = $\lambda n.\, \lambda s.\, \lambda z.\, s\ (n\ s\ z)$
- Addition: $plus$ = $\lambda m.\, \lambda n.\, \lambda s.\, \lambda z.\, m\ s\ (n\ s\ z)$, Subtraktion: sub
- Multiplikation: $times$ = $\lambda m.\, \lambda n.\, \lambda s.\, n\ (m\ s)$
- Potenzieren: exp = $\lambda m.\, \lambda n.\, n\ m$
- Boolean: c_{true} = $\lambda t.\, \lambda f.\, t$ c_{false} =$\lambda t.\, \lambda f.\, f$
- if-then-else: if b then x else y. Bsp.: ($isZero\ \ n$) $c_{true}\ \ c_{false}$
- Rekursionsoperator Y= $\lambda f.\, (\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x))$
  - sei g Rekursiv mit g = ... g ....., dann Bilde G = λg. .... g ....
    G ist dann Fixpunkt und die Rekursive funkion ergibt sich als Y G $\equiv$ g

Reduktions Varianten

Redex: λ-Term der From $(\lambda x.\, t_1)\ t_2$

- β-Reduktion: Reduziere beliebigen Redex
- Normalreihenfolge: Reduziere linksten äußersten Redex
  - findet Normalform immer, CBN und CBV finden diese nicht immer
- Call-by-name: Reduziere linkesten äußersten Redex (nicht, falls von λ umgeben)
  - Haskell, Lazy-Evaluation = CBN + shairing, terminiert öfter
- Call-by-value: Reduziere linksten Redex (nicht, falls von λ umgeben, und dessen Argument Wert ist)
  - Java, C

Äquivalenzen

- α-Äquivalenz: zwei Terme sind äquivalent, wenn sie durch Umbenennung λ-gebundener Variablen identisch sind. Beispiel: $\lambda x.\, x = \lambda y.\, y$
- η-Äquivalenz: Terme $\lambda x.\, f\ x$ und f heißen η-Äquivalent ($f = \lambda x.\, f\ x$), falls x nicht freie Variable von f. Beispiele:
  - dafür: $fz = \lambda x.\, f\, z\, x$
  - dagegen: $\lambda x.\, g\ x\ x \neq g\ x$
- β-Reduktion, zählt auch als Äquivalenz, wenn sie zu gleichen Termen auswerten

Typinferenz

Normale Regeln

$Const\ \frac{c \in Const}{\Gamma \vdash c : \tau_c}$ $Var\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$

$ABS\ \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2}$ $APP\ \frac{\Gamma \vdash t_1 : \tau_2 \to \tau \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1\ t_2 : \tau}$

Polymorphismus

$VAR\ \frac{\Gamma(x) = \phi \quad \phi \succeq \tau}{\Gamma \vdash x : \tau}$ $Let\ \frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma, x : ta(\tau_1, \Gamma) \vdash t_2 : \tau_2}{\Gamma \vdash let\ x = t_1\ in\ t_2 : \tau_2}$

Typinferenz für Let

Typschemata $ta(\tau, \Gamma) = \forall \alpha_1.\ \ldots\ \forall \alpha_n.\ \tau$ heißt Typabastraktion von τ relativ zu Γ, wobei $\alpha_i \in FV(\tau) \setminus FV(\Gamma)$. Also alle freien Variablen von τ quantifizieren, die nicht frei in der Typannahme Γ vorkommen.

$Let \frac{\Gamma \vdash t_1 : \alpha_i \quad \Gamma' \vdash t_2 : \alpha_j}{\Gamma \vdash let\ x = t_1\ in\ t_2 : \alpha_k}$

- Sammle Constraints aus linkem Teilbaum in C_{let}
- Berechne mgu σ_{let} von C_{let}
- Berechne $\Gamma' := \sigma_{let}(\Gamma), x : ta(\sigma_{let}(\alpha_i), \sigma_{let}(\Gamma))$
- Benutze Γ' in rechtem Teilbaum, sammle Constraints in $\mathbb{C}_{body}$
- Ergebnisconstraints sind: $C'_{let} \cup C_{body} \cup \{\alpha_j = \alpha_k\}$
  wobei $C'_{let} := \{\alpha_n = \sigma_{let} | \sigma_{let}$ definiert für $\alpha_n\}$

4.4 Robinson-Algorithmus

```
if C == ∅ then []
else let {θl = θr} ∪ C' = C in
  if θl == θr then unify(C')
  else if θl == Y and Y /∈ FV(θr) then unify([Y ⇒ θr] C') ◦ [Y ⇒ θr]
  else if θr == Y and Y /∈ FV(θl) then unify([Y ⇒ θl] C') ◦ [Y ⇒ θl]
  else if θl == f (θ1l , . . . , θnl) and θr == f (θ1r , . . . , θnr)
    then unify(C' ∪ {θ1l = θ1r , . . . , θnl = θnr})
  else fail
```

Allgemeinster Unifikator: σ mgu, falls ∀ Unifikator γ ∃ Substitution δ. γ = δ ◦ σ.

5 und 6 Parallels Fundamentales

Basics

- Coffman conditions: mutual exclusion, hold and wait, no preemption, circular wait
- Amdahl's Law: $S(n) = \frac{1}{(1-p)+\frac{p}{n}}$

Task / Data Parallelismus

Task Parallelismus:
möglichst unabhängige Aufgaben werden parallelisert

Daten Parallelismus:
Daten werden partitioniert und auf den Partitionen die gleiche Aufgabe ausgeführt

Flynn's Taxonomy

1. SISD: Single Instruction x Single Data
    1. von Neumann Architektur
2. SIMD: Single Instruction x Multiple Data
    1. eine Instruction, gleichzeitig auf mehrere Daten (z.B. Arrays). Vektor Maschienen
3. MIMD: Multiple Instruction x Multiple Data
    1. jeder Prozessor hat eigene Instruktion und Daten. Moderne Parrallel Maschienen
4. MISD: Multiple Instruction x Single Data
    1. mehrere Instructionen auf gleichen Daten. (Kontrovers: Piplining)

5 MPI

Communication Modes

- nur ein receive mode - für alle vier sende Operationen.
- 4 send modes:
    1. Synchronos: kein Puffer, Synchronisation (beide Seiten warten aufeinander)
    2. Bufferd: explizieter Puffer, keine Synchonisation (kein Warten aufeinander)
    3. Ready: kein Puffer, keine Synchronisation (Recieve seite muss schon bereit sein)
    4. Standard: beliebig, hängt von Implementierung ab

Blocking Communication

1. Blocking: Aufruf gibt erst nach completter Ausführung des Befehls zurück. Nach Rückgabe, ist sicher, dass man Puffer neu benutzen kann
2. Non-Blocking: Aufruf gibt direkt zurück. Während des Wartens auf beenden der Send-Operation können andere Befehele ausgeführt werden. Weniger Deadlock anfällig, mehr Error anfällig, da man Puffer nicht überschreiben sollte, ohne zu checken ob Operation beendet ist.

Datatypes

```
MPI_INT, MPI_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR
```

Basic Functions

- puts the rank of the caller process in variable rank

```
int MPI_Comm_rank(MPI_Comm comm, int* rank)
```

- puts the number of processes in variable size

```
int MPI_Comm_size(MPI_Comm comm, int* size)
```

- initializes MPI execution environment

```
int MPI_Init(int* argc, char*** argv)
```

- terminates MPI execution environment

```
int MPI_Finalize()
```

- blocks until all processes have called it

```
int MPI_Barrier(MPI_COMM comm)
```

- Send and receive operations can be checked for completion.
  Non blocking check, set flag to 1 if complete 0 otherwise

```
int MPI_Test(MPI_Request* r, int* flag, MPI_Status* s)
```

- Send and recive operations can be checked for completion.
  Blocking check

```
int MPI_Wait(MPI_Request* r, MPI_Status* s)
```

Send and Receive Functions

- performs a blocking send.
  _Send for standard, _Bsend for bufferd, _Ssend for synchronous, _Rsend for ready

```
int MPI_Send(void* buffer, int count, MPI_Datatype datatype,
int dest, int tag, MPI_Comm comm)
```

- Blocking receive for message.

```
int MPI_Recv(void* buffer, int count, MPI_Datatype datatype,
int source, int tag, MPI_Comm comm, MPI_Status* status)
```

- begins a nonblocking send.

```
int MPI_Isend(void* buf, int count, MPI_Datatype type,
int dest, int tag, MPI_Comm comm, MPI_Request* request)
```

- Begins nonblocking receive.

```
int MPI_Irecv( void* buf, int count, MPI_Datatype type,
int src, int tag, MPI_Comm comm, MPI_Request* request)
```

- kombinierte Send- and Recv-Operation (benötigt zwei Buffer), es gibt auch Sendrecv_replace (nur ein Buffer)

```
int MPI_Sendrecv(void* sendbuf, int sendcount, MPI_Datatype sendtype, int dest,
int sendtag, void* recvbuf, int recvcount, MPI_Datatype recvtype, int source, int
recvtag, MPI_Comm comm, MPI_Status *status)
```

Reduce Functions

- reduces values on all processes to a single value (using operations such as MPI SUM, MPI MAX or MPI MIN).

```
int MPI_Reduce(void* sendbuf, void* recvbuf, int count, MPI_Datatype type, MPI_Op
op, int root, MPI_Comm comm)
```

- MapReduce
    1. Map: apply a map function to the data on each computing node once,returning key-value pairs as a result,
    2. Shuffle: redistribute data by output keys of the map function, so that one computing node contains all data for one key
    3. Reduce: apply reduce function on each key once
- acceptable Operations for MPI_Op
  logical / bitwise “and” / “or”:
  `MPI_LAND, MPI_BAND, MPI_LOR, MPI_BOR`

  Math operations
  `MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD`

  find local minimum / maximum and return the value of the "causing" rank
  `MPI_MINLOC, MPI_MAXLOC`

Global Collective functions

- sends data from one process to all other processes in a communicator.

```
int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype,
void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

- like scatter, but with varying counts for data sent to each process
  sendcounts: hält an jedem Index i die Anzahl der zu versendenden Werte für den Prozess mit Rank i
  displacements: hält an jedem Index i für den Prozess mit Rank i den Startindex von sendbuf, ab dem die nächsten sendcounts[i] vielen Elemente verschickt werden

```
int MPI_Scatterv( void* sendbuf, int* sendcounts, int* displacements,
MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int
root, MPI_Comm comm)
```

- gathers together values from a group of processes.

```
int MPI_Gather( void* sendbuf, int sendcount, MPI_Datatype sendtype, void*
recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

- Broadcast. Everyone has to participate

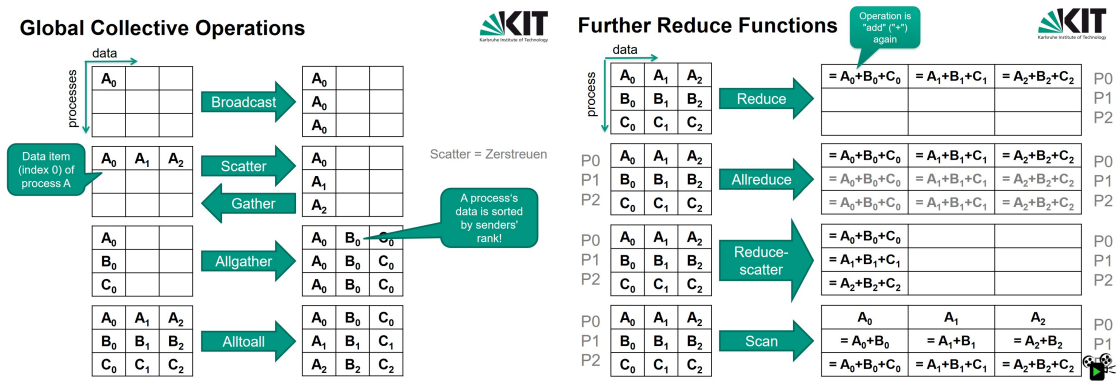
```
int MPI_Bcast( void* buffer, int count, MPI_Datatype t, int root, MPI_Comm comm)
```

- gathers data from all tasks and distribute the combined data to all tasks. Basically gather + broadcast

```
int MPI_Allgather(void* sendbuf, int sendcount, MPI_Datatype sendtype, void*
recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

- sends data from all to all processes. Also with a vector variant 'v' and with 'w' allows separate specification of count, displacement and datatype for each block

```
int MPI_Alltoall(void *sendbuf, int sendcount, MPI_Datatype sendtype, void
*recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```



6 Java

- `volatile` schreibt Werte immer zurück in Hauptspeicher, so dass keine Cachefehler auftreten können.
- `wait()` immer in while loop
- `notifyALL()` besser als `notify()`, immer aus `synchronized()` aufrufen

Executor

- `ExecutorService::newSingleThreadExecutor()`
- `ExecutorService::newFixedThreadPool(int n)`
- `ExecutorService::newCachedThreadPool()`
- `ExecutorService::execute(Runnable runnable)`
- `ExecutorService::shutdown()`
- `Future<String> future = exeService.submit(() -> return "Hi"; );`
- `String s = future.get();`

```
List<Long> result = new ArrayList<>();
List<Future<Long>> futures = new ArrayList<>();

ExecutorService executor = Executors.newFixedThreadPool(4);

for (int i = 0; i < 10; i++) {
    futures.add( executor.submit(
        () -> foo(i)
    ));
    //wenn lambda mit {} dann ist return notwendig
    // () -> {retrun foo(i);}
}


for (List<Future<Long>> future : futures) {
    result.add(future.get());
}

return result;
```

Streams

Operations: filter, map, reduce, collect, findAny, findFirst, min, max ...

Example:

```
myList.stream().filter(MyClass::filterFoo).mapToInt(MyClass::mapFoo).average();

myList.parralelStream().allMatch(element -> element > 100).count();
myList.parralelStream().reduce((a,b) -> foo(a,b)).get();
myList.parralelStream().collect(
    () -> 0,
    (currentSum, person) -> {currentSum += person.getAge();},
(leftSum, rightSum) -> {leftSum += rightsum;});
```

7 Design By Contract

```
/*@ requires true
  @ requires array.size > 0;
  @ ensures array.size > 0 ==> \old(array).size > 1;
  @ ensures \result > 0;
  @*/
```

- Liskovsche Substitution:
  - Vorbedinungen können nur schwächer werden, maximal genauso stark
    $Precondition_{Super} \Rightarrow Precondition_{Sub}$
  - Nachbedinungen können nur stärker werden, mindestens genauso stark
    $Postcondition_{Sub} \Rightarrow Postcondition_{Super}$
  - Invarianten können nur stärker werden, mindestens genauso stark
    $Invariants_{Sub} \Rightarrow Invariants_{Super}$

8 Compilerbau

- Indizmenge von $A \rightarrow \alpha$: $First_k(\alpha\, Follow_k(A))$
- SLL(k)-Bedingung: $\forall A \rightarrow \alpha | \beta :\ First_k(\alpha\, Follow_k(A)) \cap First_k(\beta\, Follow_k(A)) = \emptyset$
  - Linksrekursive (LR) Grammatiken sind nie SLL(k), man kann so eine Grammatik jedoch immer ohne Linksrekursion darstellen. (LR: T -> T a | b)
- First, Follow, Indizmenge intuitiv:
  - $First_k(A)$ alle Terminale, die bei irgendeiner Ableitung von A an erster Stelle stehen
  - $Follow_k(A)$ enthält alle Terminale, die bei irgendeiner Ableitung von S (dem Startsymbol) direkt hinter A stehen. Überlegen, welche Ableitungsschritte zu einem Vorkommen von A führen, und was dann direkt dahinter stehen kann. An # denken.
  - $First_k(\alpha\, Follow_k(A))$: entweder $Frist_k(\alpha)$, oder wenn ϵ-Produktion, dann $Follow_k(A)$
- Linksfaktorisierung: $X \rightarrow \gamma\alpha \mid \gamma\beta \Rightarrow X \rightarrow \gamma X'\ ,\ X' \rightarrow \alpha \mid \beta$
  Man möchte gemeinsame Anfänge innerhalb einer Produktion vermeiden

Rekursiven Abstiegsparser

```
void parseNichtTerminalSymbolMitMehrerenAbleitungen() {
    switch(lexer.current.type) {
        case typ1:
            lexer.lex();
            parseNichtTerminalMitEinerAbleitung()
        default:
            error(); }}

void parseNichtTerminalMitEinerAbleitung() {
    if (lexer.current.type != typ1) {
        error();}
    lexer.lex();
    parseNächstesNichtTerminal();}
```

Beachte, was passiert, wenn ε mit in der Produktion enthalten ist:

$$
\begin{aligned}
&Fact \ \rightarrow \ Term \ . \\
&Term \ \rightarrow \ atom Atom' \mid number \mid variable \\
&Termlist \ \rightarrow \ Term \ Termlist' \\
&Termlist' \ \rightarrow \ \varepsilon \mid , \ Term \ Termlist' \\
&Atom' \ \rightarrow \ \varepsilon \mid (\ Termlist \ )
\end{aligned}
$$

```
void parseAtom'() {
    switch (token.getType()) {
        case dot:   // ergeben sich Follow Menge von Term
        case comma:
        case rp:
            break;
        case lp:
            nextToken();
            parseTermlist();
            if (token.getType() != rp) {
                error();}
            nextToken();
            break;
        default:
            error();}}
```

Achtung: Die aktuelle VL nutzt nicht mehr `token`, sondern `lexer` mit den Funktionen `lexer.current.type, lexer.current.text, lexer.lex()`.

Umgekehrte Polnische Notation

Bsp: $a * (x - 1) + c \ \rightarrow a \ x \ 1 \ - \ * \ c \ +$

Aus UPN kann man 1 zu 1 die Befehle abschreiben, um Bytecode zu erhalten.