

# Programmiersprachen

Lernskript

Dozent:  
Dr. Andreas Karwath

L<sup>A</sup>T<sub>E</sub>X von:  
Sven Bamberger

Zuletzt Aktualisiert:  
6. August 2014



JOHANNES GUTENBERG  
UNIVERSITÄT MAINZ



Dieses Skript wurde erstellt, um sich besser auf die Klausur vorzubereiten.

Dieses Dokument garantiert weder Richtigkeit noch Vollständigkeit, da es aus Mitschriften und Vorlesungsfolien gefertigt wurde und dabei immer Fehler entstehen können. Falls ein Fehler enthalten ist, bitte melden oder selbst korrigieren und neu hoch laden.



# Inhaltsverzeichnis

<b>1</b>	<b>Haskell</b>	<b>1</b>
1.1	Allgemein . . . . .	1
1.2	Input, Main, Output . . . . .	1
1.2.1	IO . . . . .	1
1.3	Funktionsdeklaration . . . . .	2
1.4	Notation . . . . .	3
1.4.1	Logik . . . . .	3
1.4.2	Potenz . . . . .	3
1.4.3	Listen . . . . .	3
1.4.4	Strings $\Rightarrow$ Liste aus Char . . . . .	4
1.4.5	Tupel . . . . .	4
1.4.6	Beispiele . . . . .	4
1.5	Currying . . . . .	5
1.6	if .. then .. else und Guards . . . . .	5
1.7	Typen und Typ Klassen . . . . .	5
1.7.1	Standard Typen . . . . .	5
1.7.2	Standard Typ Klassen . . . . .	6
1.7.3	Eigene Typen . . . . .	6
1.8	Pattern Matching . . . . .	7
1.9	List Comprehension . . . . .	7
1.9.1	Beispiele . . . . .	7
1.10	Lambda ( $\lambda$ ) Funktionen . . . . .	8
1.11	Higher Order Functions . . . . .	8
1.12	Lazy Evaluation . . . . .	9
<b>2</b>	<b>Prolog</b>	<b>11</b>
2.0.1	Atome . . . . .	11
2.0.2	Variablen . . . . .	11
2.1	Notation . . . . .	12
2.1.1	Und / Oder in Regeln . . . . .	12
2.1.2	Numerische Vergleichsoperatoren: . . . . .	12
2.1.3	Numerischer Auswertungsoperator: . . . . .	12
2.1.4	Term-Vergleichsoperatoren: . . . . .	12
2.1.5	Numerische Operatoren . . . . .	12
2.1.6	Typtest (nice to know but not necessary) . . . . .	13
2.2	Wissensbasis . . . . .	13
2.3	Rekursionen . . . . .	14
2.4	Unifikation . . . . .	14
2.5	Resolution . . . . .	15
2.5.1	Was ist Resolution? . . . . .	15
2.5.2	Wie funktioniert Resolution ? . . . . .	15
2.5.3	Ein Beispiel zur Vorgehensweise . . . . .	16
2.6	Akkumulatoren . . . . .	16

## Inhaltsverzeichnis

2.7	Cut . . . . .	17
2.7.1	Wirkung: . . . . .	17
2.7.2	Arten: . . . . .	17
2.7.3	Grüner Cut . . . . .	17
2.7.4	Roter Cut . . . . .	18
2.8	Zustände . . . . .	18
<b>3</b>	<b>C/C++</b>	<b>19</b>
3.1	Programme Kompilieren . . . . .	19
3.1.1	C++ Programme bauen . . . . .	19
3.1.2	Wie lauten die Terminal Befehle dafür? . . . . .	20
3.2	Basics . . . . .	21
3.2.1	Variablen . . . . .	21
3.2.2	Arithmetische Operatoren . . . . .	21
3.2.3	Increment / Decrement . . . . .	21
3.2.4	Typen . . . . .	21
3.2.5	Logik Operationen . . . . .	22
3.2.6	Vergleiche . . . . .	22
3.2.7	Funktionen . . . . .	22
3.3	Kommandozeilen Argumente . . . . .	24
3.4	I/O from console/file . . . . .	24
3.4.1	I/O from console . . . . .	24
3.4.2	I/O from File . . . . .	25
3.5	Control Statements and Loops . . . . .	26
3.5.1	if .. then .. else / Switch . . . . .	26
3.5.2	Loops . . . . .	26
3.6	Call by Value / Call by Reference . . . . .	26
3.7	Pointer und Referenzen . . . . .	27
3.7.1	Referenzen . . . . .	27
3.7.2	Pointer . . . . .	27
3.7.3	Stolperfallen . . . . .	28
3.8	Memmmory Management (Stack/Freestore) . . . . .	29
3.8.1	Stack . . . . .	29
3.8.2	FreeStore (Heap) . . . . .	30
3.9	Structs . . . . .	30
3.10	Complex Data Structures . . . . .	31
3.11	Standard Containers and Algorithms . . . . .	32
3.11.1	Algorithmen . . . . .	33
3.11.2	Includes . . . . .	33
3.11.3	Sequentielle Container . . . . .	33
3.11.4	Assoziative Container . . . . .	34

# 1 Haskell

## 1.1 Allgemein

Haskell ist eine Funktionale Programmiersprache und ist daher um einiges anders als die bisher gelernten Objektorientierten Sprachen. Diese verwenden verstärkt funktionale Eigenschaften um sich weiter zu entwickeln. Funktionale Sprachen sind frei von Seiteneffekten und die Methoden können sich gegenseitig nicht beeinflussen. Eine Methode liefert bei dem gleichen Input immer den gleichen Output. In diesem Dokument werden verstärkt Codebeispiele genutzt um die Sprache kurz zu erläutern.

## 1.2 Input, Main, Output

Zunächst ein kleines "Hello World" Beispiel, wie es zu jeder Sprache gehört. Hier sieht man direkt wie die main von Haskell aufgebaut ist.

```
main = putStrLn "Hello World"
```

Um ein kleines Programm zu schreiben, was mehr tut, als lediglich Sachen auszugeben, müssen wir Input verarbeiten. Dafür ist Haskell nie wirklich vorgesehen worden, da dies unvorhergesehenes Verhalten erzeugen kann, welche Haskell vermeiden soll. Daher sollte man dies nur verwenden, wenn es gefordert oder nicht anders möglich ist.

```
main = do
  print "What is your name?"
  Name <- getLine
  print ("Hello " ++ Name ++ "!")
  -- main
```

Wie man sieht, kann man den Input in eine vorher nicht definierte Variable geben. In Haskell muss man Variablen nicht vorher genauer definieren. Der Compiler versucht zur Compilezeit zu ermitteln, welchen Typ die Variable hat und setzt diesen fest. Wenn dies nicht möglich ist, wird die Variable zur Laufzeit erstellt und definiert und genau das kann zu unvorhergesehenem Verhalten führen.

### 1.2.1 IO

IO Programme aus der Fragestunde, härter als das wird es aller Voraussicht nach nicht werden.

```
putToConsoleAndCheck :: String -> IO Bool
putToConsoleAndCheck s = do
  if s == "" then return False
  else do
    putStrLn s
    return True
```

```
putToConsoleAndReturnNothing :: String -> IO ()
putToConsoleAndReturnNothing s = do
    if s == "" then return ()
    else do
        putStrLn s
        return ()
```

## 1.3 Funktionsdeklaration

Eine Funktion in Haskell sieht ungewohnt aus und erinnert mich immer stark an Mathe. Wobei die späteren Listen Verarbeitungsmöglichkeiten noch stärker an die Mathematik angelehnt sind.

```
f :: Int -> Int -> Int
f x y = x*x + y*y
```

dies ist alles was benötigt wird um eine Funktion zu deklarieren. Vom Schreibaufwand ist dies sehr gering. Jedoch ist es ungewohnt eine Funktion so zu sehen. Also nehmen wir diese einmal komplett auseinander.

Funktionsname $\underbrace{f}$	$\underbrace{x \ y}$	=	was die Funktion macht $\underbrace{x * x + y * y}$
	Variablennamen der Funktion		

Es sieht zwar nicht gerade überragend aus, jedoch ist hier schon sehr genau erklärt wie eine Funktion aufgebaut ist.  $f :: Int \rightarrow Int \rightarrow Int$  ist die sogenannte Funktionstypendeklaration das erste  $Int$  steht für das  $x$  das zweite für  $y$  und das dritte für den Typ des Ergebnisses. Es ist zwar nicht notwendig dies anzugeben, jedoch fördert es das Verständnis und hilft einem dabei nochmals genau über die Funktion nachzudenken. Da diese Methode jedoch auch gut für alle Zahlen geeignet ist, sollte man den Funktionstypen undefinieren

```
f :: Num a => a -> a -> a
```

Das sind nochmals kryptischer aus, jedoch bedeutet dies nur, dass die Funktion  $f$  eine Zahl ( $Num a \Rightarrow$ ) vom Typ  $Num$  erwartet. Danach hätte es gerne noch eine Zahl und gibt eine Zahl zurück. Welche Typen diese Zahlen haben, wird zur Compilezeit definiert und entsprechend angewandt. Das  $a \rightarrow a \rightarrow a$  bedeutet, dass alle vom Typ  $Num$  sein müssen. Also wenn wir als erstes ein  $Int$  eingeben, werden alle  $a$  zu einem  $Int$  und es werden nur noch  $Int$ 's erwartet. Vorteil, wir können eine Funktion für unterschiedliche Zahlentypen verwenden, ohne diese jedes mal neu zu definieren.

Auch können Funktionen weitere Funktionen aufrufen.  $g$  ist in diesem Fall eine Section, da  $g$  einen Parameter von  $f$  vordefiniert.

```
f :: Num a => a -> a -> a
f x y = x*x + y*y
```

```
g :: Num a => a -> a -> a
g = f 3
```



Wie funktioniert nun  $g$ ? diese Funktion ruft nun  $f$  mit dem Wert 3 auf und  $f$  gibt sich die Funktion  $f$  zurück um noch einen Wert entgegen zu nehmen. Dies bezeichnet man auch als Currying. Um zu verhindern, dass Haskell weiterhin die Typ Transformationen von Variablen übernimmt, kann man diese vorher definieren. Davon wird abgeraten, da dies zu unvorhergesehen Fehlern führen kann. z.B. wird der nach folgende Code nie erfolgreich ausgeführt werden.

```
x :: Int
x = 3

y :: Float
y = 2,4

print (f x)
```

## 1.4 Notation

### 1.4.1 Logik

$\parallel \Rightarrow$  or     $\&\& \Rightarrow$  and     $== \Rightarrow$  Gleichheit     $/= \Rightarrow$  Differenz

### 1.4.2 Potenz

$x^n$      $x^{**}y$

Hierbei gilt es zu beachten, dass  $n$  ein Int || Integer sein muss, und  $y$  irgend eine Variable vom Typ Num sein darf.

### 1.4.3 Listen

- sind getypt, es können lediglich Elemente eines Typs in einer Liste vorkommen (Mischen von Typen ist nicht möglich). Dies ergänzt sich mit Tupeln: Listen sind theoretisch beliebig lange Folgen von Elementen eines Typs, Tupel sind definiert lange Folgen von Elementen, die verschiedene Typen haben können. Typsignatur von Listen: z.B. [Int], [Char], usw.
- können polymorph sein, d.h., der Typ einer Liste kann parametrisiert werden. Typsignatur in diesem Fall: z.B. [a]
- können per Aufzählung der Elemente definiert werden (Beispiel: [1,2,3]) oder mittels des Listenkonstruktors (Beispiel: 1:(2:(3)) ).
- werden per Rekursion ( $x:xs$  beschreibt das erste Element  $x$  einer Liste und die Restliste  $xs$ ) oder per Indizierung (Operator  $!!$ ) verarbeitet (die Indizierung beginnt bei 0)
- können mittels Erzeugungsschemata beschrieben werden (list comprehensions): z.B. [ transform neueListe | neueListe <- alteListe, tollerTest neueListe ], in dem die Elemente einer existierenden Liste traversiert, optional mit Prädikaten gefiltert und ebenfalls optional durch weitere Funktionen transformiert in eine neue Liste übernommen werden. In diese Rubrik fällt auch eine abgekürzte Schreibweise für Aufzählungen: z.B. [1..10] um eine Liste mit Elementen von 1 bis 10 zu erzeugen.

## 1 Haskell

- können dank Haskell's Lazy Evaluation Strategie unendlich lang sein (ihrer Beschreibung nach), wie z.B. bei [1..]. Solche Listen können verwendet werden, solange für den gewünschten Zweck die Auswertung nur eines Teils der Liste ausreicht.

### Beispiele

[ ]  $\Leftrightarrow$  leere Liste  
[1,2,3]  $\Leftrightarrow$  Liste von integralen  
["foo", "bar", "baz"]  $\Leftrightarrow$  Liste aus Strings  
1:[2,3]  $\Leftrightarrow$  [1,2,3] : ist das Symbol zum vorne anhängen an eine Liste  
1:2:[ ]  $\Leftrightarrow$  [1,2]  
[1,2]++[3,4]  $\Leftrightarrow$  [1,2,3,4] ++ ist für das konkatinieren zuständig  
[1,2,3]++["foo"]  $\Leftrightarrow$  Error, die Inhalte einer Liste müssen immer vom selben Typ sein.  
[1..4]  $\Leftrightarrow$  [1,2,3,4]  
[1,3, .. 10]  $\Leftrightarrow$  [1,3,5,7,9]  
[2,3,5,7,11 ... 100]  $\Leftrightarrow$  Haskell ist leider nicht so klug ;-)  
[10,9 ..1]  $\Leftrightarrow$  [10,9,8,7,6,5,4,3,2,1]

#### 1.4.4 Strings $\Rightarrow$ Liste aus Char

'a' :: Char  
"a" :: [Char]  
""  $\Leftrightarrow$  []  
"ab"  $\Leftrightarrow$  ['a','b']  $\Leftrightarrow$  'a':"b"  $\Leftrightarrow$  'a':['b']  $\Leftrightarrow$  'a':'b':[]  
"abc"  $\Leftrightarrow$  "ab"++ "c"

#### 1.4.5 Tupel

Tupel können unterschiedliche Typen enthalten und beliebig groß sein, jedoch funktionieren die Standard Implementationen lediglich auf Tupel der Größe 2

#### 1.4.6 Beispiele

##### Tupelkonstruktionen

(2, "foo") Unterschiedliche Typen sind legitim in Tupel  
(3, 'a', [2,3]) Listen sind auch möglich in Tupel  
((2, "a"), "c", 3) Auch können Tupel in Tupel geschrieben werden

##### Standardimplementationen

fst(x,y)  $\Rightarrow$  x  
fst(x,y,z)  $\Rightarrow$  Error, zu viele Elemente  
snd(x,y)  $\Rightarrow$  y  
snd(x,y,z)  $\Rightarrow$  Error, zu viele Elemente

## 1.5 Currying

Grundsätzlich sind alle Funktionen in Haskell curried. Curried bedeutet, dass die Funktion eine Funktion zurück liefert welche eine weitere Variable erwartet.

```
f :: Num a => a -> a -> a
f x = x*x + y*y
```

Bei diesem Beispiel wird die Funktion mit `f 4 5` aufgerufen. Als erstes Ergebnis liefert die Funktion sich selbst zurück und nimmt die 5 als `y` auf. Dadurch ist es möglich Funktionen kürzer zu gestalten. In den folgenden Beispielen als erstes die gecurriede Version und direkt im Anschluss die nicht gecurriede Version.

Klassisches Currying

```
sum :: Num a => a -> (a -> a)
sum x y = x + y
```

Klassische ugecurriede Version

```
sum :: Num a => (a,a) -> a
sum (x,y) = x + y
```

## 1.6 if .. then .. else und Guards

Hier einfach ein Beispiel der gleichen Funktion einmal mit `if .. then .. else` Konstrukt und einmal als Guards (diese erinnern mich immer sehr stark an Switches) welche auch mit Pattern Matching zusammenarbeiten. Bei einem `if .. then .. else` Konstrukt muss immer ein `else` Teil angegeben werden. Da der Compiler sonst streikt. Denn Funktionen sollen ohne Nebeneffekte ausgeführt werden können. Und ein Vergessenes `else` führt zu nicht vorhersagbaren Nebeneffekten.

```
absolute :: (Ord a, Num a) => a -> a
absolute x = if x >= 0 then x else -x
```

```
absolute' :: (Ord a, Num a) => a -> a
absolute' x
  | x >= 0 = x
  | otherwise = -x
```

## 1.7 Typen und Typ Klassen

### 1.7.1 Standard Typen

**Int** ist ein ganzzahliger, an das System gebundener Wert. 32-Bit / 64-Bit

**Integer** ist auch ein ganzzahliger, nicht an das System gebundener Wert, es kann so groß werden, wie

## 1 Haskell

der Speicher groß ist.

**Float** eine echte Gleitkommazahl mit einfacher Genauigkeit.

**Double** eine echte Gleitkommazahl mit doppelter Genauigkeit.

**Bool** Boolean halt.

**Char** ein einzelnes Zeichen in Einfachen Anführungszeichen.

### 1.7.2 Standard Typ Klassen

#### **Eq**

`(==) :: (Eq a) => a -> a -> Bool`

#### **Ord**

`(>) :: (Ord a) => a -> a -> Bool`

**Show** wandelt Zahlen und boolische Ausdrücke in Strings um

`show :: Show a => a -> String`

**Read** wandelt Strings in Zahlen oder boolische Ausdrücke um.

`read :: (Read a) => String -> a`

**Num** ist eine numerische Typ Klasse. Ihr Mitglieder lasen sich wie Zahlen benutzen und verhalten sich so.

**Enum** ist eine Typ Klasse, welche alle auf zählbaren / durchzunummerierende Sachen enthalten. dadurch lassen sich erst die unendlichen Listen darstellen. Auch können succ und pred als Funktionen auf dessen Mitglieder angewandt werden.

### 1.7.3 Eigene Typen

#### **type Name = AnotherType**

Damit kann man dem Kind AnotherType einen neuen Namen geben. Wenn man z.B. zwischen Nachname und Vorname unterscheiden möchte. Jedoch macht der Compiler hier keinen Unterschied.

#### **data Name = NameConstructor AnotherType**

Hier macht der Compiler eine Unterscheidung. Der genaue Unterschied zwischen beiden type und data wird gleich an einem Beispiel erläutert.

#### **data**

Damit kann man sich rekursive Strukturen definieren.

#### **deriving**

damit kann man sich Funktionen erzeugen lassen.

#### **Beispiele**

Bei dem nachfolgenden Beispiel, muss man sehr aufpassen, denn die Type und Data Version haben starke Unterschiede zur Laufzeit. Wenn man die Type Version nutzt, kann man als Eingabe Color und Name vertauschen und das Programm arbeitet trotzdem weiter, da es lediglich Strings erwartet.

Bei Data erwartet Haskell wirklich Datenstrukturen Name und Color. Wenn man hier zuerst Color und danach Name eingibt, meckert der Compiler.

```
Type Version
type Name    = String
type Color   = String

Data Version
data Name    = NameConstr String | ... | ...
-- /../.. sind weitere Typen moeglich jedoch nicht Datenstrukturen
data Color   = ColorConstr String

showInfos :: Name -> Color -> String
showInfos (NameConstr name) (ColorConstr color) =
  "Name:_ " ++ name ++ " ,_Color:_ " ++ color
```

Als nächstes wird eine Rekursive Datenstruktur definiert und mit deriving(Show) erweitert, was dazu führt, dass diese Datenstruktur direkt gedruckt werden kann.

```
data Tree a = EmptyTree | Node a (Tree a) (Tree a) deriving (Show, Read, Eq)
```

## 1.8 Pattern Matching

Beim Pattern Matching handelt es sich um die Möglichkeit in eine Funktion Abzweigungen einzubauen, aufgrund von bestimmten Inhalten einer Variable. Davon wird exzessiv in Prolog<sup>2</sup> gebraucht gemacht. Das Pattern Matching ist eine effiziente Art zu programmieren, wenn man wenige unterschiedliche Werte hat, auf die geprüft werden muss. Auch kann man damit gut Abbruchbedingungen erstellen für rekursive Funktionen. Dabei gilt zu beachten, dass von oben nach unten gearbeitet wird.

```
factorial :: (Integral a) => a -> a
factorial 0 = 1
factorial n = n * factorial (n - 1)
```

## 1.9 List Comprehension

Mit List Comprehensions kann man einfach Listen nach zuvor definierten Regeln erstellen. Dies ist besonders nützlich, um viele Werte schnell zu generieren.

### 1.9.1 Beispiele

```
[x^2|x <- [1..5]]
Ergebnis: [1,4,9,16,25]
```

Der Bereich nach | ist der sogenannte Generatorbereich und wird dazu genutzt, die Regeln für das erstellen der einzelnen Variablen zu definieren.

```
[(x,y)|x <- [1,2,3],y <- [4,5]]
Ergebnis: [(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
```

## 1 Haskell

Man kann auch mehrere Variablen im Generator Bereich verwenden und dadurch eine Liste von Tupeln generieren wie im oberen Beispiel gezeigt. Jedoch sollte man sehr auf die Reihenfolge der Generatoren achten, da diese das Ergebnis stark beeinflussen.

```
[(x,y)|x <- [1,2,3],y <- [x..3]] -- weitere Kontrollen wie even x moeglich
-- durch Kommata abgetrennt
Ergebnis: [(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

Hier werden Dependant Generators verwendet. Die zu erzeugenden Werte für  $y$  hängen von  $x$  ab. Dadurch kann man sehr gut doppelte Werte vermeiden und andere Spielereien machen.

## 1.10 Lambda ( $\lambda$ ) Funktionen

Lambda Funktionen sind Namenlose Funktionen welche lediglich einmal verwendet werden. Dies macht es möglich Funktionen in Methoden zu definieren und hat unter anderem auch Einzug in Java gefunden. Lambda Funktionen werden mit einem  $\lambda$  eingeleitet. Hier ein Beispiel für eine einfache Addition.

```
(\x y -> x+y) 4 3
Ergebnis: 7
```

Auf diese Art und Weise kann man sich beliebige Funktionen überall einbauen. Wodurch besonders foldl und foldr stark werden.

## 1.11 Higher Order Functions

Als Higher Order Functions werden Funktionen bezeichnet welche eine Funktion als erstes Argument entgegennehmen.

### map

nimmt eine Funktion und eine Liste entgegen und verwendet diese Funktion auf jedes Element der Liste und generiert so eine neue.

```
map :: (a -> b) -> [a] -> [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

### filter

ist eine Funktion welche eine Funktion entgegen nimmt, welche einen booleschen Wert zurück gibt. z.B. even. Dazu nimmt es noch eine Liste auf. Das Ergebnis ist eine Liste, welche zu die Übergebene Funktion mit true ergibt.

```
filter :: (a -> Bool) -> [a] -> [a]
filter _ [] = []
filter p (x:xs)
| p x      = x : filter p xs
| otherwise = filter p xs
```

### foldl

`foldl` wendet eine Funktion von **links** beginnend auf eine gesamte Liste an.

```
foldl (\x y -> x^2 + y) 0 [1,2,3]
Ergebnis: 12
```

Was genau vor sich geht:

```
foldl (\x y -> x^2 + y) 0 [1,2,3]
foldl (\x y -> x^2 + y) (0^2+1) [2,3]
foldl (\x y -> x^2 + y) (1^2+2) [3]
foldl (\x y -> x^2 + y) (3^2+3) []
foldl (\x y -> x^2 + y) 12 []
```

### **foldr**

`foldr` wendet eine Funktion von **rechts** beginnend auf eine gesamte Liste an.

```
foldr (\x y -> x^2 + y) 0 [1,2,3]
Ergebnis: 14
```

Was genau vor sich geht:

```
foldr (\x y -> x^2 + y) 0 [1,2,3]
foldr (\x y -> x^2 + y) (3^2+0) [1,2]
foldr (\x y -> x^2 + y) (2^2+9) [1]
foldr (\x y -> x^2 + y) (1^2+13) []
foldr (\x y -> x^2 + y) 14 []
```

## 1.12 Lazy Evaluation

Haskell ist Faul! es berechnet nie das Ergebnis einer Funktion komplett sondern erst, wenn dies notwendig ist. Erst dadurch ist es möglich die Ranges zu verwenden um unendliche Listen zu erstellen. Es gibt zwei Methoden der Reduktion um Funktionen abzuarbeiten Innermost und Outermost. Haskell verwendet Outermost, da dies auch terminiert, sofern Innermost terminiert und es bleibt nicht in einer Endlosschleife hängen. Jedoch ist Outermost Reduktion ineffizient und benötigt mehr Schritte. Um diesen Fehler zu umgehen benutzt Haskell Sharing. Dadurch wird es genauso effizient wie Innermost Reduktion und endet nicht in einer Endlosschleife.

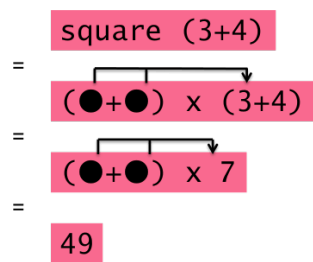


Abbildung 1.1: Sharing + Outermost



## 2 Prolog

In Prolog dreht sich alles um das erstellen von Wissensbasen und deren Auswertung. Fakten und Regeln zusammen ergeben eine Wissensbasis und sind zusammen mit Anfragen die 3 Konstrukte aus denen sich diese Sprache zusammensetzt. Regeln und Fakten gehören zu den Prädikaten. Fakten können weiterhin unterteilt werden in Funktoren und Atome. Dazu das passende Bild.

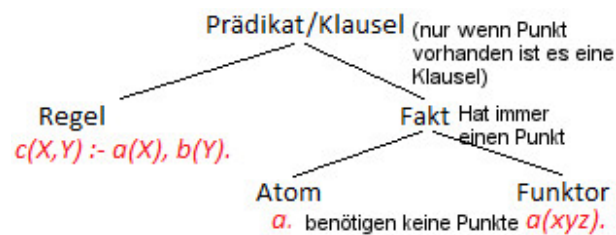


Abbildung 2.1: Prädikat Unterteilung

### 2.0.1 Atome

- fängt mit einem a-z
- bestehen aus den Zeichen a-z, A-Z, 0-9, \_
- Ein String in ' '. z.B. 'Hi', auch als Name des Atoms bezeichnet. Leerzeichen sind innerhalb dieser Anführungszeichen erlaubt.
- Spezielle Zeichenketten sind erlaubt. Diese sind:
  1. @= oder == oder ; oder :- . Jedoch fallen ; und :- Sonderrollen zu. Mit ; fragt man nach weiteren Antworten von dem Programm und :- gehört fest zu einer Regel.

Hinweis: YAP und andere Prolog Interpreter erkennen 'mia' = mia. An und geben Yes aus. Jedoch '2' = 2. wird immer mit einem No konfrontiert.

### 2.0.2 Variablen

- Variablen beginnen mit A-Z oder \_ (mit \_ sind Anonyme Variablen)
- bestehen aus den Zeichen a-z, A-Z, 0-9, \_

## 2.1 Notation

Im Nachfolgenden gibt es einen kleinen Überblick über die Notationen in Prolog.

### 2.1.1 Und / Oder in Regeln

,  $\Leftrightarrow$  Und  
;  $\Leftrightarrow$  Oder

### 2.1.2 Numerische Vergleichsoperatoren:

$X \text{ } =:= \text{ } Y$  numerisch gleich  
 $X \text{ } = \backslash = \text{ } Y$  numerisch ungleich  
 $X < Y$  X kleiner als Y  
 $X > Y$  X größer als Y  
 $X \text{ } =< \text{ } Y$  X kleiner gleich Y  
 $X \text{ } >= \text{ } Y$  X größer gleich Y

!! Bei Vergleichen müssen alle Variablen belegt sein.

### 2.1.3 Numerischer Auswertungsoperator:

Ein einfaches Beispiel um X Berechnete Werte zu übergeben.

X is 6+3.

Achtung, wenn man X is Y + Z. schreibt, müssen Y und Z bereits belegt sein. Ansonsten führt dies zu einem Error.

### 2.1.4 Term-Vergleichsoperatoren:

$X = Y$  unifizierbar  
 $X \backslash = Y$  nicht unifizierbar  
 $X == Y$  identisch mit  
 $X \backslash == Y$  nicht identisch mit

### 2.1.5 Numerische Operatoren

+ Addition  
- Subtraktion  
\* Multiplikation  
/ Division

```
// Ganzzahl-Division
** Potenz
mod Modulo
/\ bit-weises UND (& in C und C++)
\| bit-weises ODER (| in C und C++)
```

### 2.1.6 Typtest (nice to know but not necessary)

Dies dient zur Bestimmung des Types der einzelnen Atome.

atom/1 Is the argument an atom?

integer/1 Is the argument an integer?

float/1 Is the argument a floating point number?

number/1 Is the argument an integer or a floating point number?

atomic/1 Is the argument a constant? no var, or term.

var/1 Is the argument an uninstantiated variable?

nonvar/1 Is the argument an instantiated variable or another term that is not an un instantiated variable?

Dabei gilt besonders folgendes zu beachten bei dem Test auf atom.

Erst instanziiert dann getestet.

```
?- X = a, atom(X).
```

```
X = a
```

```
yes
```

Erst getestet dann instanziiert.

```
?- atom(X), X = a.
```

```
no
```

## 2.2 Wissensbasis

Als Wissensbasis bezeichnet man eine Zusammenstellung von Fakten und Regeln. Im folgenden Beispiel ist alles vorhanden.

```
happy(yolanda).
listens2Music(mia).
listens2Music(yolanda):- happy(yolanda).
playsAirGuitar(X):- listens2Music(X).
```

Es gibt insgesamt 2 Fakten in dieser Wissensbasis. happy(yolanda) und listens2Music(mia). Diese Fakten werden auch Faktoren genannt, da diese ein Atom einklammern. Ein Reiner Fakt würde z.B. yolanda sein.

Die letzten Beiden Zeilen in diesem Beispiel sind Regeln. diese zeichnen sich durch :- aus und können auch Variable gehalten werden. Die erste Regel (listens2Music(yolanda):- happy(yolanda).) wird per Pattern Matching (siehe dazu auch das Pattern Matching in Haskell<sup>1,8</sup>) lediglich auf die Anfrage ?-listens2Music(yolanda) reagieren. Alle anderen Fälle fängt die zweite Regel ab. Hier werden die Eingaben, mit X Unifiziert<sup>2,4</sup>.

Die Regel splittet sich in einen Kopf und einen Körper auf. Alles was vor dem :- steht ist der Kopf und alles was danach steht ist der Körper.

## 2.3 Rekursionen

Auch in Prolog gibt es Rekursionen und diese werden stark genutzt. Damit können zum Beispiel Listen durchgearbeitet werden um zu schauen, wie lange eine Liste ist. Listen können hier beliebige Datentypen enthalten und sind nicht Typisiert wie in Haskell.

```
len([],0) .
len([_|T],N):-len(T,X) ,N is X + 1.
```

Aufruf:  
?- len([1,2,3,4],X).

Ergebnis:  
X = 4?  
yes

Nehmen wir diese Rekursion einmal auseinander.

Die erste Zeile in diesem Beispiel ist die Abbruch Bedingung. Diese wird wahr, wenn die Liste leer ist und definiert die Länge der leeren Liste als 0. Zu dieser 0 wird jeweils durch Backtracking, dem zurück schreiten im Lösungsbaum, eine eins hinzu addiert. Dies ergibt die Länge der Liste.

Es ist **extrem wichtig**, dass die Abbruchbedingung als erste bei einer Rekursion genannt wird. Denn Prolog arbeitet von oben nach unten. Wenn wir zuerst die Rekursion (Zeile 2) geschrieben hätten würde dies zu einem Fehler führen.

Beginne bei einer Rekursion immer zuerst mit der Abbruchbedingung, und stelle den rekursiven Aufruf so weit wie möglich ans Ende der rekursiven Regel.

## 2.4 Unifikation

Das Ergebnis einer Unifikation ist eine Liste von Variablenersetzungen, die man vornehmen muss, um zwei Ausdrücke identisch zu machen. Diese Liste ist Zustandsfrei (für alle Zeiten gleich und unveränderlich). Prolog legt im Hintergrund jede Variable nach einem bestimmten Muster an. Dieses Muster ist `_5874` Wobei die Zahlen abweichen können.

Beispiel 1:  
eats(fred,tomatoes).  
eats(Whom,What) .

Beispiel 2:  
likes(jane,X).  
likes(X,jim).

Beispiel 3:  
f(foo,L).  
f(A1,A1).

Beispiel 4:  
X = t(X).

**Lösungen:**

1. Unifikation ist möglich da Whom = fred und What = tomatoes. Bis hier hin keine große Verwunderung.
2. Unifikation ist **nicht** möglich da X nicht gleichzeitig zwei Werten zugeordnet werden kann.
3. Unifikation ist möglich. Denn A1 wird mit foo unifiziert und da L auch eine Variable ist, kann diese auch mit A1 und somit mit foo unifiziert werden.
4. Unifikation wird endlos weiter geführt. Denn  $X = t(X) = t(t(X)) \dots$

## 2.5 Resolution

### 2.5.1 Was ist Resolution?

Resolution ist die Vorgehensweise von PROLOG bei der Lösung von Anfragen.  
Ob eine Zielanweisung zutrifft/erfüllbar ist wird durch den Widerspruch bewiesen.

### 2.5.2 Wie funktioniert Resolution ?

Grundvoraussetzung für eine korrekte Arbeitsweise der Resolution ist eine Wissensbasis mit vollständigen, nicht widersprüchlichen Formeln. Da das Resolutionsverfahren korrekt ist wird es immer aus einer Formelmengende einen Widerspruch ableiten wenn sich darin einer befindet.

Die Anfrage an eine Wissensbasis ist nichts anderes als dass eine negierte Formel in die bereits vorhandene Menge gebracht wird. Trifft die Anfrage zu, so findet die Resolution einen Widerspruch und beweist somit deren Richtigkeit.

Dabei benutzt die Resolution das sogenannte Ableitungsverfahren. Dieses regelt das Zusammenfassen von Klauseln und geht dabei so vor:

Wenn der Kopf der 1.Klausel im Körper der 2.Klausel vorkommt, so kann an dieser Stelle der Körper der 1.Klausel eingesetzt werden.

Zur Verdeutlichung:

1.Klausel:  $A :- B, C, D$

2.Klausel:  $G :- E, F, A, K$

ergibt

$G :- E, F, B, C, D, K$

PROLOG geht dabei von oben nach unten vor, wenn mehrere vergleichbare Formeln vorhanden sind, so wird die "oberste" genommen, bei Nichterfolg wird die nächste versucht.

Sind mehrere Formel durch UND verknüpft, so wird dabei von links nach rechts vorgegangen, solange eine Beantwortung möglich ist. Bei verschiedenen möglichen Antworten wird immer die zuerst gefundene ausgegeben.

### 2.5.3 Ein Beispiel zur Vorgehensweise

Gegebene Wissensbasis:

```
legs(X,2) :- mammal(X),arms(X,2).
legs(X,4) :- mammal(X), arms(X,0).
mammal(horse).
arms(horse,0).
```

Stellt man nun die Anfrage:

```
?- legs(horse,4).
```

geht PROLOG folgendermaßen vor:

diese Anfrage entspricht dem Kopf der 2. Regel, wenn man nun X mit horse unifiziert und den Kopf durch den Körper der Regel ersetzt erhält man folgendes

```
:- mammal(horse), arms(horse,0).
```

nun wird mammal(horse) (da es ganz links steht kommt es zuerst dran) mit dem Fakt mammal(horse). aus der Wissensbasis 'gestrichen'.

Ebendies passiert auch mit dem arms(horse,0) welches nach der ersten 'Streichung' geblieben ist und nun gestrichen wird.

Übrig bleibt die leere Klausel, d.h. der Widerspruch, womit die Zielanweisung als 'wahr' bewiesen wäre.

## 2.6 Akkumulatoren

Akkumulatoren ist eine tolle Möglichkeit Variablen hoch zählen zu lassen, während eine Liste durchgegangen wird. Hier ein kleines Beispiel zur Berechnung der Länge einer Liste.

```
accLen([],A,A).
accLen([_|T],A,L):- Anew is A + 1, accLen(T,Anew,L).
```

Was passiert.

```
?- accLen([a,b,c],0,L).
    Call: (6) accLen([a, b, c], 0, _G449) ?
    Call: (7) _G518 is 0+1 ?
    Exit: (7) 1 is 0+1 ?
    Call: (7) accLen([b, c], 1, _G449) ?
    Call: (8) _G521 is 1+1 ?
    Exit: (8) 2 is 1+1 ?
    Call: (8) accLen([c], 2, _G449) ?
    Call: (9) _G524 is 2+1 ?
    Exit: (9) 3 is 2+1 ?
    Call: (9) accLen([], 3, _G449) ?
    Exit: (9) accLen([], 3, 3) ?
    Exit: (8) accLen([c], 2, 3) ?
    Exit: (7) accLen([b, c], 1, 3) ?
    Exit: (6) accLen([a, b, c], 0, 3) ?
```

Da der Rückgabewert nicht nochmal berechnet werden muss während des Backtrackings, sondern direkt bekannt ist, sind Akkumulatoren Programme effizienter als andere.

Besser erkenntlich wird dies wenn man sich folgendes Beispiel anschaut und durch den Kopf gehen lässt Mit hilfe des vorherigen Beispiels.

```
accRev ([H|T], A, R) :- accRev (T, [H|A], R).
accRev ([], A, A).
rev (L, R) :- accRev (L, [], R).
```

## 2.7 Cut

Das Prädikat Cut (Symbol !) wird verwendet, um unnötiges Backtracking zu verhindern. Der Cut ist stets erfolgreich.

### 2.7.1 Wirkung:

Wenn Prolog bei seiner Beweissuche den Cut (!) erreicht, dann können Entscheidungen links von dem Cut nicht mehr rückgängig gemacht werden.

### 2.7.2 Arten:

Ein grüner Cut schneidet Zweige des Suchbaumes ab, die keine Lösungen enthalten.

Ein roter Cut schneidet Zweige des Suchbaumes ab, die Lösungen enthalten. Diesen Cut sollte man vermeiden.

### 2.7.3 Grüner Cut

```
max(X, Y, Z) :- X <= Y, !, Z=Y.
max(X, _, X).
?- max(2, 3, 2).
No.
```

Es wird kein Backtracking mehr betreiben um weitere unnötige Lösungen zu finden.

Dies war z.B. Hilfreich um bei dem Affen Beispiel in der Vorlesung den Affen, nachdem er die Banane hatte nicht noch weiter im Raum herumlaufen zu lassen.

### 2.7.4 Roter Cut

```
max(X,Y,Y) :- X =< Y, !.  
max(X,_,X).  
?- max(2,3,2).  
Yes.
```

Hier ist die Lösung offensichtlich falsch, denn  $3 \neq 2$ , jedoch wird daraufhin die zweite Regel angewandt, welche die Anfrage mit Yes beantwortet. Daher sind weitere Lösungsmöglichkeiten abgeschnitten.  
 $\Rightarrow$  roter Cut.

## 2.8 Zustände

Zustände werden genutzt um einen Weg (im besten Fall den kürzesten) zum Ziel zu beschreiben. Solche Zustände sind z.B. die Funktoren state bei der Monkey Aufgabe aus der Vorlesung. Anhand von diesen kann Prolog, mit Hilfe der Resolution, den Weg finden und das Ergebnis ausgeben.



## 3 C/C++

In der Vorlesung wurde mehr C als C++ gelehrt und gezielt auf fortgeschrittenen Konzepte wie Klassen, Exceptions, Templates, etc. verzichtet.

### 3.1 Programme Kompilieren

#### 3.1.1 C++ Programme bauen

Ein Programm in C++ kann aus mehreren Header Dateien (\*.h) und Quellcode Dateien (\*.cpp) bestehen. Um diese zu einem Lauffähigen Programm umzuwandeln. Benötigt man einen Compiler wie g++. Dieser Besteht aus einem Linker, Preprocessor und dem Compiler selbst.

Der Preprocessor nimmt alle in den Sourcecode verlinkten Header Dateien und fügt diese zu 100 % in den Sourcecode in dieser Stelle ein. Die Header Dateien können gleichzeitig in mehreren Sourcecode Dateien stecken.

Der Compiler wandelt anschließend die Sourcecode Dateien in Objectdateien um, welche vom Menschen nicht mehr gelesen werden können.

Der Linker fügt alle Object Dateien mit den verwendeten externen Bibliotheken zu einem ausführbaren Programm zusammen. Dabei werden auch alle Sprungadressen für Funktionsaufrufe gesetzt

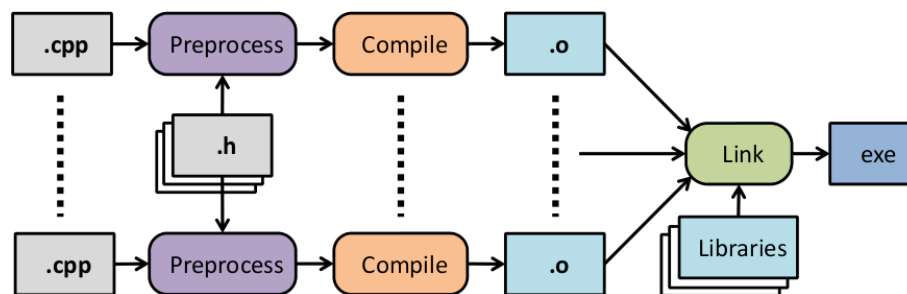


Abbildung 3.1: C++ Programm Kompilieren

#### Wozu dieses große Konstrukt?

Viele Projekte enthalten eine sehr große Menge an Quellcode Dateien. Da man nicht bei jeder kleinen Änderung alle cpp Dateien neu in Objekt Dateien umwandeln will. Kann man auch schlicht nur die geänderten cpp Dateien durch den Preprocessor jagen und anschließend alle .o Dateien neu zu verlinken um die Änderungen zu testen.

Außerdem können vorhandene doppelte #includes verhindert werden, indem man diese mit Guards definiert. Diese verhindern, dass die header Dateien, selbst bei mehrfacher Verlinkung, mehrfach in den Code einkopiert werden.

### 3 C/C++

```
//Datei: foo.h
#ifndef A_H
#define A_H
struct foo {
int member;
};
void f(foo&);
...
#endif
```

Selbst wenn man nun die foo.h header Datei in mehreren Dateien included, wird sie lediglich einmal hinzugefügt. Denn der Preprocessor, erkennt alle mit # markierte Zeilen als seine an und führt das if entsprechend der Definition aus. Dabei geht der Guard von #ifndef A\_H bis #endif. Das #define bedeutet, wenn A\_H noch nicht definiert wurde, definiere diese wie folgt.

#### 3.1.2 Wie lauten die Terminal Befehle dafür?

Die Terminal Befehle lauten wie folgt.

```
g++ -c source.cpp
g++ -c dbl.cpp
g++ source.o dbl.o -o test
```

Mit dem Parameter -c werden die angegebenen cpp Dateien in Object Files umgewandelt. Danach linkt man die Object Files mit Hilfe des Parameters -o. Man kann auch direkt -o verwenden wenn man die Dateien direkt in eine .exe Datei umwandeln will.

Um das Kompilieren zu vereinfachen, benutzt man bei vielen Dateien eine Makefile. Einen solche sieht wir folgt aus.

```
#name test
#comment
#compile main
main.o : main.cpp
g++ -c main.cpp
#compile other
other.o : other.cpp
g++ -c other.cpp
#link
prog: main.o other.o
g++ main.o other.o -o prog
```

Der Aufruf der Makefile läuft folgendermaßen

```
make test
./prog
```

Mit ./ führt man die generierte .exe aus.

## 3.2 Basics

### 3.2.1 Variablen

- fängt mit a-z oder `_` an.
- enthält a-z, A-Z, 0-9, `_`
- darf **nicht** mit einer Zahl beginnen.
- enthalten Werte eines festen Typs.
- muss deklariert werden, bevor diese verwendet werden kann. Am besten direkt mit einem Startwert deklarieren.
- nach Konventionen, muss die Variable mit einem kleinen Buchstaben beginnen.

### 3.2.2 Arithmetische Operatoren

- `+` Addition
- `+=` Addition mit Zuweisung
- `-` Subtraktion
- `-=` Subtraktion mit Zuweisung
- `*` Multiplikation
- `*=` Multiplikation mit Zuweisung
- `/` Division
- `/=` Division mit Zuweisung
- `%` Modulo

### 3.2.3 Increment / Decrement

	prefix	postfix
increment	<code>++a</code>	<code>a++</code>
decrement	<code>--a</code>	<code>a--</code>

Der Unterschied zwischen prefix und postfix ist, dass bei einer Zuweisung wie

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

der Wert von b erst erhöht wird und danach erst zugewiesen wird. Bei

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

wird der Wert erst zugewiesen und daraufhin erhöht.

### 3.2.4 Typen

Jedoch gilt es zu beachten das `long` und `int` meist gleich lang sind auf 32Bit Systemen und das der C++ Standard lediglich die Minimale Größe Garantiert.

Name	Represents	Usual Size	Usual Range
<b>bool</b>	truth value	1	<i>true</i> or <i>false</i>
<b>char</b>	(character)	1	-128 ... 127
unsigned char		1	0 ... 255
<b>short</b>	integer	2	-32768 ... 32767
unsigned short	integer	2	0 ... 65535
<b>int</b>	integer	4	-2147483648 ... 2147483647
unsigned	positive integer	4	0 ... 4294967295
<b>long</b>	integer	4	-2147483648 ... 2147483647
<b>long long</b> (C++11)	integer	8	-9223372036854775807 ... 9223372036854775807
unsigned long long	positive integer	8	0 ... 18446744073709551615
<b>float</b>	real number	4	+/-3.4e+/-38
<b>double</b>	real number	8	+/-1.7e+/-308

Abbildung 3.2: Alle Typen im Überblick

### 3.2.5 Logik Operationen

&& / and = Und  
 || / or = Oder  
 ! / not = Negation  
 0 = false  
 alles ≠ 0 = true

### 3.2.6 Vergleiche

== ; < ; > ; >= ; <= ; !=

### 3.2.7 Funktionen

```
return_type name (parameters) {
    //body
}
```

Die Main Funktion wird mit dem return\_type int versehen, da Programme in der Regel eine 0 zurückliefern, sofern diese ohne Probleme durchgelaufen sind.

Eine Funktion darf lediglich einen Rückgabewert enthalten. Um sich mehrere Rückgabewerte zu generieren bedient man sich eines Tricks, dieser heißt Call by Reference<sup>3.6</sup>. Dadurch arbeitet man auf den Variablen außerhalb der Methode.

Alle in der Main benutzten Methoden müssen vor der Main bereits bekannt sein. Dies kann man erreichen, indem man alle Funktionen bereits vor der Main definiert, oder diese vorher definiert und später mit leben füllt.

```
void odd (int a);
```

```
int main ()
{
    int i;
```

```

do {
    cout << "Type_a_number: \n";
    cin >> i;
    odd (i);
} while (i!=0);
}

void odd (int a)
{
    if ((a%2)!=0)
        cout << "Number_is_odd.\n";
    .
    .
    .
    .

```

Um Standard Parameter zu nutzen, (also Parameter, welche einen Wert haben, ohne übergeben worden zu sein.) muss man bei der Parameterliste diese initialisieren.

```

//Beispiel 1:
float mul(float a, float b = 1.5f) {
    return (a * b);
}

//Beispiel 2:
float mul(float a, float b = 1.5f, float c) {
    return (a * b * c);
}

```

Beispiel 1 funktioniert wie erwartet, jedoch wird Beispiel 2 nicht funktionieren, da die Variable c nicht initialisiert ist, obwohl diese nach einer bereits initialisierten kommt.

### Überladen

```

int abs (int i) {
    return ((i < 0) ? -i : i);
}
float abs (float i) {
    return ((i < 0) ? -i : i);
}
double abs (double i) {
    return ((i < 0) ? -i : i);
}
//-----
int foo (int i) {
    return (2 * i);
}
double foo (int i) {
    return (2.5 * i);
}
//-----
int foo (int i, int k = 2) {
    return (2 * i * k);
}
int foo (int i, int k = 2, int l = 3) {

```

### 3 C/C++

```
        return (2 * i * k * l);
    }
```

Das was im ersten Abschnitt steht funktioniert einwandfrei. Genau so sollte an Funktionen überladen. Der zweite Abschnitt funktioniert nicht, da nicht nur der Rückgabewert geändert werden kann, sondern auch die Parameterliste verändert werden muss.

Das Beispiel aus dem dritten Abschnitt funktioniert nicht wie erwartet. Denn der Compiler muss wissen können, welche Funktion er nimmt. Wenn man nun `foo(3)` aufruft, kann der Compiler nicht entscheiden, welche der Funktionen genutzt wird.

## 3.3 Kommandozeilen Argumente

Hier ein kleines Beispiel, wie man Kommandozeilen Argumente beim Start eines Programmes direkt übergibt, ausliest und auf der Konsole ausgibt.

```
#include <iostream>
```

```
using namespace standard
```

```
int main(int argc, char* argv[])
{
    for(int i = 1; i<argc;++i){
        cout << argv[i] <<endl;
    }
    return 0;
}
```

```
> ./test 12 3 abc def
test
12
3
abc
def
```

`argc` und `argv` sind nur Konventionen und können anders benannt sein. `argc` enthält die Anzahl der Elemente und `argv` ist ein C Array. Die Elemente von `argv` sind wiederum C Strings.

## 3.4 I/O from console/file

Damit das Programm auch von einem Benutzer gesteuert werden kann, Reichen meist die Kommandozeilenparameter<sup>3.3</sup> nicht aus.

### 3.4.1 I/O from console

Damit der Benutzer während des Programmablaufs Daten eingeben kann, fordert man ihn über die Konsole zu einer Eingabe auf.

```
#include <iostream>
using namespace std;
int main(){
    int i = 0;
    cout << "Geben_Sie_eine_Zahl_ein" << endl;
```

```

    cin >> i ;
    cout << "Sie_haben_" + i + "_eingegeben" << endl;
}

```

`#include <iostream>` ermöglicht es Text von der Kommandozeile zu lesen und in dieser wiederzugeben. `using namespace std;` ermöglicht es `cout`, `cin` und `endl`; so abgekürzt zu schreiben.

### 3.4.2 I/O from File

Um mit größeren Datenmengen zu arbeiten ist es sinnvoll Dateien auslesen zu können. Dies kann man z.B. mit folgendem Programm machen.

```

#include <fstream>
using namespace std;
int main(){
    // Datei lesen
    ifstream datei ("file.txt");
    if (datei.is_open()){
        while (!datei.eof()){
            string s;
            getline(datei,s);
            cout << s << endl;
        }
    }
    datei.close();
    // Datei schreiben
    ofstream output ("file_copy.txt");
    if (output.is_open()){
        for (int i = 0; i < 20; ++i){
            double i2 = i*i;
            output << i << '\t' << i2 << '\n';
        }
    }
    output.close();
    return 0;
}

```

Um Dateien zu öffnen gibt es unterschiedliche Modi.

- Default:
  - `ifstream is("in.txt", ios::in);`
  - `ofstream os("out.txt", ios::out);`
  - allow input / output operations
- Append:
  - `ofstream os("log.txt", ios::app);`
  - each insertion is done at the stream end
- Binary:
  - `ifstream is("pic.jpg", ios::binary);`
  - `ofstream os("pic.jpg", ios::binary);`
  - consider stream as binary (rather than text)

## 3.5 Control Statements and Loops

### 3.5.1 if .. then .. else / Switch

```

if (condition1) {
    //do this if condition1 is true
}
else if (condition2) {
    //otherwise do this if
    //condition2 is true
}
else {
    //otherwise do this
}

```

```

switch(m){
    case 0:
        //do this if 0
        break;
    case 1:
        //do this if 1 and got to the next
        //cause of missing break
    case 3:
        //do this if 3
        break;
    default:
        // if everything else fails do this
}

```

Für das m im switch können char, int, long und weitere Zahlenwerte verwendet werden.

### 3.5.2 Loops

```

for (int i = 0; i < 5; ++i) {
    cout << i << endl;
}

```

```

int j = 5;
while (j < 10) {
    cout << j << endl;
    ++j;
}

```

```

do {
    cout << j << endl;
    --j;
} while (j > 0);

```

## 3.6 Call by Value / Call by Reference



```
// Call by Value
int foo(int n) {
    n += 10;
    return (2 * n);
}
// Call by Reference
int bar(int& n) {
    n += 10;
    return (2 * n);
}
int main()
{
    int i = 10;
    cout << foo(i) << " " << i << endl; //Ausgabe 40 10
    cout << bar(i) << " " << i << endl; //Ausgabe 40 20
    return 0;
}
```

Bei Call by Value wird eine Deep Copy angelegt von dem Wert. Der Wert wird kopiert und mit dieser Kopie wird in der Methode gearbeitet.

Bei Call by Reference wird der Wert als Referenz übergeben. Dadurch wird der Wert außerhalb der Funktion verändert. Durch diesen Trick kann man sich z.B. mehrere Rückgabewerte erstellen.

**!!Achtung!!** Wenn eine Referenz als Argument erwartet wird, kann keine Zahl direkt übergeben werden. Lediglich Variablen, welche auf den Wert referenzieren, oder diesem direkt zugewiesen sind.

## 3.7 Pointer und Referenzen

### 3.7.1 Referenzen

Referenzen referenzieren einen Wert lediglich. Wenn man den Wert einer Referenz ändert, so ändert man den Wert der Variable, auf die die Referenz zeigt. Dies ist nützlich, da man Werte als Referenzen an eine Methode übergeben kann. Dadurch hat man die Möglichkeit, sich mehr als einen return Wert zu generieren. Da man die Variable außerhalb der Methode ändert und nicht nur die Unbekannte innerhalb der Methode.

```
// Initialisierung
int i = 2;
int& ri = i;
const int& ci = i;
double& blub = i; // => Fehler da die Typen identisch sein müssen.
int& rj; // Fehler, da Referenzen immer initialisiert sein müssen

// Wertzuweisung
i = 5; // i, ri, ci geben den Wert 5 aus
ri = 6; // i, ri, ci geben den Wert 6 aus
ci = 88 // Fehler, da ci als const definiert wurde und nicht geändert werden kann.
```

Mit `int& i` erstellt man eine Referenz. Mit `&i` bekommt man die Speicheradresse der Variable `i`.

### 3.7.2 Pointer

Pointer sind Variablen, die auf eine Speicheradresse verweisen. Sie verweisen nie direkt auf einen Wert.

```
int a, b; //two ints
```

### 3 C/C++

```
int* p; //a pointer to an int
p = &a; //nehme Speicheradresse von a und Speichere diese in p
*p = 10; //Dereferenziere p und veraendere den Wert.
p = &b;
*p = 20;
cout << a << endl; //10
cout << b << endl; //20
```

#### 3.7.3 Stolperfallen

```
int* p1, p2; // int pointer, int
int *p1, *p2; // int pointer, int pointer
//better
int* p1; //int pointer
int* p2; //int pointer
int* p = nullptr; // nullptr, sollte die Standard initialisierung sein.
//Damit der Pointer nicht mit irgendwelchen Werten initialisiert ist.
```

Pointer können hochgezählt werden wie normale integer. Denn der Wert der im Pointer steht ist lediglich eine Speicheradresse, also eine Zahl. Mit jedem p++ springt man um so viele Speicherstellen weiter, wie p groß ist.

```
int v = 5;
int* p = &v;
int** pp = &p;

cout << v; // 5
cout << p; // 0x40
cout << pp; //0x44

cout << &v; //0x40 (= p)
cout << &p; //0x44 (= pp)
cout << &pp; //0x48

cout << *p; //5
cout << *pp; //0x40 (= p)
cout << **pp; //5
```

Da Array Bezeichner gleichzusetzen sind mit einem Pointer auf den ersten Wert des Arrays, kann man Arrays genauso behandeln. Man kann Arrays in Funktionen als Pointer übergeben, (dabei sollte man die Größe des Arrays mit übergeben.) und diese hoch zählen um zum nächsten Element zu kommen oder auch Addition ausführen um eine Variable an einer bestimmten Stelle zu bekommen. Wenn man mit dem Wert in einer Funktion arbeiten will, muss man den Pointer dereferenzieren, ansonsten arbeitet man dort mit der Speicheradresse.

as	*	&
<b>type modifier</b>	<b>Pointer Declaration</b> int* ptr = nullptr;	<b>Reference Declaration</b> int& ref = a;
<b>unary operator</b>	<b>Dereference Operator</b> value = *ptr;	<b>Address Operator</b> ptr = &a;
<b>binary operator</b>	<b>Multiplication</b> prod = a * b;	<b>Bitwise Logical AND</b> c = (a & b);

Abbildung 3.3: Pointer/Referenzen Syntax

pointer to type T	pointed-to value modifiable	pointer itself modifiable
T*	yes	yes
const T*	no	yes
T* const	yes	no
const T* const	no	no

Abbildung 3.4: const Pointer

reference to type T	referenced value modifiable	reference itself modifiable
T&	yes	no
const T&	no	no

Abbildung 3.5: const Referenzen

## 3.8 Memory Management (Stack/Freestore)

### 3.8.1 Stack

- sequentieller Speicher von vordefinierter Größe
- zusammenhängender Speicher
- wird für lokale Variablen benutzt
- automatisches Speichermanagement
- last in first out
- für gewöhnlich lediglich ein paar MB groß

### 3 C/C++

- STack wurde für lokale Variablen und Parameter

Im Stack werden die einzelnen Variablen angelegt, sobald sie definiert werden. Wird eine Funktion aufgerufen, wird eine Return Adresse auf den Stack gelegt. Also wohin das Programm zurück springen muss, wenn die Funktion abgearbeitet ist. Darauf wird der Return Value im Speicher gelegt, sofern es einen gibt, dieser dient als Platzhalter für den Wert der Variable. Sobald die CPU in der nächsten Funktion ist wird ein Stack Frame angelegt, dieser ist eine Art Markierung auf dem Stack, bis wohin die nun kommenden Variablen gültig sind. Als erste Werte werden die Funktionsparameter auf dem Stack abgelegt. Wenn die Funktion beim return ankommt, wird dieser Value in den Placeholder (RV) geschrieben und der Stack wird bis einschließlich dem StackFrame frei gegeben, jedoch nicht gelöscht und die CPU springt zur Return Adresse.

**!!ACHTUNG!!** Wenn man eine Referenz zurück gibt, wird versucht diese aus dem Stack zu lesen, was nicht funktioniert, da diese Variable bereits frei gegeben wurde. Wenn man Glück hat, und keine Funktion zwischenzeitlich den Wert überschrieben hat, bekommt man den passenden Wert ansonsten einen vorher nicht definierten. Daher sollte man immer den Wert zurück geben, da der Stack von jeder Funktion überschrieben werden kann.

#### 3.8.2 FreeStore (Heap)

- großer Speicher Bereich (der größte Teil des RAM's)
- muss nicht zusammenhängend sein.
- wird benutzt um dynamisch Speicher zu allozieren
- manuelles Speichermanagement.
- Operator new alloziert ein neues Element im FreeStore
- Operator delete gibt den Speicher des Elements im FreeStore frei.
- dort angelegte Variablen / Objekte etc. leben so lange wie notwendig.

```
Type* p = new Type(args...); //create new variable of type "Type"
Type* a = new Type[s] //create new C-array of size s of type "Type"
delete p; //free variable p
delete[] a; //free C-array a (square bracket syntax)
```

#### Fallstricke und wie man diese umgeht

- nicht instantiierte Pointer können alles mögliche enthalten.
- der Programmierer muss sicherstellen, dass die Pointer instantiiert sind. Am besten mit nullptr .
- bei Übergabe von Pointer diese immer auf nullptr testen.
- wenn die Elemente im Pointer gelöscht werden, sollte der Pointer auf nullptr gesetzt werden.

### 3.9 Structs

- Typearg1,arg2,...,argN
- arguments in order of member declaration
- syntax also works for int, double, ...

- copies of plain struct objects are always deep copies
- use (const) references to avoid copies

```

struct Date { // description of data type Date
    int year;
    int month;
    int day;
};

void print(const Date& d) {
    cout << d.year << "-" << d.month << "-" << d.day << endl;
}

int main() {
    Date d; // create new object of type Date
    d.year = 2013; // set year to value 2013
    d.month = 5; //set month to value 5
    d.day = 15; //set day to value 15

    Date today {2014, 6, 25};

    auto yesterday = Date{2014, 6, 24};
}

```

Value Semantics:

deep copies (all data is copied)

Reference Semantics:

shallow copies (references are copied)

## 3.10 Complex Data Structures

Hier wird nur eine komplette Baum Implementation gezeigt. Da Listen einfach sind als Bäume, sollte diese Implementation reichen.

```

struct Node {
    int value;
    Node* left;
    Node* right;
};

Node* create_node(int value) {
    return new Node {value, nullptr, nullptr};
}

void insert(Node* current, Node* newNode) {
    if(newNode->value < current->value) {
        if(!current->left)
            current->left = newNode;
        else

```

```

        insert(current->left , newNode);
    }
    else {
        if(!current->right)
            current->right = newNode;
        else
            insert(current->right , newNode);
    }
    return;
}
void print_tree(Node* node)
{
    if(node->left)
        printNode(node->left );

    cout << node->value << endl;

    if(node->right)
        printNode(node->right );
}
void erase_tree(Node* node)
{
    if (node->left)
        erase_tree(node->left );
    if (node->right)
        erase_tree(node->right );
    delete node;
}

int main(){
    const int listLen=1;

    srand(time(0));

    Node* root = createNode(50);

    for(int i = 0; i < listLen; i++){
        int v = rand() %100; // random number from [0,99]
        insert(root ,createNode(v));
    }

    print_tree(root);

    erase_tree(root);
}

```

### 3.11 Standard Containers and Algorithms

Mit Dank an Leonie Pelle

### 3.11.1 Algorithmen

- Copy(firstIN, LastIN, firstOut)
- Find(first, last, wert)
  - First & last sind pointer
- Transform(first, last, out, f)
- Sort(first, last)

### 3.11.2 Includes

- #include<iostream> für cin und cout
- #include <fstream> instream & outstream
- #include <string> string verarbeitung
- #include<algorithm> algorithmen

### 3.11.3 Sequentielle Container

- Vector <T>
  - Variable Liste
  - Vector<int> v ...,..
  - V[0]
  - V.front() first element
  - V.back() last element
  - V.push\_back(...) einfügen am ende
  - V.size() größe
  - V.rezise(anzahl, wert) vergrößern auf anzahl und füllen mit wert
  - V.clear() leeren
  - V.insert(begin(v), wert); wert kann auch eine Menge sein
  - V.pop\_back() löschen letztes element und speicher freigebe
  - V.erase(pointer) löschen eines elements an der stelle
  - V.erase(pointer, pointer2) löschen von pointer zu pointer2
  - V.assign(von, bis) einfügen von bis
- Array<T,size>
  - Feste liste
  - Wie vector
- List<T>
  - Doppelt verkettete liste
  - V.Splice(begin(v), k, begin(k), end(k)) einfügen von k (auch list) an der stele begin(v) von begin k bis end k (AUFPASSEN: die elemente warden aus k gelöscht)

### 3.11.4 Assoziative Container

- Set<key>
  - Key:
    - \* Eindeutig
    - \* Sortierbar
  - S.insert(...) sortiertes einfügen
  - S.find(...) finden (falls nicht gefunden Rückgabe von end s)
- Map<key, value>
  - Wie set
  - AUFPASSEN Aufruf von m[42] (42 ist ein nicht existenter key) wird angelegt mit einem leerem wert
- Unordered\_set<key>
  - Hashset
  - Key muss nicht sortierbar sein
  - Wie set
- Unordered\_map<key, value>
  - Wie unordered\_set<key>



# Abbildungsverzeichnis

1.1	Sharing + Outermost . . . . .	10
2.1	Prädikat Unterteilung . . . . .	11
3.1	compiling . . . . .	19
3.2	typen . . . . .	22
3.3	Pointer/Referenzen Syntax . . . . .	29
3.4	const Pointer . . . . .	29
3.5	const Referenzen . . . . .	29