Programmierparadigmen macht Spaß

Darius Schefer, Max Schik

Contents

Haskell

Haskell is great.

General Haskell stuff

```haskell
-- type definitions are right associative
foo :: (a -> (b -> (c -> d)))
-- function applications are left associative
((((foo a) b) c) d)

-- guards are unnecessary if you know how pattern matching works
foo x y
  | x > y = "shit"
  | x < y = "piss"
  | x == y = "arschsekretlecker"
  | default = "love"

-- case of does pattern matching so its okay
foo x = case x of
          [] -> "fleischpenis"
          [1] -> "kokern"
          (420:l) -> "pimpern"

-- list comprehension is not as good as in python
[foo x | x <- [1..420], x `mod` 2 == 0]

-- alias for pattern matching
foo l@(x:xs) = l == (x:xs) -- returns true

-- combine two functions
f :: a -> b
g :: b -> c

h :: a -> c
h = f . g

data Tree a = Leaf
              | Node (Tree a) a (Tree a)
              deriving (Show)
```

```haskell
-- defines interface
class Eq t where
  (==) :: t -> t -> Bool
  (/=) :: t -> t -> Bool
  -- default implementation
  x /= y = not $ x == y

-- extends interface
class (Show t) => B t where
  foo :: (B t) -> String

-- implement interface
instance Eq Bool where
  True == True = True
  False == False = True
  True == False = False
  False == True = False
```

Important functions

```haskell
-- maps a function to a list
map :: (a -> b) -> [a] -> [b]
-- filters a list with a predicate
filter :: (a -> Bool) -> [a] -> [a]
-- fold from left
foldl :: Foldable t => (b -> a -> b) -> b -> t a -> b
-- fold from right
foldr :: Foldable t => (a -> b -> b) -> b -> t a -> b
-- checks if a in collection
elem :: (Foldable t, Eq a) => a -> t a -> Bool
-- in a list of type [(key, value)]  returns first element where key matches given value
lookup :: Eq a => a -> [(a, b)] -> Maybe b
-- repeated application of function
iterate :: (a -> a) -> a -> [a]
-- repeats constant in infinite list
repeat :: a -> [a]
-- applies function until the predicate is true
until :: (a -> Bool) -> (a -> a) -> a -> a
-- returns true if the predicate is true for at least one element
any :: Foldable t => (a -> Bool) -> t a -> Bool
-- return true if the predicate is true for all elements
all :: Foldable t => (a -> Bool) -> t a -> Bool
-- flips the parameters of a function
flip :: (a -> b -> c) -> b -> a -> c
-- combines two lists to a list of tuples
zip :: [a] -> [b] -> [(a, b)]
-- combines two lists with the given function
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
```

Lambda Calculus

General stuff

- Function application is left associative $\lambda x.\ f\ x\ y = \lambda x.\ ((f\ x)\ y)$
- untyped lambda calculus is turing complete

Primitive Operations

Let

- let $x = t_1$ in t_2 wird zu $(\lambda x.\, t_2)\, t_1$

Church Numbers

- $c_0 = \lambda s.\, \lambda z.z$
- $c_1 = \lambda s.\, \lambda z.s\, z$
- $c_2 = \lambda s.\, \lambda z.s\,(s\, z)$
- $c_3 = \lambda s.\, \lambda z.s\,(s\,(s\, z))$
- etc...
- **Successor Function**
  - $succ\, c_2 = c_3$
- **Arithmetic Operations**
  - **TODO**
  - Addition: *plus*
  - Multiplikation: *times*
  - Potenzieren: *exp*

Boolean Values

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

Equivalences

α-equivalence

Two terms t_1 and t_2 are α-equivalent $t_1 \overset{\alpha}{=} t_2$ if t_1 and t_2 can be transformed into each other just by consistent renaming of the bound variables.

Example

$$\lambda x.x \overset{\alpha}{=} \lambda y.y$$

$$\lambda x.(\lambda z.f(\lambda y.zy)x) \overset{\alpha}{\neq} \lambda z.(\lambda z.f(\lambda y.z\ y)z)$$

η-equivalence

Two terms $\lambda x.f\ x$ and f are η-equivalent $\lambda x.f\ x \overset{\eta}{=} f$ if x is not a free variable of f.

Example

$$\lambda x.f\, z\, x \overset{\eta}{=} f\, z$$

$$\lambda x.g\, x\, x \overset{\eta}{\neq} g\, x$$

Reductions

β-reduction

A λ-term of the shape $(\lambda x.x)\ y$ is called a Redex. The β-reduction is the evaluation of a function application on a redex.

$$(\lambda x.t_1)\, t_2 \Rightarrow t1\ [x \mapsto t_2]$$

Normal Form

A term that can no longer be reduced is called Normal Form. The Normal Form is unique. Terms that don't get reduced to Normal Form diverge (grow infinitely large.

Church-Rosser

The untyped λ is confluent $\Leftrightarrow$ If $t \overset{*}{\Rightarrow} t_1$ and $t \overset{*}{\Rightarrow} t_2$ then there exists a t' with $t_1 \overset{*}{\Rightarrow} t'$ and $t_2 \overset{*}{\Rightarrow} t'$,

Recursion

For a recursive function $G = \lambda g.\ (\lambda x.\ g\ x)$ has the fixpoint $g^* = Gg^*$ if it exists.

$Y = \lambda f. (\lambda x. f\,(x\,x))\,(\lambda x. f\,(x\,x))$ is called the recursion operator. $Y\ G$ is the fixpoint of G.

Evaluation Strategies

Full β-Reduction

Every Redex can be reduced at any time.

Normal Order

The leftmost outer redex gets reduced.

Call by Name (CBN)

Reduce the leftmost outer Redex if not surrounded by a lambda.

Example

$$(\lambda y.\ (\lambda x.\ y\ (\lambda z.\ z)\ x))\ ((\lambda x.\ x)\ (\lambda y.\ y))$$

$$\Rightarrow (\lambda x.\ ((\lambda x.\ x)\ (\lambda y.\ y))\ (\lambda z.\ z)\ x) \not\Rightarrow$$

Call by Value (CBV)

Reduce the leftmost Redex if not surrounded by a lambda and the argument is a value. A value means the term can not be further reduced.

Example

$$(\lambda y.\ (\lambda x.\ y\ (\lambda z.\ z)\ x))\ ((\lambda x.\ x)\ (\lambda y.\ y))$$

$$\Rightarrow (\lambda y.\ (\lambda x.\ y\ (\lambda z.\ z)\ x))\ (\lambda y.\ y)$$

$$\Rightarrow (\lambda x.\ (\lambda y.\ y)\ (\lambda z.\ z)\ x) \not\Rightarrow$$

Call by Name and Call by Value may not reduce to the Normal Form! Call by Name terminates more often than Call by Value.

Typen

Regelsysteme

- definieren bestimmte Terme als "herleitbar" (geschr. "$\vdash \psi$")
- Frege'sche Regelnotation: aus dem über dem Strich kann man das unter dem Strich herleiten
- Introduktions- und Eliminationsregeln für und/oder, Quantoren etc. **TODO:** screenshot oder mathtex dafür
- Modus Ponens $\frac{\vdash \psi \Rightarrow \phi \quad \vdash \psi}{\vdash \phi}$
  - Elimination von Implikation
- LEM $\frac{}{\vdash \phi \vee \neg \phi}$
  - (Law of excluded middle)
  - "Es gilt immer ϕ oder $\neg \phi$"
- Beweiskontext: $\Gamma \vdash \phi$
  - ϕ unter Annahme von Γ herleitbar
  - Erleichtert Herleitung von $\phi \Rightarrow \psi$
  - Assumption Introduktion $\frac{}{\Gamma, \phi \vdash \phi}$

Typsysteme

- Einfache Typisierung
  - $\vdash (\lambda x.\, 2) : \mathit{bool} \to \mathit{int}$
  - $\vdash (\lambda x.\, 2) : \mathit{int} \to \mathit{int}$
  - $\vdash (\lambda f.\, 2) : (\mathit{int} \to \mathit{int}) \to \mathit{int}$
- Polymorphe Typen
  - $\vdash (\lambda x.\, 2) : \alpha \to \mathit{int}$

Regeln

- $\Gamma \vdash t : \tau$: im Typkontext Γ hat Term t den Typ τ
- Γ ordnet freien Variablen x ihren Typ $\Gamma(x)$ zu
- **CONST**

$$CONST \ \frac{c \in Const}{\Gamma \vdash c : \tau_c}$$

- **VAR**

$$VAR \ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

- **ABS**

$$ABS \ \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.\, t : \tau_1 \to \tau_2}$$

- **APP**

$$APP \ \frac{\Gamma \vdash t_1 : \tau_2 \to \tau \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1 \ t_2 : \tau}$$

Forts. Typsysteme

- Nicht alle sicheren Programme sind Typsierbar
  - Typsystem nicht vollständig bzgl. β-Reduktion
    - insb. Selbsapplikation im Allgemeinen nicht Typisierbar
    - damit auch nicht Y-Kombinator

Polymorphie

- **Polymorphe Funktionen**
  - Verhalten hängt nicht vom konkreten Typ ab
  - z.B. Operationen auf Containern, wie z.B. Listen

Typschema

- Für $n \in \mathbb{N}$ heißt $\forall\alpha_1 \ldots . \forall\alpha_n.\tau$ *Typschema* (Kürzel ϕ)
- Es bindet freie Typvariablen $\alpha_1, \ldots, \alpha_n$ in τ
- VAR-Regel muss angepasst werden

$$VAR \ \frac{\Gamma(x) = \phi \quad \phi \succeq \tau}{\Gamma \vdash x : \tau}$$

- LET-Typregel

$$LET \ \frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma, x : ta(\tau_1, \Gamma) \vdash t_2 : \tau_2}{\Gamma \vdash let\ x = t_1\ in\ t_2 : \tau_2}$$

- ta(τ, Γ): Typabstraktion
  - Alle freien Typvariablen von τ quantifiziert, die nicht frei in Typannahmen von Γ
  - => Verhindere Abstraktion von globalen Typvariablen im Schema

Prolog

Generelles Zeug

Prolog ist nicht vollständig da die nächste Regel deterministisch gewählt wird, daher können Endlosschleifen entstehen und keine Lösung gefunden werden obwohl sie existiert.

```
% klein geschriebene Namen sind Atome
mag(ich, dich). % nein tu ich nicht

% Prolog erfüllt Teilziele von links nach rechts
foo(X) :- subgoal1(X), subgoal2(X), subgoal3(X).

% ! signalisiert einen cut, alles vor dem cut ist nicht reerfüllbar.
% Arten von Cuts:
%  - Blauer Cut
%      - beeinflusst weder Programmlaufzeit, noch -verhalten
%  - Grüner Cut
%      - beeinflusst Laufzeit, aber nicht Verhalten
%  - Roter Cut
%      - beeinflusst das Programmverhalten
% Zuweisungen immer nach dem cut!
foo(X, Y) :- operation_where_we_only_want_the_first_result(X, Z), !, Y = Z.

% generate and test
foo(X, Y) :- generator(X, Y), tester(Y).

% listen sind so wie in haskell
foo([H|T]) :- ...

% weitere listen sachen
[1,2,3|[4,5,6,7]] = [1,2,3,4,5,6,7]

% Arithmetik ist komisch. 2 - 1 ist ein Term, keine Zahl!
2 - 1 \= 1

% Um Terme auszuwerten braucht man "is"
N1 is N - 1.
```

Wichtige Funktionen

```
% prüft ob X in L
member(X, L).
```

```
% fügt A und B zu C zusammen.
append(A, B, C).

% Länge N einer Liste L
length(L, N).

% sowas wie append kann auch als Generator verwendet werden, sofern C instanziiert ist.
append(A, B, C) % A und B gehen durch alle Teillisten von C

% Negation
not(X). % X ist ein prädikat
```

Unifikation

Unifikator

- Gegeben: Menge C von Gleichungen über Terme
- τ = Basistyp, X = Var
- Gesucht ist eine Substitution, die alle Gleichungen erfüllt: **Unifikator**
- **most general unifier**, mgu ist der allgemeinste Unifikator

Definition Unifikator

Substitution σ unifiziert Gleichung $\theta = \theta'$, falls $\sigma\theta = \sigma\theta'$.

σ unifiziert C, falls $\forall c \in C$ gilt: σ unifiziert c.

Bsp. $C = \{f(a, D) = Y, X = g(b), g(Z) = X\} \Rightarrow \sigma = [Y \to f(a, b), D \to d, X \to g(b), Z \to b]$

Definition mgu

σ mgu, falls $\forall$ Unifikator γ $\exists$ Substitution δ. $\gamma = \delta \circ \sigma$.

- Unifikator mit der minimalen Menge an Substitutionen
- Für das Beispiel: $\sigma = [Y \to f(a, D), X \to g(b), z \to b]$
  - für γ z. Bsp. $\delta = [D \to b]$

Unifikationsalgorithmus: `unify(C)` =

`if C ==` $\emptyset$ `then []`

`else let` $\{\theta_l = \theta_r\} \uplus \mathtt{C}' = \mathtt{C}$ `in`

  `if` θ_l `==` θ_r `then unify(C')`

  `else if` θ_l `== Y and Y` $\notin$ `FV(`θ_r`) then unify([Y` $\to \theta_r$`]C')` $\circ$ `[Y` $\to \theta_r$`]`

  `else if` θ_r `== Y and Y` $\notin$ `FV(`θ_l`) then unify([Y` $\to \theta_l$`]C')` $\circ$ `[Y` $\to \theta_l$`]`

  `else if` θ_l `== f(`$\theta_l^1, ..., \theta_l^n$`) and` θ_r `== f(`$\theta_r^1, ..., \theta_r^n$`)`

    `then unify(C'` $\cup \{\theta_l^1 = \theta_r^1, ..., \theta_l^n = \theta_r^n\}$`)`

  `else fail`

`unify(C)` terminiert und gibt **mgu** für C zurück, falls C unifizierbar, ansonsten **fail**.

Parallelprogrammierung

Uniform Memory access (UMA): .

Parallelismus: Mindestens zwei Prozesse laufen gleichzeitig.

Concurrency: Mindestens zwei Prozesse machen Fortschritt.

Amdahls' Law:

$$S(n) = \frac{T(1)}{T(n)} = \frac{\text{execution time if processed by 1 processor}}{\text{execution time if processed by n processors}} = \text{speedup}$$

$$S(n) = \frac{1}{(1-p) + \frac{p}{n}} \text{ with } p = \text{parallelizable percentage of program}$$

Data Parallelism: Die gleiche Aufgabe wird parallel auf unterschiedlichen Daten ausgeführt.

Task Parallelism: Unterschiedliche Aufgaben werden auf den gleichen Daten ausgeführt.

Flynn's Taxanomy

Name	Beschreibung	Beispiel
SISD	a single instruction stream operates on a single memory	von Neumann Architektur
SIMD	one instruction is applied on homogeneous data (e.g. an array)	vector processors of early supercomputer
MIMD	different processors operate on different data	multi-core processors
MISD	multiple instructions are executed simultaneously on the same data	redundant architectures

MPI

```
// default communicator, i.e. the collection of all processes
MPI_Comm MPI_COMM_WORLD;

// returns the number of processing nodes
int MPI_Comm_size(MPI_Comm comm, int *size);

// returns the rank for the processing node, root node has rank 0
int MPI_Comm_size(MPI_Comm comm, int *size);

// initializes MPI
int MPI_Init(int *argc, char ***argv);


// Cleans up MPI (called in the end)
int MPI_Finalize();

// blocks until all processes have called it
int MPI_Barrier(MPI_Comm comm);

// blocking asynchrounous send. blocks until message buffer can be reused, i.e. message has been received.
int MPI_Send(const void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)

// blocking asynchrounous receive. blocks until message is received in the buffer completly.
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
             MPI_Comm comm, MPI_Status *status)
```

Communication Modes

Synchronous

No buffer, synchronization (both sides wait for each other)

Buffered

Explicit buffering, no synchronization (no wait for each other)

Ready

No buffer, no synchronization, matching receive must already be initiated

Standard

May be buffered or not, can be synchronous (implementation dependent)

There is only one receive mode.

```
MPI_Send() // standard-mode blocking send
MPI_Bsend() // buffered-mode blocking send
MPI_Ssend() // synchronous-mode blocking send
MPI_Rsend() // ready-mode blocking send

// non-blocking send and receive operations
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source,
              int tag, MPI_Comm comm, MPI_Request * request);
int MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int dest, int tag,
              MPI_Comm comm, MPI_Request *request);

// send and receive operations can be checked for completion
int MPI_Test(MPI_Request* r, int* flag, MPI_Status* s);
// blocking check
int MPI_Wait(MPI_Request* r, MPI_Status* s);
```

Collective Operations

`MPI_Bcast`

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

$$\begin{bmatrix} A_0 & & \\ & & \\ & & \end{bmatrix} \overset{Broadcast}{\rightarrow} \begin{bmatrix} A_0 & & \\ A_0 & & \\ A_0 & & \end{bmatrix}$$

`MPI_Scatter MP_Gather`

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

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

$$\begin{bmatrix} A_0 & A_1 & A_2 \\ & & \\ & & \end{bmatrix} \underset{gather}{\overset{scatter}{\rightleftarrows}} \begin{bmatrix} A_0 & & \\ A_1 & & \\ A_2 & & \end{bmatrix}$$

MPI_Allgather

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

$$\begin{bmatrix} A_0 & & \\ B_0 & & \\ C_0 & & \end{bmatrix} \overset{Allgather}{\rightarrow} \begin{bmatrix} A_0 & B_0 & C_0 \\ A_0 & B_0 & C_0 \\ A_0 & B_0 & C_0 \end{bmatrix}$$

MPI_Alltoall

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

$$\begin{bmatrix} A_0 & A_1 & A_2 \\ B_0 & B_1 & B_2 \\ C_0 & C_1 & C_2 \end{bmatrix} \overset{Alltoall}{\rightarrow} \begin{bmatrix} A_0 & B_0 & C_0 \\ A_1 & B_1 & C_1 \\ A_2 & B_2 & C_2 \end{bmatrix}$$

MPI_Reduce

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

$$\begin{bmatrix} A_0 & A_1 & A_2 \\ B_0 & B_1 & B_2 \\ C_0 & C_1 & C_2 \end{bmatrix} \overset{Reduce}{\rightarrow} \begin{bmatrix} A_0 + B_0 + C_0 & A_1 + B_1 + C_1 & A_2 + B_2 + C_2 \\ & & \\ & & \end{bmatrix}$$

MPI_allreduce

```
int MPI_Allreduce(const void *sendbuf, void *recvbuf, int count,
                  MPI_Datatype datatype, MPI_Op op, MPI_Comm comm);
```

$$\begin{bmatrix} A_0 & A_1 & A_2 \\ B_0 & B_1 & B_2 \\ C_0 & C_1 & C_2 \end{bmatrix} \overset{Allreduce}{\rightarrow} \begin{bmatrix} A_0 + B_0 + C_0 & A_1 + B_1 + C_1 & A_2 + B_2 + C_2 \\ A_0 + B_0 + C_0 & A_1 + B_1 + C_1 & A_2 + B_2 + C_2 \\ A_0 + B_0 + C_0 & A_1 + B_1 + C_1 & A_2 + B_2 + C_2 \end{bmatrix}$$

MPI_Reduce_scatter

```
int MPI_Reduce_scatter(const void *sendbuf, void *recvbuf, const int recvcounts[],
                       MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
```

$$\begin{bmatrix} A_0 & A_1 & A_2 \\ B_0 & B_1 & B_2 \\ C_0 & C_1 & C_2 \end{bmatrix} \overset{Reduce-scatter}{\rightarrow} \begin{bmatrix} A_0 + B_0 + C_0 & & \\ A_1 + B_1 + C_1 & & \\ A_2 + B_2 + C_2 & & \end{bmatrix}$$

MPI_Scan

```
int MPI_Scan(const void *sendbuf, void *recvbuf, int count, MPI_Datatype datatype,
             MPI_Op op, MPI_Comm comm);
```

$$\begin{bmatrix} A_0 & A_1 & A_2 \\ B_0 & B_1 & B_2 \\ C_0 & C_1 & C_2 \end{bmatrix} \overset{Scan}{\rightarrow} \begin{bmatrix} A_0 & A_1 & A_2 \\ A_0 + B_0 & A_1 + B_1 & A_2 + B_2 \\ A_0 + B_0 + C_0 & A_1 + B_1 + C_1 & A_2 + B_2 + C_2 \end{bmatrix}$$

Java

Multithreading

Race conditions

A race condition exists if the order in which threads execute thei operations influences the result of the program.

Mutual Exclusion

A code section, of which only one thread is allowed to execute operations at a time, is called a critical section. If one thread exectues operations of a critical section, other threads will be blocked if they want to enter it as well.

```java
// Synchronized block, someObject is used as the monitor
synchronized(someObject) {
  ...
}

// synchronized function
synchronized void foo() {
  ...
}
```

Caching and code reordering

- cached variables can lead to inconsistency
- code can be reordered by the compiler

`volatile`-keyword

`volatile` ensures that changes to variables are immediately visible to all threads/processors.

- establishes a happens-before relationship
- values are not locally cached in a CPU cache
- no optimization by compiler

```java
// declares a volatile variable
volatile int c = 420;
```

Functional programming

```java
// lambdas
(int i, int j) -> i + j;

// functional interfaces
@FunctionalInterface
interface Predicate {
  boolean check(int value);
}

public int sum(List<Integer> values, Predicate predicate) {
  ...
};

sum(values, i -> i > 5);

// method reference to static function
SomeClass::staticFunction;
// method reference to object function
someObject::function;
```

Executors

- Executors abstract from thread creation.
- provides an execute method that accepts a `Runnable`

  ```
  void execute(Runnable runnable);
  ```

- `ExecutorService` is an interface that provides further lifecycle management logic

```
Callable<Integer> myCallable = () -> { return currentValue; };
Future<Integer> myFuture = executorService.submit(myCallable);
```

Streams

Provides functions like

- `filter`
- `map, reduce`
- `collect`
- `findAny`, `findFirst`
- `min`, `max`

Any Java collection can be treated as a stream by calling the `stream()` method

Example

```
List<Person> personsInAuditorum = ...;
double average =
  personsInAuditorum
  .stream()
  .filter(Person::isStudent)
  .mapToInt(Person::getAge) // converts a regular Stream to IntStream
  .average()
  .getAsDouble();

// collector
R collect(
  Supplier<R> supplier,
  BiConsumer<R, ? super T> accumulator,
  BiConsumer<R, R> combiner // only used for parallel streams
);

personsInAuditorum.stream().collect(
  () -> 0,
  (currentSum, person) -> { currentSum += person.getAge(); }.
  (leftSum, rightSum) -> { leftSum += rightSum; }
);


// parallel stream
someValues.parallelStream();
```

Design by Contract

Form of a Hoare triple $\{P\}\ C\ \{Q\}$

- P: precondition → specification what the supplier expects from the client
- C: series of statements → the method of body
- Q: postcondition → specification of what the client can expect from the supplier if the precondition is fulfilled

- client has to ensure that the precondition is fulfilled
- client can expect the postcondition to be fulfilled, if the precondition is
- **Non-Redundancy-Principle:** the body of a routine shall not test for the routine's precondition

```
/*@ requires size > 0;
  @ ensures size == \old(size) - 1;
  @ ensures \result == \old{top()};
  @ ensures true; // trivial constraint
  @*/
Object pop() { ... }
```

Liskov Substitution Principle

- preconditions must not be more restrictive than those of the overwritten method: $\texttt{Precondition}_{Super} \Rightarrow \texttt{Precondition}_{Sub}$
- postcondition must be at least as restrictive as thos of the overwritten methods: $\texttt{Postcondition}_{Sub} \Rightarrow \texttt{Postcondition}_{Super}$

Compiler

Basics

- **Lexikalische Analyse:**
  - Eingabe: Sequenz von Zeichen
  - Aufgaben:
    - erkenne bedeutungstragende Zeichengruppen: Tokens
    - überspringe unwichtige Zeichen (Leerzeichen, Kommentare, ...)
    - bezeichner identifizieren und zusammenfassen in Stringtabelle
  - Ausgabe: Sequenz von Tokens
- **Syntaktische Analyse:**
  - Eingabe: Sequenz von Tokens
  - Aufgaben:
    - überprüfe, ob Eingabe zu kontexfreier Sprache gehört
    - erkenne hierachische Struktur der Eingabe
  - Ausgabe: Abstrakter Syntaxbaum (AST)
- **Semantische Analyse:**
  - Eingabe: Syntax Baum
  - Aufgaben: kontextsensitive Analyse (syntaktische Analyse ist kontextfrei)
    - Namensanalyse: Beziehung zwischen Deklaration und Verwendung
    - Typanalyse: Bestimme und prüfe Typen von Variablen, Funktionen, ...
    - Konsistenzprüfung: Alle Einschränkungen der Programmiersprache eingehalten
  - Ausgabe: Attributierter Syntaxbaum
  - ungültige Programme werden spätestens in Semantischer Analyse abgelehnt
- **Codegenerierung:**
  - Eingabe: Attributierter Syntaxbaum oder Zwischencode
  - Aufgaben: Erzeuge Code für Zielmaschine
  - Ausgabe: Program in Assembler oder Maschinencode

Linksrekursion

- Linksrekursive kontextfreie Grammatiken sind für kein k SLL(k).
- Für jede kontextfreie Grammatik G mit linksrekursiven Produktionen gibt es eine kontextfreie Grammatik G' ohne Linksrekursion mit $L(G) = L(G')$

Java Bytecode

General

```
// this list partly is stolen from some guy on discord, but I forgot which one
// types
i -> int
l -> long
s -> short
b -> byte
c -> char
f -> float
d -> double
a -> reference

// load constants on the stack
aconst_null // null object
dconst_0 // double 0
dconst_1 // double 1
fconst_0 ... fconst_2 // float 0 to 2
iconst_0 ... iconst_5 // integer 0 to 5

// push immediates
bipush i // push signed byte i on the stack
sipush i // push signed short i on the stack

// variables (X should be replaced by a type, for example i (integer))
// there exists Xload_i for i in [0, 3] to save a few bytes
Xload i // load local variable i (is a number)
Xstore i // store local variable i

// return from function
return // void return
Xreturn // return value of type X

// conditional jumps
if_icmpeq label // jump if ints are equal
if_icmpge label // jump if first int is >=
if_icmpgt label // jump if first int is >
if_icmple label // jump if first int is <
if_icmplt label // jump if first int is <=

ifeq label // jump if = zero
ifge label // jump if >= zero
ifgt label // jump if > zero
iflt label // jump if < zero
ifle label // jump if <= zero
ifne label // jump if != zero

ifnull label // jump if null
ifnonnull label // jump if not null

// Arithmetic, always operates on stack
iinc var const // increment variable var (number) by const (immediate)
isub // Integer subtraction
iadd // Integer addition
imul // Integer multiplication
```

```
idiv // Integer division
ineg // negate int
ishl // shift left (arith)
ishr // shift right (arith)

// Logic (für [i, l])
iand // Bitwise and
ior // Bitwise or
ixor // Bitwise or

// Method calls. Stack: [objref, arg1, arg2] <-
invokevirtual #desc // call method specified in desc
invokespecial #desc // call constructor
invokeinterface #desc // call method on interface
invokestatic #desc // call static method (no objref)

// Misc
nop // No operation

// Arrays
newarray T // new array of type T
Xaload // load type X from array [Stack: arr, index] <-
Xastore // store type X in array [Stack: arr, index, val] <-
arraylength // length of array
```

Examples

Arithmetic

Java:

```java
void calc(int x, int y) {
  int z = 4;
  z = y * z + x;
}
```

Bytecode:

```
iconst_4 // lege eine 4 auf den stack
istore_3 // pop stack und speichere Wert in Variable 3 (z)
iload_2 // lade Variable 2 (y) und lege sie auf den stack
iload_3 // lade Variable 3 (z) und lege sie auf den stack
imul // multipliziere die oberen zwei elemente und lege das ergebnis auf den stack (y * z)
iload_1 // lade Variable 1 (x) und lege sie auf den Stack
iadd // addiere die oberen zwei Elemente und lege sie auf den stack
istore_1 // pop stack und speichere Wert in Variable 3 (z)
```

Loops

Java:

```java
public int fib(int steps) {
  int last0 = 1;
  int last1 = 1;
  while (--steps > 0) {
    int t = last0 + last1;
    last1 = last0;
    last0 = t;
  }
}
```

Bytecode:

```
  iconst_1 // put 1 on stack
  istore_2 // store top of stack in var 2
  iconst_1 // put 1 on stack
  istore_3 // store top of stack in var 3

loop_begin: // label
  iinc 1 -1 // increment var 1 by -1
  iload_1 // load var 1 and put on stack
  ifle after_loop // if top of stack <= 0, jump to after_loop
  iload_2 // put var 2 on stack
  iload_3 // put bar 3 on stack
  iadd // add top two elements and put on stack
  istore 4 // store top of stack in var 4
  iload_2 // load var 2 and put on stack
  istore_3 // store top of stack in var 3
  iload 4 // load var 4 and put on stack
  istore_2 // store top of stack in var 2
  goto loop_begin // jump to loop_begin
after_loop: // label
  iload_2 // load var 2 and put on stack
  ireturn // return top of stack
```