Fachseminar Programmiersprachen WS07/08 (Entwurf)

11. Januar 2008

Inhaltsverzeichnis

1 Einführung **2**

2 OCaml
(Steffen Brümmer) **3**
2.1 Was ist OCaml? . . . 3
2.2 Von Caml zu OCaml . . . 3
2.3 Compiler und Tools . . . 4
2.4 Grundlagen der Sprache . . . 6
2.5 Imperative Konzepte . . . 8
2.6 Listen . . . 9
2.7 Module . . . 10
2.8 Objektorientierung . . . 11
2.9 Fazit, Verwendung von OCaml in der Praxis . . . 15

3 F#
(Martin Lohrmann) **17**
3.1 Einleitung . . . 17
3.2 Ziele und Einsatzgebiete . . . 17
3.3 Werkrzeuge zur Entwicklung . . . 18
3.3.1 .NET Framework . . . 18
3.3.2 der Compiler . . . 18
3.3.3 die Entwicklungsumgebung . . . 18
3.3.4 F# Interactive . . . 18
3.3.5 Nutzung unter Linux . . . 18
3.4 Ein erstes Beispiel . . . 19
3.5 Sprachfeatures . . . 20
3.5.1 Variablen und Funktionen . . . 20
3.5.2 Tuple . . . 21
3.5.3 Records . . . 21
3.5.4 Listen und Arrays . . . 22
3.5.5 Referenz . . . 22
3.5.6 if-Anweisung . . . 22
3.5.7 Schleifen . . . 23
3.5.8 Object Types . . . 23

4 Scala (Fabian Keller) **26**
4.1 Einleitung . . . 26
4.1.1 Was ist Scala . . . 26
4.1.2 Geschichte . . . 26
4.1.3 Ziele bei der Entwicklung . . . 27
4.2 Verfügbare Compiler und Interpreter . . . 27
4.2.1 Der Interpreter . . . 27
4.2.2 Der Compiler . . . 28
4.2.3 Scala als Shellscript . . . 28
4.3 Ausführungsmodell . . . 29
4.4 Allgemeine Sprachkonstrukte . . . 29
4.4.1 Variablendeklaration . . . 30
4.4.2 Bedingungen . . . 30
4.4.3 Schleifen . . . 30
4.4.4 Arrays . . . 31
4.4.5 Methoden . . . 31
4.4.6 Funktionen . . . 31
4.4.7 Infixnotation . . . 32
4.5 Typensystem . . . 32
4.5.1 Typen . . . 32
4.5.2 Superklassen . . . 33
4.6 Objektorientiertes Paradigma . . . 33
4.6.1 Klassen, Objekte und Vererbung . . . 33
4.6.2 Mixin Klassen . . . 34
4.6.3 Traits . . . 34
4.6.4 Generische Klassen . . . 35
4.6.5 Singleton-Objekte . . . 35
4.7 Funktionales Paradigma . . . 35
4.7.1 Anonyme Funktionen . . . 35
4.7.2 Currying . . . 36
4.7.3 Higher Order Functions . . . 36
4.7.4 Pattern Matching . . . 37
4.8 Webservices Paradigma . . . 37
4.8.1 XML Processing . . . 37
4.9 Beispiele . . . 39
4.9.1 Interaktion mit Java . . . 39
4.9.2 Setterfunktion . . . 40
4.10 Zusammenfassung . . . 40
4.11 Anhang . . . 41
4.11.1 Installation (Quickstart) . . . 41

5 Python (Robert Rauschenberg) **42**
5.1 Einleitung . . . 42
5.2 Geschichte . . . 42
5.2.1 Woher kommt Python ? . . . 42
5.2.2 Python der Name . . . 42
5.2.3 Weiteres . . . 43
5.3 Python wofuer steht es? . . . 43

5.3.1 Typisierung und Plattformunabhängigkeit 43
5.3.2 Plattformunabhängig 43
5.4 Beispiele zur Programmierung 43
5.4.1 Hello World in Python 44
5.4.2 Wertzuweisungen 44
5.4.3 Arbeiten mit Listen 46
5.4.4 GUI eine schnelle Oberfläche 46
5.5 Weitere Informationen 46

6 Haskell
(Andreas Textor) 47
6.1 Einleitung 47
6.2 Haskellcompiler und -Interpreter 48
6.3 Die Grundeigenschaften am Beispiel 48
6.3.1 Eine „funktionale“ Sprache 49
6.3.2 Verschiedene Schreibweisen für Funktionen 50
6.3.3 Typklassen 51
6.4 Datentypen und ihre Manipulation 52
6.4.1 Typsystem und Grundlegende Datentypen 52
6.4.2 Listen 53
6.4.3 Tupel 56
6.4.4 Selbstdefinierte Datentypen 56
6.5 Funktionale Programmierung 59
6.5.1 Verzögerte Auswertung 59
6.5.2 Lambda-Kalkül 60
6.5.3 Funktionen höherer Ordnung 61
6.5.4 Funktionskomposition 63
6.6 Monaden 63
6.6.1 Das Problem mit Seiteneffekten 63
6.6.2 Ein- und Ausgabe 64
6.6.3 Kombination von Monaden 65
6.7 Entwicklung von Haskell und sonstige Features 65
6.7.1 Fazit 66

7 Ada
(Alexey Korobov) 68
7.1 Einleitung 68
7.2 Exkurs in die Geschichte 68
7.3 Tools & Co. 69
7.3.1 Reference Manual 69
7.3.2 Compiler & Editoren 69
7.4 Ausführungsmodell 70
7.5 Besondere Sprachmerkmale 70
7.5.1 Strenge statische Typisierung auf angenehme Art 71
7.5.2 Skalare Typen 71
7.5.3 Syntaktische Highlights 71
7.5.4 Überladung 71
7.5.5 Zeiger & Co. 71
7.5.6 OOP à la Ada 71
7.5.7 Generische Programmierung 72

7.5.8 Nebenläufigkeit 72
7.5.9 Anschluß an andere Sprachen 72
7.5.10 Pragmas 72
7.5.11 Compiler und -Laufzeittests 72
7.5.12 Beispielprogramm 72
7.6 Anwendungsgebiete 72
7.7 Fazit 73
7.8 Quellenangaben 73

8 D
(Alexander Stolz) 74
8.1 Einleitung 74
8.2 Geschichte 75
8.3 Compiler 75
8.3.1 DMD 75
8.3.2 GDC 76
8.3.3 Compilerbenutzung 76
8.4 Sprachfeatures 77
8.4.1 Primitivie Datentypen 77
8.4.2 Objektorientierung 78
8.4.3 Arrays 80
8.4.4 Funktionen/Methoden 83
8.4.5 Templates 88
8.4.6 Produktivität 91
8.4.7 Zuverlässigkeit 93
8.5 Fazit 95

9 objective C
(Marco Rancinger) 96
9.1 Einleitung 96
9.1.1 Wer sollte das hier lesen 96
9.1.2 Warum Objective C? 96
9.2 Geschichte der Sprache 98
9.2.1 Warum so viele C's? 99
9.2.2 Nach der Entwicklung 99
9.3 Die Entwicklungsumgebung 99
9.3.1 Tools unter Mac OS X 100
9.3.2 Tools unter Windows 100
9.4 DasCocoaFramework 102
9.4.1 Mac OS X 102
9.5 Sprachliche Elemente von Objective C 103
9.5.1 Alte Freunde 103
9.5.2 Klassen 106
9.6 Besonderheiten der Sprache 112
9.6.1 Speicherverwaltung 112
9.6.2 Typlose Objekte 114
9.6.3 Vergleich mit anderen Sprachen 116
9.7 Beispielapplikation 116
9.7.1 Unser "kleines"Beispiel 117

10 Erlang
(Arend Kühle) 125
10.1 Wie Erlang entstand 125
10.2 Die Entwicklungsumgebung 127
10.3 Der Code 128
10.3.1 Variablen und Konstanten 128
10.3.2 Datentypen 128
10.3.3 Funktionen 129
10.3.4 Hello, World! 131
10.4 Ausführungsmodell 131
10.4.1 Funktional 131
10.4.2 Nebenläufigkeit 131
10.4.3 Prozesse 132
10.4.4 Beispiel für Prozesskummunikation 132
10.4.5 Errorhandling 133
10.5 Erlang in der Welt 134
10.5.1 Yaws - Yet another webserver 134
10.5.2 Der Erfolg 134
10.5.3 Fazit 134

11 Prolog
(Lars Thielmann) 135
11.1 Einleitung 135
11.2 Geschichte 135
11.3 Prinzip von Prolog 136
11.3.1 Aufbau der Sprache 136
11.3.2 Vom Programm zur Berechnung 137
11.4 Charakterisierende Eigenschaften 138
11.4.1 Der Cut-Operator '!' 138
11.5 Typische Probleme und Beispielapplikationen 139
11.5.1 Quicksort 139
11.5.2 Einsteins Rätsel 139
11.6 Editoren und Interpreter 141
11.7 Zusammenfassung 142

12 XQuery
(Fabian Meyer) 143
12.1 Einleitung 143
12.2 XML 144
12.2.1 Einführung 144
12.2.2 Aufbau 144
12.2.3 Beispiel 144
12.3 XPath 145
12.3.1 Einführung 145
12.3.2 Aufbau 145
12.3.3 Beispiele 147
12.4 XQuery 147
12.4.1 Einführung 147
12.4.2 Aufbau 147
12.4.3 Variablen 147

12.4.4 Abfragen 148
12.4.5 Funktionen 149
12.4.6 Intallation 150
12.5 Beispiele 152
12.6 Fazit 155

Zusammenfassung

Neben den häufig gebräuchlichen Programmiersprachen Java und C(++), gibt es eine schier unüberblickbare Menge unterschiedlichster Programmiersprachen. Exemplarisch werden ein paar unterschiedliche Programmiersprachen in ihren Grundcharakteristiken vorgestellt.

Kapitel 1

Einführung

Eine der Sprachen ist zum Beispiel Scala[Oa06a].

Kapitel 2

OCaml (Steffen Brümmer)

2.1 Was ist OCaml?

Der Begriff OCaml bezeichnet eine Programmiersprache, die am *Institut national de recherche en informatique et en automatique* (INRIA) in Frankreich entwickelt wird, einem staatlichen Forschungsinstitut für Wissenschaft und Technologie in Bereichen wie Netzwerke, Systeme und Softwareengineering. OCaml erweitert die vorher entwickelte Sprache Caml um objektorientierte Konzepte.

2.2 Von Caml zu OCaml

Caml ist die Abkürzung für Categorical Abstract Machine Language. Der Begriff ist ein Wortspiel auf CAM, was für Categorical Abstract Machine steht, und ML, die Abkürzung für Meta Language. Meta Language wurde 1973 von Robin Milner an der Universität Edinburgh entworfen. Es handelt sich dabei um eine funktionale Programmiersprache. Merkmale dieser Sprache sind beispielsweise automatische Speicherbereinigung, Polymorphie oder statische Typisierung. Es gibt mehrere Vertreter von ML, wobei Standard ML, Lazy ML und Caml die bekanntesten sind. Caml selbst wurde Anfang der 80er Jahre am INRIA von Gérard Hue auf der Basis von ML entwickelt. Die erste Implementation erschien 1987. Drei Jahre später wurde Caml dann von Xavier Leroy komplett neu designt. Unter anderem wurde ein neues Speicherbereinigungssystem hinzugefügt. Diese neue Version, bekannt als CamlLight, war höchst portabel und lief auf kleinen Desktops wie dem Mac oder dem PC. Diese Implementation basierte auf einem Bytecode-Interpreter, der in C geschrieben war. Im Jahr 1995 veröffentlichte man die nächste Version mit dem Namen Caml Special Light. Als erstes wurde ein sogenannter Native-Code Compiler hinzugefügt, welcher Bytecode in Maschinencode übersetzt. Der Compiler hatte eine ähnliche Performanz wie die besten bereits existierenden Compiler für funktionale Sprachen, konnte diese sogar noch übertreffen. Somit ermöglichte es Caml in diesem Aspekt wettbewerbsfähiger zu geläufigeren Programmiersprachen, wie etwa C++, zu werden. Des weiteren bot das neue Caml ein hervorragendes Modulsystem an, das

von Xavier Leroy designt wurde und auf dem Modulsystem von Standard ML basiert. Dieses System stellt mächtige Abstraktions- und Parameterisierungseigenschaften zur Verfügung, insbesondere für große, umfassende Anwendungen. Typsysteme und Typinferenz für objektorientiertes Programmieren waren ein wichtiges Thema bei Forschungsarbeiten seit den frühen Neunzigern. Didier Rémy, später auch Jérome Vouillon vom INRIA designten eine elegantes und höchst ausdrucksvolles Typsystem für Objekte und Klassen. Dieses Design wurde durch Caml Special Light integriert und implementiert, was letztendlich zu OCaml (Objective Caml) führte und 1996 veröffentlicht wurde. OCaml war die erste Sprache (und ist dies auch noch), welche die volle Mächtigkeit der Objektorientierung mit ML-typischer statischer Typisierung und Typinferenz kombiniert. Es unterstützt viele fortgeschrittene Askpekte der objektorientierten Programmierung. Seit seinem Erscheinen gewinnt OCaml stetig an Popularität und spricht eine bedeutende Basis an Usern an. Zusätzlich zu beeindruckenden Programmen, die in OCaml geschrieben wurden, steuerte die Community viele erstklassige Bibliotheken bei, beispielsweise Frameworks und Tools in Bereichen wie GUI's, dem webbasierten Einbinden von Datenbanken oder der Netzwerkprogrammierung. Währenddessen hält das OCaml Entwicklungs-Team das Basissytem aufrecht, verbessert dabei die Qualität der Implementation und bezieht die neuesten Systeme und Architekturen mit ein.

2.3 Compiler und Tools

OCaml verfügt über einen interaktiven Interpreter, einen Bytecode-Compiler sowie den oben genannten Native-Code Compiler. Der Interpreter wird mit dem Befehl "caml" aufgerufen. Hier, im sogenannten "Top-Level", kann der Nutzer Ausdrücke eingeben, die sofort ausgewertet werden. Der OCaml-Interpreter wartet nun auf unsere Eingabe:

```
Objective Caml version 3.10.0
#
```

Geben wir nun beispielsweise "1;;" ein, führt dies zu folgender Ausgabe:

```
Objective Caml version 3.10.0
# 1;;
- : int = 1
#
```

Erläuterung: Der eingegebene Ausdruck "1" muss mit ";;" beendet werden. Die Ausgabe "- : int = 1" zeigt, das automatisch ein Integer-Wert erkannt und somit der Ausdruck sofort ausgewertet wurde. Mehr dazu in den nächsten Kapiteln.
Der Compiler kann mit "ocamlc" aufgerufen werden. Dieser compiliert Caml Quelldateien in Bytecode Objekt-Dateien, die zu Objektdateien gelinkt werden, wodurch wiederrum ausführbare Dateien erzeugt werden. Die ausführbaren Dateien können dann vom Bytecode-Interpreter "ocamlrun" gestartet werden. ocamlc hat ein Kommandozeileninterface ähnlich wie die meisten C-Compiler. Er akzeptiert mehrere Argumente, die dann sequentiell ausgeführt werden. Wesentlich schneller als camlc ist der Native-Code Compiler, der Bytecode in Maschinencode übersetzt und eigenständige ausführbare Dateien erzeugt. In OCaml

ist jede Datei ein Modul. Im folgenden Beispiel wird gezeigt, wie zum Beispiel eine ausführbare Datei aus zwei Caml-Modulen erzeugt wird. Nehmen wir an, die Datei “amodul.ml” enthält folgenden Quellcode:

```
let hello () = print_endline "Hello"
```

Und die Datei “bmodul.ml” diesen Quellcode:

```
Amodul.hello ()
```

Hier fällt sicher der Funtionsaufruf “Amodule.hello()” mit einem Großbuchstaben auf, wo hingegen die eigentliche Datei “amodul.ml” heißt. Wenn man ein Programm schreibt, definiert jede Datei automatisch ein Modul, wie in diesem Fall das Modul “Amodul”. Gewöhnlich werden die Dateien dann nacheinander compiliert:

```
ocamlopt -c amodul.lm
ocamlopt -c bmodul.lm
```

Die Option “-c” sorgt dafür, daß nur compiliert wird und das Linken unterdrückt wird. Quelldateien werden in compilierte Dateien umgewandelt, allerdings wird keine ausführbare Datei erzeugt. Somit können Module einzeln compiliert werden. Als Ergebnis erhält man nun eine .o-Datei, die den Objektcode enthält, sowie eine .cmx-Datei, die Informationen zum Linken enthält.

```
ocamlopt -o hello amodule.cmx bmodule.cmx
```

Der obige Aufruf sorgt schließlich dafür, daß die ausführbare “hello.exe” erzeugt wird. Wird kein Dateiname angegeben, würde wie bei C-Compilern eine Datei “a.out” (Linux) oder “camlprog.exe” (Windows) erzeugt werden.
Zusätzlich dazu verfügt OCaml noch über folgende Tools:

ocamllex
ein Programm, mit dem Quellcode für andere Programme (Scanner) erzeugt werden kann. Dient wie das weit verbreitete Werkzeug “Lex” dazu, Eingabedaten nach bestimmten Mustern (einer Grammatik folgend) zu durchsuchen und bestimmte Aktionen auszuführen.

ocamlyacc
ein Tool zur Generierung von Parsern, d.h. zur Erkennung von grammatikalischen Strukturen in Programmen.

ocamldep
dieses Tool scannt mehrere OCaml Quelldateien und sucht jeweils nach Referenzen auf externe Dateien, um sicher zu gehen, daß die Quelldateien in der richtigen Reihenfolge compiliert werden bzw. richtig recompiliert werden, falls Dateien geändert werden.

ocamlbrowser
stellt einen Editor bereit und bietet die Möglichkeit, innerhalb von Modulen zu navigieren.

ocamldoc
generiert eine Dokumentation auf Grund der Kommentare im Quelltext. Dieses Tool kann Dokumentationen in folgenden Formaten erzeugen: HTML, LaTex, TEXinfo, Unix man pages

Darüber hinaus verfügt OCaml über das Tool **ocamlprof**, mit dem etwa die Anzahl von Funktionsaufrufen oder von besuchten Verzweigungen aufgezeichnet werden können, sowie den Debugger **ocamldebug**.

2.4 Grundlagen der Sprache

Kommentare

Kommentare in OCaml werden von (* und *) umschlossen. Dabei sind mehrzeilige Kommentare

```
(* Das ist ein
 * mehrzeiliger
 * Kommentar
 *)
```

und auch verschachtelte Kommentare erlaubt.

```
(* Hier fehlt noch etwas
  (* Prüfe, ob gerade *)
  let is_odd i =
*)
```

Funktionsaufrufe

Funktionen in OCaml werden folgendermaßen aufgerufen:

```
repeat "Das ist ein Test" 3
```

Hier wird die Funktion repeat mit zwei Argumenten, nämlich einem string und einem int aufgerufen. Die Funktion repeat könnte beispielsweise den übergebenen Text drei mal ausgeben. Im Gegensatz zu geläufigeren Sprachen wie C oder Java werden hinter dem Funktionsnamen keine Klammern benötigt. Jedoch wäre ein Funktionsaufruf mit Klammern, also

```
repeat ("Das ist ein Test", 3)
```

eigentlich richtig, da in diesem Fall die Funktion mit einem Wertepaar, also mit einem Argument aufgerufen würde. Allerdings erwartet die Funktion repeat

zwei Argumente. Zum Schluss noch ein weiteres Beispiel mit Funktionen höherer Ordnung:

```
f 5 (g "Hallo") 3
```

f hat drei Argumente, nämlich 5, (g "Hallo") und 3, g hat ein Argument, nämlich "Hallo".

Definition von Funktionen

Die folgende C-Funktion

```
double average(double a, double b){
    return a + b / 2;
}
```

wird in OCaml folgendermaßen geschrieben:

```
let average a b = (a +.b) /. 2.0
```

Die OCaml-Version soll nun kurz erläutert werden.
Mit dem Schlüsselwort "let" können Variablen oder Funktionen deklariert werden. Danach folgt der Name der Funktion sowie die Parameter. Der Wert des letzten Ausdrucks wird von OCaml zurückgeliefert, daher wird ein return nicht benötigt. Genauerer Erklärung bedarf es allerdings den Operatoren "+." und "/.", sowie der Tatsache, daß nirgendwo die Typen der Parameter definiert wurden. Hier kommt ein Konzept der Ursprungssprache MetaLanguage zum Tragen, nämlich der statischen Typisierung. Die Punkte hinter den Operatoren bedeuten, daß es sich um Rechenoperationen für Fließkommazahlen handelt. Deshalb erwartet die Funktion auch zwei float-Werte, die auch in der Form 1.0 übergeben werden müssen. Übergibt man Ganzzahl-Werte, zeigt der Compiler einen Fehler an. Daher müssen die Typen der Parameter nicht explizit angegeben werden, sondern werden per Typableitung ermittelt. OCaml führt also niemals implizite Typumwandlungen durch, wie z.B. C oder mit C verwandte Sprachen. Denn hier könnte man den Ausdruck "4 + 2.5" schreiben, wobei das Ergebnis eine Fließkommazahl wäre. Die 4 wird bei einer solchen Operation implizit in einen float-Wert umgewandelt.

Primitive Typen

Es gibt die folgenden primitiven Typen in OCaml:

int (0, 5, 42, -17, 0x00FF)
float (0.0. -5.3, 1.7e14)
bool (true, false)
string (, test")
char (a, \n)
unit (ähnlich wie void in C)

2.5 Imperative Konzepte

Im Folgenden werden einige Kontrollstrukturen erklärt.

if-Anweisung

OCaml kennt zwei if-Anweisungen:

```
if Bedingung then Anweisung
```

sowie

```
if Bedingung then Anweisung1 else Anweisung2
```

Diese Syntax ähnelt sehr der aus konventionellen Sprachen bekannten Syntax

```
Bedingung ? Anweisung1 : Anweisung2
```

Anhand des folgenden Beispiels kann man ein weiteres Merkmal von OCaml erkennen, nämlich das Prinzip der Polymorphie.

```
let max a b = if a > b then a else b;;
```

Die Funktion ist einfach, sie gibt lediglich den größeren der beiden übergebenen Werte zurück. Man kann aber verschiedene Typen an die Funktion übergeben, zum Beispiel int, float oder string. Der Operator "¿" ist nämlich polymorphisch und kann auf verschiedenen Typen angewendet werden. Zu beachten ist allerdings, daß, wie bereits oben eklärt, OCaml keine implizite Typumwandlung durchführt. Daher würde die Eingabe von zwei Parametern verschiedenen Typs zu einem Fehler führen.

Schleifen

OCaml verfügt nur über eine begrenzte Anzahl von Schleifen. Zunächst die allgemeine Syntax für eine for-Schleife:

```
for variable = anfangswert to endwert do
   Anweisung
done
```

Die Schleifen in OCaml können nicht wie in anderen Sprachen durch "break" oder "continue" ausgesetzt werden. Man könnte stattdessen eine Exception in der Schleife werfen, die dann außerhalb aufgefangen wird. Auch die while-Schleife verfügt über keine Möglichkeit zum Aussetzen. Auch hier nur kurz die allgemeine Syntax:

```
while boolesche Bedingung do
    Anweisung
done
```

Aufgrund der Tatsache, daß die Schleifen nicht ausgesetzt werden können, außer durch Werfen von Exceptions, gelten diese Sprachkonstrukte in OCaml als "Bürger zweiter Klasse". Man verwendet daher eher das Prinzip der Rekursion.

Rekursion

Rekursive Funktionen werden mit dem Schlüsselwort "rec" eingeleitet:

```
let rec fac n = if n = 0 then 1 else n * fac(n - 1);;
```

Die obige Funktion berechnet die Fakultät von n rekursiv. Eleganter kann man die obige Funktion mit einem Konzept der MetaLanguage-Sprachfamilie schreiben, dem so genannten Pattern-Matching. Ein Pattern (Schablone) kann dabei aus Konstanten, Variablen oder Wildcards bestehen. Das Pattern-Matching funktioniert folgendermaßen:
Ein Pattern und ein Wert werden gegenübergestellt und dann verglichen. Die Match-Ausdrücke müssen dabei vollständig sein, d.h. es müssen alle Möglichkeiten abgedeckt sein. Die Syntax für eine Match:

```
match Ausdruck with
    Pattern1 -> Ausdruck1
  | Pattern1 -> Ausdruck2
    ...
```

Ist kein passendes Pattern zu einem Match, wird eine Exception geworfen. In unserem Rekursionsbeispiel würde man das Pattern-Matching also wie folgt umsetzen:

```
let rec fac = function
     0 -> 1
   | n -> n * fac (n - 1);;
```

2.6 Listen

OCaml stellt bereits einen Datentyp Liste zur Verfügung. Die leere Liste wird dargestellt durch "[]" . Die Elemente in einer Liste müssen alle vom selben Typ sein. Es kann Listen von primitiven Typen wie int, float und auch bool, sowie Listen von Funktionen geben. Konstruieren wir als Beispiel einige Listen. Zunächst eine int-Liste.

```
# [1;2;3];;
- : int list = [1; 2; 3]
```

Wendet man den Operator "::" auf einen Wert und eine Liste an, ergibt das eine neue Liste:

```
# 1::2::3::[4;5;6];;

- : int list = [1; 2; 3; 4; 5; 6]
```

Listen verfügen über ein Head und ein Tail, also das erste Element sowie die angehängten Elemente. OCaml bringt bereits eine Reihe von Funktionen mit, die auf Listen angewendet werden können. Zum Beispiel length, die auch polymorph ist. Bestimmen wir nun die Länge einer Liste rekursiv:

```
# let rec len l =
      if l = []
      then 0
      else l + len (List.tl l);;
```

Mapping auf Listen

Die Mächtigkeit der Ausdruckskraft funktionaler Sprachen zeigt sich durch die Einfachheit bei Manipulationen von Werten. Das soll am Beispiel des Mappings von polymorphen Funktionen auf polymorphe Listen gezeigt werden. Mapping heißt in diesem Fall das Anwenden von Funktionen auf alle Elemente einer List. Wir definieren uns eine rekursive Funktion map:

```
let rec map f  l = match l with
     [] -> []
   | h::t -> f h :: map f t
```

Die rekursive Funktion map hat zwei Parameter, nämlich einmal eine Funktion "f", sowie eine Liste "l". Durch das Matching wird dann überprüft, ob die Liste leer ist, in diesem Fall wird auch eine leere Liste zurückgegeben. Verfügt die Liste über ein Head und ein Tail, wird die Anweisung "f h :: map f t" ausgeführt. Das "f h" vor den beiden Doppelpunkten übergibt den Head der Liste "h" an die Funktion "f", welche dann den Wert verarbeitet. Dieser veränderte Wert wird dann zurückgegeben und mittels des "::"-Operators vorne an die Liste gehängt. Genauer gesagt wird der veränderte Head vorne an das Tail angefügt, welches dann wieder mit der Funktion f an map übergeben wird. So werden rekursiv die Werte der Liste verändert. Rufen wir nun die Funktion map mit einer Funktion und einer Liste als Parameter auf:

```
# map (fun x -> x + x) [1;2;3;4];;
```

So erhalten wir das Ergebnis

```
- : int list = [2; 4; 6; 8]
```

2.7 Module

Die Motivation, in OCaml Module zu verwenden, ist verwandte Definitionen zu kapseln. Zum Beispiel definiert man einen Datentyp mit allen diesen Typ betreffenden Operationen. Module sind ein grundlegendes Strukturierungsmittel für große Programme. Die erste Sprache, für die ein Modulsystem entworfen wurde, war übrigens StandardML. Das Modulsystem besteht aus drei Komponenten: Zum einen der Signatur, die aussagekräftige Schnittstellen nach außen bereit

stellen soll. Zum anderen den Strukturen, welche die einzelnen Programmteile kapseln sowie den sogenannten Funktoren, die Module auf andere Module abbilden. Wird nun eine Signatur einer Struktur zugeordnet, kann der Zugriff auf die Typen und Methoden beschränkt werden, die Signatur fungiert also wie bereits erwähnt als Interface. Signaturen sind vergleichbar mit header-Dateien in C, in denen die Typen und Funktionen deklariert werden, in den Strukturen werden diese hingegen implementiert (.c Dateien). Module können auch in andere Module eingebunden werden. So kann ein Modul durch das Schlüsselwort "open" für andere Module geöffnet werden, oder mit "include" andere Module einbinden. Mit Hilfe von Funktoren kann man Strukturen quasi als Funktionen auf Strukturen anwenden. Zum Beispiel: Wir haben eine Struktur, die Listen implementiert. Diese Struktur können wir nun mit einer Struktur, die den Typ einer Liste angibt oder einer Struktur, die eine Sortierfunktion auf die Listenstruktur anwendet, parametrisieren.

Signatur

```
# module type SignaturName =
     sig
         signatur
     end;;
```

Struktur

```
# module ModulName =
     struct
          implemantation
     end;;
```

Funktor

```
# module FunktorName (ArgumentName : ArgumentSignatur)
     struct
           implementation
     end;;
```

2.8 Objektorientierung

OCaml ist die erste auf MetaLanguage basierende Sprache, die objektorientiertes Programmieren erlaubt. Im Kapitel Module haben wir das Konzept der Erweiterbarkeit kennen gelernt. Dem steht nun das objektorientierte Konzept der Vererbung gegenüber. In OCaml können Klassen von mehreren Oberklassen erben, genau wie in C++. Im Gegensatz zu andern OO-Sprachen wie C++ oder Eiffel unterscheidet OCaml zwischen Subtyphierarchie und Vererbungshierarchie. Das bedeutet, daß Unterklassen von einem anderen Typ sind als ihrer jeweiligen Oberklasse. Auch das Prinzip des "Late-Binding" wird in OCaml verwendet. Objekte werden also erst zur Laufzeit typisiert. Von außen kann grundsätzlich auf die Methoden einer Klasse zugegriffen werden, jedoch nicht

auf die Attribute (es sind also getter- und setter-Methoden notwendig). Des weiteren können Interfaces benutzt werden, die die Sichtbarkeit von Methoden und Typen beschränkt. Wie in Java kann mit dem Schlüsselwort "super" auf die Methoden einer Oberklasse zugegriffen werden. Ein Unterschied zu Java liegt allerdings darin, daß beim Instanziieren eines Objektes mit dem Schlüsselwort new der Klassenrumpf vollständig ausgewertet wird. Im Folgenden möchte ich durch einige konkrete Beispiele die Schreibweise und den Aufbau von Klassen in OCaml verdeutlichen.

Klassen und Objekte

Zunächst soll der Aufbau einer Klasse gezeigt werden:

```
# class punkt =
   object
     val mutable x = 0
     method get_x = x
     method bewege d = x <- x + d
   end;;
```

Die Klasse punkt definiert eine Instanzvariable x, die mit 0 initialisiert wird und veränderbar ist ("mutable"). Diese Veränderung kann die Methode bewege durchführen, die dem x-Wert ein delta d hinzuaddiert. Aus Überschichtlichkeit wurde hier auf die zweite y-Komponente des Punktes verzichtet. Als nächstes bilden wir eine Instanz von punkt:

```
# let p = new punkt;;
```

Methodenaufrufe werden mit dem #-Operator durchgeführt,
also objekt#methode:

```
# p#bewege 3;;
- unit = ()
# p#get_x;;
- : int = 3
```

Wie erwartet wurde der Punkt p verschoben. Der Variable x kann auch anders initialisiert werden, in dem wir beim Instanziieren des Objektes Parameter übergeben:

```
# class punkt x_init =
   object
     val mutable x = x_init
     method get_x = x
     method bewege d = x <- x + d
   end;;
```

Eine Instanz von Punkt erwartet jetzt also einen Parameter:

```
# let p = new punkt 5;;
```

Der Parameter x_init ist im gesamten Klassenrumpf sichtbar. So könnten wir beispielsweise eine Methode getVerschiebung hinzufügen, die den Wert von x einbezieht:

```
class punkt x_init =
  ...
  val mutable x = x_init
  method getVerschiebung = x - x_init
  ...
```

Es können auch sogenannte "immediate Objects" in OCaml erzeugt werden, das heißt es gibt einzelne Objekte ohne Klassendefinitionen. Diese Objekte werden wie Variablen mit dem Schlüsselwort "let" deklariert:

```
# let p = object
            val x = 0
            method get_x = x
          end;;
```

Aus C++ oder Java ist das Schlüsselwort this"bekannt. OCaml kennt eine ähnliche Notation. Wenn wir eine Klasse deklarieren, kann eine Variable deklariert werden, die an das Objekt der Klasse gebunden ist. In OCaml wird das ßelf"genannt. Schreiben wir dazu eine Klasse printable_punkt, die die Methode print zur Ausgabe hat.

```
class printable_punkt x_init =
   object (self)
   (* self wird deklariert, damit ist
      das momentane Objekt gemeint *)
      val mutable x = x_init
      ...
      (* hier wird die get_x
         Methode des momentanen
         Objektes aufgerufen *)
      method print = print_int self#get_x

   end
```

Auch in OCaml ist es möglich, virtuelle Methoden zu deklarieren. Dazu muss die Klasse auch als virtuell gekennzeichnet werden:

```
class virtual punkt =
   object (self)
      (* virtuelle Methode
         mit einem int-Rückgabewert*)
      method virtual get_x : int
      ...
```

Virtuelle Methoden werden wie in C++ den Unterklassen zur Verfügung gestellt. Ebenso können keine Objekte einer virtuellen Klasse erzeugt werden.

Vererbung

Um das Prinzip der Vererbung in OCaml zu erklären, schreiben wir eine Klasse farb_punkt, die von der Klasse punkt erben soll:

```
class farb_punkt x (c: string) =
   object
     inherit punkt x
     val c = c
     method farbe = c
   end
```

Beim Erzeugen eines Objekts vom Typ farb_punkt müssen wir zwei Parameter mit übergeben: zum einen einen Startwert für x, zum anderen einen String, der hier vereinfacht die Farbe des Punktes darstellen soll. "inherit punkt x" besagt, daß die Klasse von Punkt erbt. Diese Klasse erweitert punkt noch um die Eigenschaft farbe. Erzeugen wir nun der Vollständigkeit halber ein Objekt:

```
# let p1 = new farb_punkt 5 "blau";;
val p1 : farb_punkt = <obj>
```

Auf die Konsole wird nochmal der Name, die Klasse und der Typ des Objektes geschrieben.

Mehrfachvererbung

Wie bei der Vererbung üblich, überschreibt die Redefinition einer Methode in einer Unterklasse die Methode der Oberklasse. Vorige Definitionen einer Methode können benutzt werden, in dem sie an die betreffende Elternklasse gebunden werden. Im unteren Beispiel ist das Schlüsselwort "super" an die Elternklasse printable_punkt gebunden. "super" wird wie bei Java dazu verwendet, Methoden einer Oberklasse aufzurufen.

```
class printable_farb_punkt y c =
   object (self)
   val c = c
   method farbe = c
   inherit printable_punkt y as super
   method print =
      print_string "(";
      super#print;
      print_string ", ";
      print_string (self#farbe);
      print_string ")";
   end
```

Die Klasse printable_farb_punkt erbt nun von den Klassen farb_punkt und printable_punkt. In der 5. Zeile wird mit dem übergebenen Parameter y der Super-Konstruktor aufgerufen. Die Methode print erweitert die print-Methode der

Oberklasse printable_punkt. Diese wird innerhalb der Methode dieser Unterklasse aufgerufen. Beim Aufruf von print, nachdem ein Objekt von printable_farb_punkt erzeugt wurde, bekommen wir die folgende Ausgabe:

```
# let p2 = new printable_farb_punkt 10 "blau";;
# p2#print;;
(10, blau)- : unit = ()
```

2.9 Fazit, Verwendung von OCaml in der Praxis

OCaml verfügt auf Grund seiner Herkunft über funktionale, imperative und zusätzlich objektorientierte Konzepte. Der große Vorteil von OCaml ist die Kombination vieler mächtiger Konzepte anderer Programmiersprachen:

- funktionale Programmierung
- Typinferenz
- automatische Speicherbereinigung
- Pattern-Matching
- Objektorientierung

Darüber hinaus wird der oben erwähnte Native-Code Compiler zur Verfügung gestellt, der sehr effizienten, mit C++ vergleichbaren Maschinencode erzeugt. Außerdem sind Bibliotheken (Standard-, Grafik-, Thread-,
Gui-Bibliotheken etc.) für eine Vielzahl von Plattformen (Windows, Unix etc.) erhältlich. Es können außerdem weitere bestehende Bibliotheken aus Unix oder Mac OS eingebunden werden, wie zum Beispiel GTK oder LAPACK (Linear Algebra Package).
OCaml hat sich mittlerweile zu einer der populärsten funktionalen Programmiersprachen entwickelt. Zum Schluss noch eine Auswahl an in OCaml geschriebenen Anwendungen:

Unison File Synchronizer

Ein Werkzeug für die Datei-Synchronisation unter Windows und Unix. Es ermöglicht, eine Ansammlung von Dateien und Verzeichnissen auf getrennten Rechnern unabhängig voneinander zu modifizieren und anschließend beide Kopien mit dem jeweils aktuellen Stand der Gegenseite abzugleichen. Änderungen einer Kopie werden dabei auf die andere übertragen.

MLdonkey

Ein multi-plattform peer-to-peer client. MLdonkey war der erste OpenSource Client, der auf das eDonkey-Netzwerk zugreifen konnte. MLdonkey läuft auf mehreren Betriebssystemen. Eine Stärke von MLdonkey ist, daß der Kern und die grafische Oberfläche getrennt sind und somit der Kern von einem anderen Rechner aus gesteuert werden kann.

LexFi's Modelling Language for Finance (MLFi)

MLFi ist die erste Sprache, die sehr genau Kapitalmärkte, Kredite und Investitionsprodukte beschreibt. Auf Grund der funktionalen Merkmale eignet sich OCaml besonders, um komplexe Datenstrukturen und Algorithmen zu beschreiben. Zusätlich profitiert MLFi von den mächtigen Listenfunktionen, da Listen eine zentrale Rolle im Finanzwesen einnehmen.

The Coq Proof Assistant

Das Coq-Tool ist ein System um formale mathematische Beweise zu manipulieren, oder die Korrektheit von Programmen zu zertifizieren.

Kapitel 3

F#
(Martin Lohrmann)

3.1 Einleitung

Dieser Artikel befasst sich mit der Programmiersprache F# und soll einen grundelegenden Eindruck über die Eigenschaften und Fähigkeiten geben. Der Artikel stellt keine Anleitung zum Erlernen der Sprache dar. Die Erklärungen gehen zum grössten Teil davon aus, dass Microsoft Windows als Betriebssystem zur Entwicklung genutzt wird, was an der Herkunft der Sprache liegt.
F# ist ein Mix aus funktionaler, imperativer, objektorientierter Programmiersprache von Microsoft für das .NET Framework, entwickelt von Don Syme und heute betreut von Don Syme und James Margetson. F# ist mit den Sprachen OCaml und ML verwandt, die Syntax der Sprachen ähneln sich daher stark. Genau genommen wurde OCaml als Vorlage für die Entwicklung von F# benutzt, die Ähnlichkeit ist daher kein Zufall.
ML und OCaml sind funktionale Programmiersprachen, wodruch auch F# dieses Paradigma erfüllt. Sie haben Eigenschaften wie statische Typisierung, Polymorphie, automatischer Speicherbereinigung und im Allgemeinen strenge Auswertung.Die Möglichkeit des objektorientierten Programmierens in F# ermöglicht die Interoperabilität mit anderen .NET Sprachen. Genutzt wird sie, um Datentypen ab zu bilden oder um User Interfaces zu programmieren. Aus der Zugehörigkeit zum .Net Framework resultiert, dass man aus F# heraus vollen Zugriff auf alle API's die im Framework enthalten sind, hat.
Durch die Verwandheit mit OCaml können einige OCaml- Programme direkt oder durch keine Änderungen in F# cross- kompiliert werden. über OCaml berichtet ein anderer Artikel.

3.2 Ziele und Einsatzgebiete

F# soll die Eigenschaften von ML in das .NET Framework integrieren, so dass es die Vorteile einer funktionalen Sprache mit denen eines solchen Frameworks vereint. Zielgruppen sind der wissenschaftliche und Ingenieurs- Bereich.

Mathematische Berechnungen, oder anspruchsvolle Analysen, wie die Prüfung von Hard- und Software gehören zu den Einsatzgebieten von F#

3.3 Werkrzeuge zur Entwicklung

Bevor auf die Eigenschaften und Fähigkeiten von F# genauer eingegangen wird, möchte ich an dieser Stelle einen kurzen Überblick über die verfügbaren Werkzeuge zur Entwicklung von F# geben. Eine allgemeine Dokumentation zu F# ist unter Microsoft Research[1] zu finden.

3.3.1 .NET Framework

Informationen zum Framework, den vorhanden Modulen und vielem mehr findet man im .Net Framework Developer Center[2]

3.3.2 der Compiler

Der F# Compiler kann kostenfrei von Microsoft Research[3] heruntergeladen werden und liegt derzeit in der Version 1.9.2.9 vom 31.07.2007 vor. Enthalten in dem Packet sind ein *Kommandozeilen- Compiler*, *F# für Visual Studio* sowie *F# Interactive*. Auf *F# Interactive* wird in einem späteren Abschnitt eingegangen.

3.3.3 die Entwicklungsumgebung

Als Entwicklungsumgebung für F# Anwendungen empfielt sich *Microsoft Visual Studio*, da wie oben bereits erwähnt, der Compiler ein entsprechendes Plugin mitliefert. Es enthält Syntaxhighlighting, Parsing und Typechecking

3.3.4 F# Interactive

F# Interactive, im weiteren *FSI* genannt, ist eine Kommandozeilen-Shell für F#, die es ermöglicht Programme zur Laufzeit zu schreiben. Diese kann entweder direkt über die Kommandozeile oder über ein Fenster in MS Visual Studio genutzt werden. Ein Befehl in der FSI, endet immer mit ";;".

3.3.5 Nutzung unter Linux

Um F# und andere auf .NET basierte Programmiersprachen auch untert Linux oder Mac OS X nutzen zu können, kann das Open Source Projekt *Mono*[4] genutzt werden, welches eine .Net- kompatible Laufzeitumgebung darstellt. Es wird hauptsächlich von Novell betrieben.

[1] http://research.microsoft.com/fsharp/fsharp.aspx
[2] http://msdn2.microsoft.com/de-de/library/aa139615.aspx
[3] http://research.microsoft.com/fsharp/release.aspx
[4] http://www.mono-project.com

3.4 Ein erstes Beispiel

Als ein erstes kurzes Beispiel für eine F# Anwendung soll hier ein "Hello World" Programm dienen. Zuerst erstellen wir eine Datei namens hello.fs. In diese schreiben wir nun folgenden Code:

```
let x =''Hello World'';;
val x : string
System.Console.WriteLine(x);;
Hello World
val it : unit = ()
```

Wir deklarieren also einen Variable x mit dem Wert "Hello World" und geben diese hinterher über die .NET- Methode *System.Console.printLine(x)* aus. Danach führen wir in einer Konsole das Kommando *fsc hello.fs* aus und erhalten eine Datei namens hello.exe. Diese kann nun wie gewohnt ausgeführt werden. Als Ergebnis erhalten wir auf der Konsole die Ausgabe "Hello World".
Eine noch kürzere Version sieht so aus:

```
printf "Hello World!\n"
```

Nun noch ein Beispiel für ein "Hello World" Programm, welches Winforms nutzt.

```
    open System
    open System.Windows.Forms

    let form = new Form()
    do form.Width  <- 400
    do form.Height <- 300
    do form.Text <- "Hello World Form"

    (* Menu bar, menus *)
    let mMain = form.Menu <- new MainMenu()
    let mFile = form.Menu.MenuItems.Add("&File")
    let miQuit  = new MenuItem("&Quit")
    let _ = mFile.MenuItems.Add(miQuit)

    (* RichTextView *)
    let textB = new RichTextBox()
    do textB.Dock <- DockStyle.Fill
    do textB.Text <- "Hello World\n\nCongratulations!"
    do form.Controls.Add(textB)

    (* callbacks *)
    let opExitForm sender args = form.Close ()
    do miQuit .add_Click (new EventHandler(opExitForm)
       )

    (* run! *)
```

```
26     do Application.Run(form)
```

Mit dem Befehl *open* werden Module in der Datei verfügbar gemacht. Danach wird eine Variable namens *form* vom Typ Form aus dem Modul *System.Windows.Forms* angelegt und ihre Werte *Width*, *Height* und *Text* entsprechend gesetzt. Anschliessend wird die Variable *mMain* erzeugt, welche auf das Menü des Formulars veweist. Darin wird nun ein Item *mFile* erzeugt, welches ein Item *miQuit* enthält. Zum Schluss wird noch das Event *Close* mit *miQuit* verknüpft und die Applikation ausgeführt mit der Methode *Application.Run(from)*. Das soll nun als kurzes Beispiel genügen.

3.5 Sprachfeatures

Im Folgenden werden einige Features der Sprache erläutert. Für Beispiele wird die FSI genutzt.

3.5.1 Variablen und Funktionen

Variablen und Funktionen werden in F# durch *let* oder *let rec* deklariert. Die Werte sind nach ihrer Deklaration nicht mehr änderbar. Der Scope von Variablen innerhalb einer Funktion endet mit ihrer Deklaration, das heißt nach *let x = 3* kann die selbe Anweisung noch einmal stehen und erneut eine Variable x, allerdings mit dem Wert 4 deklarieren. Eine Variablendeklaration könnte also wie folgt aussehen:

```
1 let x = 1+2;;
```

Um Werte als änderbar zu kennzeichnen dient das Schlüsselwort *mutable*. Um einem änderbaren Feld einen Wert zu zu weisen wird der ¡- Operator benutzt. Eine solche Zuweisung sieht dann folgendermaßen aus:

```
1 let mutable y = ``Hallo ich bin aenderbar'';;
2 y <- ``jetzt habe ich einen anderen Wert'';;
```

Funktionendeklarationen sehen einer Variablendeklaration sehr ähnlich. Sollte die funktion einen Parameter haben, ist sie gut von einer Variable zu unterscheiden. In diesem Fall steht vor der Zuweisung noch der Name des Paramters.

```
1 > let createAdder n = (fun arg -> n + arg);;
2 val createAdder : int -> int -> int
3 > let add10 = createAdder 10;;
4 val add10 : int -> int >
5 add10 32;;
6 val it : int = 42
```

Funktionen in F# können genau so wie andere Typen genutzt werden, sie können als Argument für andere Funktionen genutzt werden oder aber als Rückgabewert einer Funktion dienen. Man kann sogar listen von Funktionen erzeugen. Mit dem Schlüsselwort *fun* kann man anonyme Funktionen deklarieren.

int -¿ int -¿ int bedeutet, dass die Funktion *createAdder* ein int als Parameter übergeben bekommt, dann eine Funktion aufruft, die ebenfalls ein int als Paramter hat und einen Paramter zurückliefert.

Sequencing Operator

der Sequencing Operator eem—¿ erlaubt es, einen Wert als Parameter an eine andere Funktion zu übergeben, ohne diesen in die Parameterliste zu schreiben. An einem Beispiel wird die Arbeitsweise deutlich.

```
1 > let nums = [1; 2; 3; 4; 5];;
2 val nums : list<int>
3 > let odds_plus_ten =
4       nums
5       |> List.filter (fun n-> n%2 <> 0)
6       |> List.map (add 10)
7 val odds_plus_ten : list<int> = [11; 13; 15];;
```

List.filter übernimmt den Rückgabewert von *nums* und *List.map* übernimmt wiederrum den Rückgabewert von *List.filter*. Die Anweisungen werden genau in der Reihenfolge ausgeführt, wie sie angegeben wurden. Es werden also keine Optimierungsversuche unternommen.

3.5.2 Tuple

Ein Tuple ist ein einfacher Datentyp der 2 oder mehr Werte von beliebigen, verscheidenen Typen zusammen gruppiert.

```
1 let tuple = (42, "Hello world!");;
```

Das Tupel besteht nun also aus einem int und einem string. Da sich der Typ des Tupels ändert, sobald man einen anderen Wert hinzupackt, ist dieser Datentyp nicht für die Sammlung einer unbekannten Anzahl an Werten geeignet. Die einzelnen Werte kann man hinterher auch einzeln auslesen. Das geht wie folgt:

```
1 let (num, str) = tuple;;
```

In *num* und *str* sind nun die Werte der Felder des Tupels zu finden. Tupel ermöglichen die Rückgabe mehrerer Werte durch eine Funktion, ohne dass man komplexe Datentypen anlegen muss.

3.5.3 Records

Records können als Tupel mit benannten Einträgen gesehen werden. Man kann auf ihre Felder mit den vorher definierten Namen über die Punktnotation zugreifen. Außerdem kann bei der Definition eines Records der Datentyp der einzelnen Felder konkret festgelegt werden.

```
1 > // Deklaration eines Records
2 type Product = { Name:string; Price:int };;
3 > // erstellen eines Wertes vom Typ Product
```

```
let p = { Name="Test"; Price=42; };;
val p : Product
> p.Name;;
val it : string = "Test"
> // Erstellen einer Kopie von p mit einem anderen
    Namen
let p2 = { p with Name="Test2" };;
val p2 : Product
```

An dem Beispiel wird deutlich, dass die meisten Datentypen in F#unveränderlich sind. Man muss also eine Kopie anlegen, um einen oder mehrere Werte des Datentyps zu ändern. Dabei erleichtert das Schlüsselwort *with* eine Menge. Man braucht nur die Namen der zu ändernden Werte und die entsprechenden neuen Werte angeben.

3.5.4 Listen und Arrays

Eine Liste in F# wird durch *[]* gekennzeichnet. Man kann sowohl leere Listen als auch Listen mit Inhalt deklarieren.

```
> let nums = [1; 2; 3; 4; 5];;
val nums : list<int>
```

list ist ein generischer Typ und kann somit jeden F# -Typen beinhalten.
Arrays sind nicht Bestandteil von F#, sondern sind über das .NET Framework verfügbar. Man kann Arrays entweder durch Nutzung von *[—..—]* oder durch Nutzung von Funktionen aus dem Array- Modul wie *Array.create* erzeugen. Arrays sind grundsätzlich änderbar, man kann sie also mit dem *¡-* Operator bearbeiten. Im Beispiel bleiben wir bei der F# typischen Schreibweise.

```
> let arr = [| 1 .. 10 |]
val arr : array<int>
> for i = 0 to 9 do
    arr.[i] <- 11 - arr.[i]
> arr;;
val it : array<int
    > = [| 10; 9; 8; 7; 6; 5; 4; 3; 2; 1 |]
```

3.5.5 Referenz

Referenzen werden durch die Operatoren *1*, um den Wert der Referenz zu holen und *:=*, um der Referenz einen Wert zu zu weisen, dargestellt.

3.5.6 if-Anweisung

Eine if-Anweisung ist ähnlich den gewohnten Konstrukten aus anderen Programmiersprachen so aufgebaut:

```
if (x <= 0 or
    x >= 10) then
```

```
    printf "Blablubb";
```

3.5.7 Schleifen

Schliefen in F# sind ähnlich, wie in anderen Sprachen. Eine While- Schleife sieht zum Beispiel so aus:

```
while (!x > 3) do
  x := !x - 1;
  printf "x = %s\n" !x;
done;
```

Eine For-Schleife sieht wie folgt aus:

```
for i = 0 to Array.length(x) - 1 do
  printf "x[%d] = %d\n" i x.(i);
done;
```

3.5.8 Object Types

"Object Types" ist die F# Bezeichnung für Klassen. Man kann diese auf .NET kompatible Art und Weise compilieren, so dass sie hinterher von jeder anderen Sprache des .NET Frameworks aus nutzbar sind. Ausserdem kann man beliebige Klassen, die in anderen .NET Sprachen geschrieben sind erweitern. Klassen in F# können maximal eine Oberklasse haben und können beliebig viele Interfaces implementieren. Unterklassen können in ihre Basisklassen gekastet werden. Alle Klassen stammen von der Basisklasse *obj* ab, welche ein Synonym für die .NET CLR Klasse *System.Object* ist. Object Types können Felder, Konstruktoren, Methoden und Properties haben. Eine KLasse wird als type deklariert, der mit *class* als Klasse identifiziert wird.

```
type TestKlasse(n:int)=
   class
      let mutable datum = n + 1
      do printf "erzeuge Testklasse(%d)" n

      member x.Datum with
         get() = datum
         and set(v) = datum <- v

      member x.Print() =
         printf "Datum: %d" datum

      override x.ToString() =
         sprintf "(Datum: %d)" datum

      static member FromInt(n) =
         TestKlasse(n)
   end
```

Die Klasse *TestKLasse* hat ein änderbare Feld *datum*, eine dazugehörige Property *Datum* mit Getter und Setter, eine Instanzmethode *Print()*, welche Das Datum ausgiebt und eine statische Methode *fromInt*. Ausserdem überschreibt sie die Methode *ToString* von der Basisklassse obj. Die Klasse hat einen impliziten Konstruktor, der es erlaubt, den Code direkt in die Typdeklaration zu schreiben. Explizite Konstruktoren sind ebenfalls möglich. Diese haben die gleiche Syntax, wie andere Member- Methoden.

Vererbung

Eine Klasse erbt von genau einer anderen Klasse. Solange nicht explizit angegeben, erbt eine Klasse immer direkt von der Basisklasse System.Object. Um von einer anderen Klasse zu erben, muss diese mit dem Befehl *inherit ¡Klassenname¿* angegeben werden.

Abstract Object Types

Abstract Object Types sind Interfaces. Sie sind durch das Schlüsselwort *interface gekennzeichnet*. Ein Interface besteht nur aus Methoden, die als abstrakt gekennzeichnet sind.

```
type TestInterface = interface
   abstract Datum : int with get, set
   abstract Print : unit -> unit
end
```

Eine Klasse, die dieses Interface implementiert sieht dann folgendermassen aus:

```
type ImplInter(n:int)= class
   let mutable datum = n + 1
   interface TestInterface with
      member x.Datum with
         get() = datum
         and set(v) = datum <- v

      member x.Print() =
         printf "Datum: %d" datum
   end
end
```

nachträgliches Hinzufügen von Members zu F# Types

Es ist möglich, nach der Deklaration eines Typs einzelne Member hinzu zu fügen. Man kann zum Beispiel außerhalb eines Typs eine Funktion schreiben und diese dann über den Typ verfügbar machen. Ein zuvor deklarierter Typ *TestTyp* kann also um eine Methode erweitert werden indem man in der gleichen Datei wie der Typ per *type TestTyp with* weitere member- Methoden hinzufügt, die zuvor deklarierte Funktionen nutzen. Die Erweiterung muss in der gleichen Datei wie die Typdeklaration stehen, sonst hat sie keinen Zugriff auf die interne Implementierung des Typs. Es ist aber dennoch möglich, zum Beispiel um eine

Funktion für die Arbeit mit einem Typen über die Punktnotation verfügbar zu machen.

```
type Variant = | Num of int | Str of string

let print x =
   match x with
   | Num(n) -> printf "Num %d" n
   | Str(s) -> printf "Str %s" s

type Variant with
   member x.Print() = print x
```

Es wurde ein type namens *Variant* deklariert, der entweder ein int oder einen String enthält. Danach wird die funktion *print* deklariert, die je nach Typ des übergebenen Parameters einen Text ausgibt. Zum Schluss wird nun das oben erwähnte Konstrukt genutzt, um einem Typen eine weitere Methode hinzu zu fügen. Diese ruft die zuvor deklarierte print- Funktion auf.

Type casts

F# unterstützt das Casten von Object Types in ihre Basisklasse, das nennt man upcast *object :¿ Zieltyp*, downcast *object :?¿ Zieltyp* und einen dynamischen Typcheck *object :? Zieltyp*, der prüft, ob es möglich ist ein Objekt in ein anderes zu casten.

Kapitel 4

Scala (Fabian Keller)

4.1 Einleitung

4.1.1 Was ist Scala

Scala ist die Abkürzung für Scalable Language. Sie wurde am Programming Methods Laboratory am Ecole Polytechnique Federale de Lausanne (EPFL) entwickelt. Scalas Hauptaspekt umfasst eine objektorientierte und funktionale Multiparadigmensprache. Als Runtime Enviroment dient meistens eine Java Virtual Machine, da sich die .Net Version noch in der Entwicklung befindet. Genauso wie Java ist Scala somit auf vielen Plattformen lauffähig. Ebenso können die Java - Klassenbibliotheken direkt in Scala verwendet werden. Scala ist im Gegensatz zu Java strikt objektorientiert, denn jeder Wert und jede Funktion ist ein Objekt. Die Sprache stellt bekannte objektorientierte Mechanismen wie Vererbung und Interfaces (Traits) zur Verfügung. Aus der funktionalen Welt stehen Mechanismen wie Higher-Order-Functions[1], Pattern-Matching[2], anonyme Funktionen[3] und viele weitere zur Verfügung.

Im weiteren soll Scala hauptsächlich im Vergleich zu Java betrachtet werden.

4.1.2 Geschichte

Scala wird seit 2001 von Martin Odersky am EPFL in der Schweiz entwickelt. Seit 2003 befindet sich Scala in der Version 2 und ist aktuell mit dem Release 2.6.0 verfügbar. Ein Compiler für .Net befindet sich in der Entwicklung.

[1] Es ist möglich Funktionen als Argument zu übergeben.

[2] Ist die Möglichkeit zu ermitteln, ob sich ein gegebenes Muster (Pattern) in einem Suchbereich wiederfindet.

[3] Sind Funktionen die auf Grund ihrer kürze oder ihrer einmaligen Verwendung nicht Benannt werden. Auch Lambda-Funktionen genannt.

4.1.3 Ziele bei der Entwicklung

Die Scala Entwickler hatten zwei Hauptaspekte für die Entwicklung einer neuen Sprache:

- *Grund 1:* Es sind zum Teil fundamentale Paradigmenwechsel nötig, da immer wichtiger werdende Software wie z.B. Webservices dies erfordern.
- *Grund 2:* Und durch eben diese Paradigmenwechsel werden wiederum neue Programmiersprachen gebraucht. Ähnlich wie damals bei den grafischen Benutzeroberflächen und der Objektorientierung.[Ode07c]

Zudem sollten Synergieeffekte durch Verknüpfung von bewährten Paradigmen (Objektorientierung und funktionale Welt) geschaffen werden.Auch die (spätere) Erweiterbarkeit (scalability), was im Name Scala schon zum Ausdruck kommt, war wichtig. Neue Sprachkonstrukte sollten leicht in Scala hinzufügen bzw. erweitern werden können.

4.2 Verfügbare Compiler und Interpreter

Scala wird mit einem eigenen Compiler (scalac) und Interpreter (scala[int]) ausgeliefert. Das Interessante dabei ist, dass der Compiler nur ein Wrapper für den Javacompiler „javac“ ist und somit Java Bytecode erzeugt. Alle benötigten Verweise auf jar Dateien werden durch ihn automatisch hinzugefügt und an javac übergeben.
Zur Entwicklung gibt es bereits mehrere Werkzeuge. Unter anderem steht ein Eclipse-Plugin zur verfügung.

Eine kurze Installationsanleitung wie in wenigen Minuten eine lauffähige Scala Entwicklungsumgebung installieren kann, befindet sich im Anhang.

4.2.1 Der Interpreter

Der Interpreter kann unter Debian Etch mit dem Befehl scala aufgerufen werden. Unter anderen Distributionen kann dieser Aufruf auch scalaint heissen.

Bitte beachten sie, dass das anlegen des Objekts „HelloWorld“ in einer Zeile steht und hier nur wegen der Optik umgebrochen wurde.

```
1 www:~/scala# scala
2 This is an interpreter for Scala.
3 Type in expressions to have them evaluated.
4 Type :quit to exit the interpreter.
5 Type :compile followed by a filename to compile a
    complete Scala file.
6 Type :load followed by a filename to load a sequence
    of interpreter commands.
7 Type :replay to reset execution and replay all
    previous commands.
8 Type :help to repeat this message later.
9
```

```
10 scala> object HelloWorld extends Application { Console
      .println("Hello, world!") }
11 defined module HelloWorld
12
13 scala> HelloWorld
14 Hello, world!
15 line1: HelloWorld.type =
      line0$object$HelloWorld$@1fe8ce8
16
17 scala>
```

4.2.2 Der Compiler

Zum Compilieren des Beispiels, dass in der Datei hello.scala abgelegt wurde können folgende Zeilen verwendet werden:

```
1  www:~/scala# cat hello.scala
2  object HelloWorld extends Application {
3          Console.println("Hello, world!")
4  }
5
6  www:~/scala# scalac hello.scala
7  www:~/scala# ll
8  insgesamt 12K
9  -rw-r--r-- 1 root root   79 2007-12-25 15:08 hello.
      scala
10 -rw-r--r-- 1 root root  626 2007-12-25 15:16
      HelloWorld.class
11 -rw-r--r-- 1 root root 1,1K 2007-12-25 15:16
      HelloWorld$.class
12 www:~/scala# scala -classpath . HelloWorld
13 Hello, world!
14 www:~/scala#
```

Man sieht auch das sich der Objektname aus der Datei geholt und dieses als .class Datei erstellt wird.

4.2.3 Scala als Shellscript

Man kann Scala auch in ein kleines Shellscript verpacken und somit als Interpretersprache, ähnlich wie von Perl oder PHP-Cli bekannt, benutzen.

```
1 #!/bin/sh
2 exec scala "$0" "$@"
3 !#
4 object HelloWorld {
5    def main(args: Array[String]) {
6       println("Hello, world!")
7    }
8 }
```

Das ganze abgespeichert, in z.B. script.sh kann dann, mit den entsprechenden Rechten versehen wie ein normales Programm gestartet werden.

4.3 Ausführungsmodell

Scala setzte anfangs ausschließlich auf Java auf. Daher entspricht das Ausführungsmodell auch dem von Java. Es wird allerdings durch ein weiteren Schritt ergänzt. Der Scala Compiler kümmert sich um die Abhängigkeiten und erzeugt mit seiner Übergabe an javac dann Java Bytecode. Dieser wird schließlich in der virtuellen Maschine von Java ausgeführt. Wie im letzten Kapitel gezeigt, kann Scala somit als Compilierte class Datei laufen oder mittels des Interpreters und einem kleinen Shellscript als quasi Scriptsprache genutzt werden.

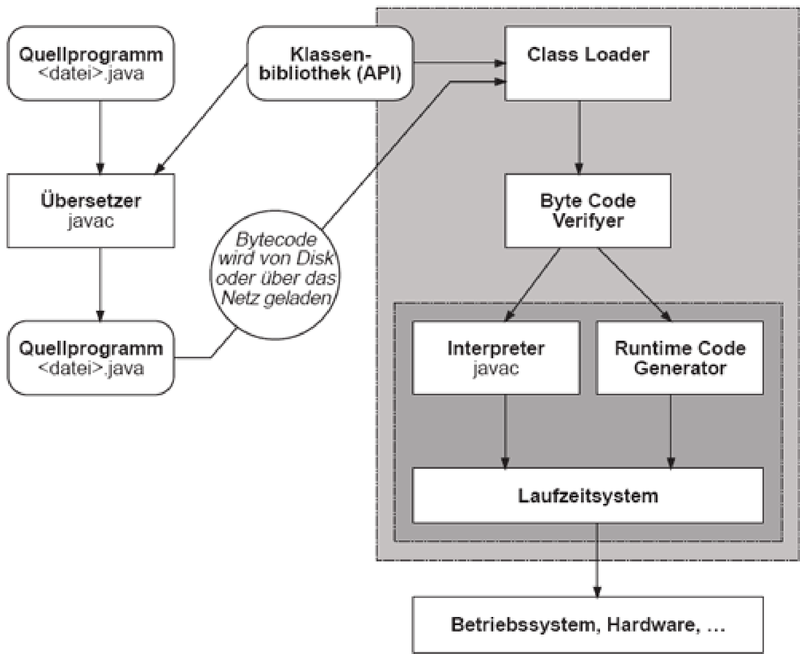


Abbildung 4.1: Das Ausführungsmodell von Java[4]

4.4 Allgemeine Sprachkonstrukte

Zunächst sollen einmal allgemein bekannte Sprachkonstrukte betrachtet werden. Sie stellen keine Besonderheit von Scala dar, helfen aber bei dem späteren Verständnis der Beispiele.

[4]http://www.ifi.uzh.ch/rerg/fileadmin/downloads/staff/berner/programmieren_in_java.pdf

4.4.1 Variablendeklaration

Anders als in Java muss in Scala in den meisten Fällen keine explizite Typendeklaration stattfinden. Mehr dazu im Kapitel Typensysten.

Das setzen einer Variablen auf den Integerwert 3, sähe also so aus:

```
1 var x = 3
```

Wenn man das ganze im Scala Interpreter eingibt, zeigt einem dieser automatisch den Typ der Variablen an

```
1 scala> var x = 3
2 x: scala.Int = 3
```

Und das Ganze mit einer expliziten Deklaration:

```
1 var x: Int = 3;
```

Die Interpreterausgabe sieht gleich aus

```
1 scala> var x: Int = 3
2 x: scala.Int = 3
```

4.4.2 Bedingungen

Mit eine der wichtigsten Konstrukte ist natürlich die Bedingung. Auch in Scala ist sie mit dem Schlüsselwort if belegt. Nicht verwunderlich ist dann auch, dass sie genau so wie in Java, C oder C++ benutzt werden kann.

```
1 var i = 4
2 if (i < 4) {
3    Console.println("klein");
4 }
5 else if (i <= 7) {
6    Console.println("mittel");
7 }
8 else
9    Console.println("gross");
```

4.4.3 Schleifen

Schleifen sind fast identisch zu den Javafunktionen. Hier nur ein paar kleine Beispiele die die Zahlen 1 bis 4 ausgeben.

Die While Schleife:

```
1 var i = 1
2 while ( i < 5 ) {
3    Console.println(i);
```

```
    i = i + 1;
}
```

Die For Schleife sieht etwas anders aus:

```
for (val i <- Iterator.range(1, 5)) Console.println(i)
    ;
```

Mit dieser Besonderheit kann Scala in der For Schleife ähnlich dem Pattern Matching, dass später näher erklärt wird, auf einem erzeugten Objekt Bedingungen anwenden.
Es wird über die Liste oder das Objekt persons iteriert. In jedem Schritt steht das aktuelle Objekt in p. Die Bedingung prüft auf dem Objekt p dann, ob diese zutrifft. Durch yield wird ein Array erzeugt, dass die Elemente p.name enthällt.

```
for (p <- persons if p.age > 20) yield p.name
```

4.4.4 Arrays

Leider sind auch in Scala Arrays nicht dynamisch. Hier wird ein Array mit 5 Elementen angelegt. An zwei Stellen wird es befüllt und danach ausgegeben.

```
var arr = new Array[Int](5)

arr(2) = 2
arr(4) = 9

for(val e <- Iterator.fromArray(arr)) {
    Console.println(e);
}
```

4.4.5 Methoden

Um eine Methode in Scala zu definieren gibt es das def Konstrukt. Man kann ähnlich wie in Java Parameter definieren und muß am Ende den Rückgabetyp unit, der void entspricht, angeben. Interessant ist auch die Einheitlichkeit. Die Parameterangaben aber auch das gesamte Methodenkonstrukt entspricht der Schreibweise einer Deklaration von z.B. Variablen. Erst das Schlüsselwort (hier def), dann der Objektname gefolgt von einem Doppelpunkt. Anschließend eine Typangabe. Bei einer Zuweisung folgt noch ein Gleichzeichen.

```
def test (x: Int): unit = {
    Console.println ("Ausgabe: " +x)
}
```

4.4.6 Funktionen

Man konnte es schon erahnen, aber Funktionen sind das eigentliche Sprachelement hinter Methoden. Sie sind genau so aufgebaut wie diese, haben aber

anstelle von unit einen „echten“ Rückgabeparameter. In diesem Beispiel ist die Funktion verkürzt und ohne Klammern geschrieben.

```
def test2 (x: Int): String = "[" + x + "]";
```

Wenn man sich dieses Beispiel zudem einmal im Interpreter anschaut, sieht man die Interaktion mit Java aus dem der Typ String genutzt wird.

```
scala> def test2 (x: Int): String = "[" + x + "]";
test2: (scala.Int)java.lang.String

scala> test2(5)
line7: java.lang.String = [5]

scala>
```

4.4.7 Infixnotation

Eine weitere interessante Möglichkeit in Scala ist Methoden, mit nur einem Parameter, als Infix-Operator zu verwenden.

Der Ausdruck

```
val x = a + 3;
```

ist somit nicht anderes als der Methodenaufruf und die verkürzte Schreibweise von

```
val x = a.+(3);
```

Die standard Operatoren wie Plus, Minus, usw. können beliebig überschrieben werden. Das gleiche gilt auch für die Typumwandlungsfunktionen. Ein Beispiel dazu ist im nächsten Abschnitt „Objektorientiertes Paradigma“ zu finden.

4.5 Typensystem

Scala ist statisch getypt und besitzt ein *striktes* Typensystem. Jede Funktion repräsentiert einen Wert. Jeder Wert stellt ein Objekt dar. Daher ist jede Funktion gleichzeitig auch ein Objekt.

4.5.1 Typen

Die bekannten Datentypen sind in Scala als Objekte vordefiniert.

```
type byte = scala.Byte
type short = scala.Short
type char = scala.Char
type int = scala.Int
type long = scala.Long
type float = scala.Float
```

```
type double = scala.Double
type boolean = scala.Boolean
type String = java.lang.String
```

4.5.2 Superklassen

Neben der allgemeinen Superklasse Any, gibt es noch den Referenzsupertyp AnyRef, unter dem sich alle Klassendatentypen wie Object ud String einordnen. Integer Zahlen wie 42 oder Floats die nicht wie in Java, zum Sprachbestandteil gehören sind von AnyVal abgeleitet. Da diese Werttypen keine primitive Typen (im eigentlichen Sinne) sind, können sie demnach beliebige Funktionen wie z.B. +, - und testAufPrimzahl haben. Da sie Objekte darstellen können sie natürlich auch beliebig erweitert und überschreiben werden.

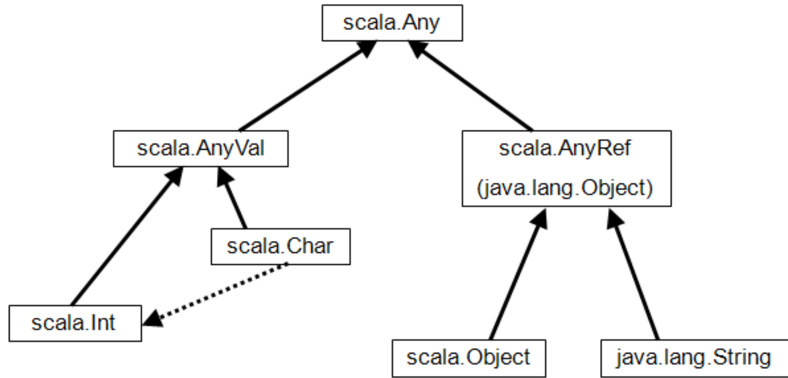


Abbildung 4.2: Ausschnitt aus dem Klassenbaum[5]

4.6 Objektorientiertes Paradigma

Hier sollen nun ein paar der wichtigsten Eigenschaften der Objektorientierung vorgestellt werden. Die Objektorientierung ist mit dem Funktionalen Paradigma eines der Haupteigenschaften von Scala.

4.6.1 Klassen, Objekte und Vererbung

Eine weitere Besonderheit von Scala ist, dass es im Gegensatz zu Java *keinen expliziten Konstruktor* besitzt und Parameterübergaben erlaubt.
Ausserdem sieht man in diesem Beispiel wie Methoden überschrieben werden. Sie sind mit dem Schlüsselword *override* gekennzeichnet.

```
class Point(xc: Int, yc: Int) {
  val x: Int = xc;
  val y: Int = yc;

```

[5] http://www.wi.uni-muenster.de/pi/lehre/SS04/SeminarProgrammiersprachen/VortragScala.ppt

```
5    def move(dx: Int, dy: Int): Point = new Point(x +
        dx, y + dy);
6 }
7
8 class ColorPoint(u: Int, v: Int, c: String) extends
    Point(u, v) {
9    val color: String = c;
10
11    def compareWith(pt: ColorPoint): Boolean =
12       (pt.x == x) && (pt.y == y) && (pt.color == color
           );
13
14    override def move(dx: Int, dy: Int): ColorPoint =
        new ColorPoint(x + dy, y + dy, color);
15 }
```

4.6.2 Mixin Klassen

Mixin-Klassen sind Scalas Möglichkeit zur Mehrfachvererbung. Ein Vorteil von dieser Variante ist, dass Namenskonflikte die dann entstehen, wenn mehrere Unterklassen gleichnamige Methoden mit identischer Signatur implementieren, vermieden werden.
Interessanterweise ist somit in diesem Beispiel, dass auf den vorherigen Klassen aufbaut, im Objekt ColoredPoint3D keinerlei Sourcecode nötigt.

```
1 class Point3D(xc: Int, yc: Int, zc: Int) extends Point
     (xc, yc) {
2    val z = zc;
3    override def toString() = super.toString() + ", z
        = " + z;
4 }
5
6 class ColoredPoint3D(xc: Int, yc: Int, zc: Int, col:
    String) extends Point3D(xc, yc, zc) with
    ColoredPoint (xc, yc, col);
```

4.6.3 Traits

Traist können wie Java Interfaces gesehen werden, mit dem Unterschied, dass Trais teilimplementiert sein können.

```
1 trait TraitTest {
2    def abstractMethod(x: any): unit;
3    def concretMethod(x: any): String = x.toString();
4 }
```

4.6.4 Generische Klassen

Generische Klassen können analog zu anderen Programmiersprachen angelegt werden. Hier wird ein Stack vom Typ Int(eger) erzeugt.

```
class Stack[T] {
   var elems: List[T] = Nil;
   def push(x: T): Unit = elems = x :: elems;
   def top: T = elems.head;
   def pop: Unit = elems = elems.tail;
}
val s = new Stack[Int];
s.push(1);
```

4.6.5 Singleton-Objekte

In diesem einfachen Beispiel soll gezeigt werden, dass es in Scala ein weiteres Blockelement namens *object* gibt. Dieses wird hier mit dem Schlüsselwort *with* und *Application* verknüpft und macht somit alles was in diesem „object“ steht ausführbar. Die aus Java bekannte main Methode kann hier allerdings optional verwendet werden, ist aber nicht zwingend nötig.

```
object HelloWorld with Application {
   println("Hello, world!")
}
```

Alternativ wäre die eher aus Java bekannte Vorgehensweise wie folgt:

```
object HelloWorld {
   def main(args: Array[String]): unit = println("
      Hello, world!")
}
```

Scala unterstützt ausserdem Sub-Klassen und kann somit Klassen in Klasse schachteln. Das gleiche gilt für Methoden die geschachtelt werden können.

4.7 Funktionales Paradigma

Scala enthält als sein zweites wichtiges Paradigma funktionale Konzepte, von denen hier einige verdeutlicht werden sollen.

4.7.1 Anonyme Funktionen

Anonyme Funktionen sind unter anderem auch als Lambda-Funktionen bekannt. Sie bezeichnen relativ kurze Funktionen deren Referenz nur einmalig benötigt wird. Um sich das Benennen solcher Funktionen zu sparen, wurden die Anonymen Funktionen mit in Scala übernommen.
Ein einfaches Beispiel für eine Anonyme Funktion ist dieses, dass einfach nur zwei Interger Zahlen addiert und keine eigene Funktion definiert.

```
(x: Int,y: Int) => x + y
```

Im folgenden stehen einige Beispiele für Anonyme Funktionen die in Scala verkürzt geschrieben werde können. Auf der rechten Seite steht jeweils die Anonyme Funktion. Das Leerzeichen zwischen dem _ und dem Methodennamen ist erforderlich, da sonst dieser zum Namen hinzuzählen würde.

```
Math.sin _            x => Math.sin(x)

Array.range _         (x1, x2) => Array.range(x1, x2)

List.map2 _           (x1, x2) => (x3) => List.map2(x1
    , x2)(x3)

List.map2(xs, ys)_    x => List.map2(xs, ys)(x)
```

4.7.2 Currying

Des weiteren unterstützt Scala Currying. Dabei besitzen Methoden mehrere Parameterlisten. Ein Aufruf einer solchen Methode mit zu wenig Parameterlisten liefert eine Referenz auf die Methode mit bereits teilweise übergebenen Parametern, die nur noch die fehlenden Parameter erwartet.

```
class curry {
   def modN(n: Int)(x: Int) = ((x % n) = 0);
}
```

Ein Aufruf von modN(2) ergibt somit eine Referenz auf eine Methode, die der folgenden entspricht:

```
class curry {
   def modN(x: Int) = ((x % 2) = 0);
}
```

4.7.3 Higher Order Functions

Higher Order Funktionen ermöglichen es Funktionen als Parameter zu übergeben. Dabei handelt es sich hier um die generische Klasse *Decorator*, die einfach nur ein übergebenes Objekt formatiert ausgibt. Die Formatierung besteht dabei aus 2 Strings, von denen jeweils einer vor und einer nach dem Objekt angehängt wird. Auf dem Objekt selbst, dass in der Variablen x liegt, wird dann nur noch die toString() Methode ausgeführt. Beim Aufruf der Funktion spielt es keine Rolle ob sich diese in einer Klasse befindet. Der kleine Wrapper app ruft da nur noch die übergebene Funktion mit dem übergebenen Parameter auf.

```
class Decorator(left: String, right: String) {
   def layout[A](x: A) = left + x.toString() + right;
}

```

```
5  object FunTest with Application {
6     def app(f: Int => String, v: Int) = f(v);
7  
8     val decorator = new Decorator("[", "]");
9  
10    Console.println(app(decorator.layout, 7))
11 }
```

4.7.4 Pattern Matching

Im folgenden Beispiel wird über die Übergabeparameter, die als String-Array vorliegen, iteriert. In jedem Iterationsschritt liegt das aktuelle Objekt vom Typ String in der Variablen a. Dort wird zunächst mittels Pattern-Matching für jedes Element betrachtet, ob der String einem zulässigen Parameter entspricht oder ansonsten in den default Case (*case x*) gesprungen. Hier zulässig währen -h und -v.

```
1  object Main {
2     var verbose = false
3  
4     def main(args: Array[String]) {
5        for (a <- args) a match {
6           case "-h" | "-help"    =>
7              println("Usage: scala" + "Main [-help|-
                  verbose]");
8           case "-v" | "-verbose" =>
9              verbose = true;
10          case x =>
11             println("Unknown option: '" + x + "'");
12       }
13       if (verbose)
14          println("How are you today?");
15       }
16    }
17 }
```

4.8 Webservices Paradigma

In jüngster Vergangenheit wurde das Paradigma der Webservices immer wichtiger. Diese setzen u.a. mit SOAP und vergleichbaren standardisierten Webschnittstellen eine XML Verarbeitung zur Kommunikation vorraus. Dies wurde, im Gegensatz zu vielen anderen (modernen) Sprachen, in Scala direkt mit integriert und soll hier näher betrachtet werden.

4.8.1 XML Processing

Im folgenden Beispiel wird einen XML Baum mit verschiedenen Elementen erzeugt.

```
object verboseBook {

    import scala.xml.{ UnprefixedAttribute, Elem, Node
        , Null, Text, TopScope }

    val pbook =
        Elem(null, "phonebook", Null, TopScope,
            Elem(null, "descr", Null, TopScope, Text("
                This is a "),
                Elem(null, "b", Null, TopScope, Text("
                    sample")),Text("description")
            ),
            Elem(null, "entry", Null, TopScope,
                Elem(null, "name", Null, TopScope, Text("
                    Burak Emir")),
                Elem(null, "phone",
                    new UnprefixedAttribute("where", "work
                        ", Null), TopScope, Text
                        ("+41 21 693 68 67"))
            )
        )

    def main(args: Array[String]) = Console.println(
        pbookVerbose )
}
```

Da diese Variante allerdings zumeist sehr unleserlich ist, wurde in Scala direkt die XML Notation mit integriert. Somit kann man auch die Variable pbook wie folgt anlegen. Man beachte, dass es sich nicht um einen String handelt. Der XML Baum wird nicht durch Anführungszeichen begrenzt, sondern direkt eingegeben.

```
    val pbook =
    <phonebook>
        <descr>This is a <b>sample</b>description</descr
            >
        <entry>
            <name>Burak Emir</name>
            <phone where="work">+41 21 693 68 67</phone>
        </entry>
    </phonebook>;
```

Das Abspeichern in eine XML Datei funktioniert demnach genau so einfach. Es ist ähnlich wie in Java in Verbindung mit DOM, wenn man ein serialisiertes XML Objekt abspeichert. Die funktion XML.save() erledigt alles und speichert den kompletten XML Baum in eine Datei als Plain XML, der genau so aussieht wie das zweite Beispiel ohne Umbrüche.

```
object Foo with Application {
    val pbook = ...
    scala.xml.XML.save("myfile.xml", pbookVerbose);
```

```
}
```

Zum Laden des gesamten Baums genügt die Funktion XML.loadFile(). Da in Scala die XML Notation direkt mit integriert ist, muss in dem Sinne auch kein Parsing gemacht werden. Es wird vielmehr einfach nur ein Objekt mit dem Inhalt angelegt, welcher dann einem XML Baum entspricht.

```
object Foo with Application {
   val x = scala.xml.XML.loadFile("myfile.xml");
   Console.println(x);
}
```

Wenn man mit Scala und XML DTDs arbeiten will, wird einem das Tool *schema2src* zur Verfügung gestellt. Mit diesem Tool werden einem alle benötigten Klassen und Funktionen erstellt, die zum verarbeiten der DTD nötig sind.

4.9 Beispiele

In diesem Teil des Artikels sollen ein paar allgemeine Beispiele zu Scala gezeigt werden. Unter anderem werden hier auch ein paar interessante Möglichkeiten gezeigt, die in anderen Programmiersprachen und vor allem Java nicht ohne weiteres möglich sind.

4.9.1 Interaktion mit Java

In Scala können alle Klassen und Bibliotheken die aus Java bekannt sind genutzt werden. Dabei werden nicht nur die Funktionen und Methoden importiert sondern auch alle Symbole und Konstanten.

Im Gegensatz zu Java wird in Scala nicht der Stern als Wildcard genutzt sondern der *Unterstrich* _. Ein weiterer Unterschied ist, dass es in Scala kein void gibt. Als Pondant ist hier das Schlüsselwort *unit* zu sehen.

```
object BigFactorial {
   import java.math.BigInteger, BigInteger._;

   def fact(x: BigInteger): BigInteger =
      if (x == ZERO) ONE
      else x multiply fact(x subtract ONE);

   def main(args: Array[String]): unit =
      System.out.println("fact(100) = " + fact(new
         BigInteger("100")));
}
```

Mit dem ersten Teil des Import-Ausdrucks wird die Klasse BigInteger importiert. Um diese zu instanzieren wird allerdings noch der Konstruktor benötigt, der in Scala nicht ohne weiteres bekannt ist und nur eine Methode mit dem Namen der Klasse darstellt. Dazu importiert der zweite Teil des Import-Ausdrucks sämtliche Namen, die in der Klasse BigInteger bekannt sind.

4.9.2 Setterfunktion

Dieser Codeausschnitt zeigt wie man sogenannte Setterfunktionen realisieren kann. Die Setterfunktion heißt genau so wie der „getter“, mit dem Unterschied, dass in ihrem Funktionsnamen ein _ (Underscore) angehängt wird. Diese Funkion wird dann nur in schreibender Richtung benutzt, um z.B. die Übergabeparameter zu prüfen. In lesender Richtung wird hier nur die Variable h zurückgegeben.

```
object Test {
   class DateError extends Exception

   class TimeOfDayVar {
   private var h : Int = 0

   /* getter */
   def hours = h

   /* setter */
   def hours_ = (h: Int) =
      if (0 <= h && h < 24) this.h = h
      else throw new DateError()
   }
   def main(args: Array[String]) {
      val time = new TimeOfDayVar

      /* Man beachte es wir kein _ mit angegeben
       * Die Setterfunktion wird aber aufgerufen */
      time.hours = 8;

      /* Der nächste Aufruf schmeißt eine
       * DateError Exception  */
      time.hours = 25
   }
}
```

4.10 Zusammenfassung

Scala ist nicht, wie von mir erst fälschlicherweise gedachtes, ein „besserer Abklatsch von Java“. Wenn man sich einmal die nützliche Möglichkeiten anschaut, sieht man das Potential das hinter Scala steckt. Dinge die man in Java teilweise nur auf Umwegen realisieren kann, lassen sich so einfach und elegant Programmieren. Auch Benchmarks, die bei so einer junge Sprache nicht selbstverständlich sind, belegen, dass Scala durch seinen Aufsatz auf Java nur minimal langsamer ist als seine Wirtssprache. Es lassen sich aber die gleichen Problemstellungen teilweise nur mit einem Bruchteil der Lines of Code realisieren.
Es gibt aber auch ein paar Dinge, wie in jeder Sprache, die einem nicht verständlich erscheinen. Hierzu zählen z.B. dass Umbenennen des Schlüsselworts void in unit, was sich mir leider auch nicht aus der offiziellen Dokumentation erschloß.
Direkt eingebaute und unterstützte Features wie die XML-Verarbeitung machen

Scala zumindest Zukunftssicher und auch sicherlich für den einen oder anderen Programmierer interessant.

4.11 Anhang

4.11.1 Installation (Quickstart)

Hier soll nur kurz beschrieben werden, wie man schnell mal Scala ausprobieren kann. Als Testbetriebssystem wurde Debian gewählt, da hier alle Pakete verfügbar sind und die Installation nur wenige Minuten benötigt. Da Scala in dieser Installation auf Java aufsetze soll, wird zunächst die Java Runtime Enviroment(JRE) und das Java Development Kit (JDK) benötigt. Danach kann Scala über apt installiert werden. Bitte denken sie daran, dass Java nicht als Open Source gilt und sie somit in ihren Quellen *non-free* und *contrib* hinzufügen.

Kapitel 5

Python (Robert Rauschenberg)

5.1 Einleitung

(PYTHON SYMBOL EINFÜGEN)

In diesem Artikel wird nun im folgenden ein Einblick in die Programmiersprache Python gegeben. Ich werde hier die Geschichte dieser Sprache genauso ansprechen wie ich auch aktuelle Beispiele für die Anwendung geben werde. Des weiteren werden die einzelnen Aspekte der vor allem bei Design und Medieninformatikern sehr beliebten Sprache aufgezeigt.

5.2 Geschichte

In diesem Kapitel werde ich auf die Geschichte um die Entstehung von Python und die Namensgebung der Sprache eingehen.

5.2.1 Woher kommt Python ?

Python wurde Ende der 80er, Anfang der 90er Jahre in Amsterdam am 'Centrum vor Wiskunde en Informatica' von Guido van Rossum entwickelt. Van Rossum wollte sie als Nachfolger der Programmierlehrsprache ABC entwickeln ursprünglich für das verteilte Betriebsystem Amoeba.

5.2.2 Python der Name

Python als Name hat etwas Vertrautes an sich. Das liegt zum einen daran das es eine Schlangenart gibt, welche diesen Namen trägt. Des weiteren gibt es eine Gruppe englischer Komiker mit dem Namen 'Monty Pythons Flying Circus'. Da Guido van Rossum mehr ein freund des Lachens, als der Schlangen war wurde diese Sprache auch nach dem 'Flying Circus' benannt.[1] Aufgrund der Nähe zum Circus finden sich auch viele Anspielungen auf Sketche aus dem 'Flying Circus'.

[1] Ob van Rossum auch eine Beziehung zu den Schlangen hatte ist nicht bekannt.

5.2.3 Weiteres

Weitere Informationen zu der aktuellen Entwicklung von Python, wie Beispielsweise die aktuelle Version und weitere Entwicklungen werde ich in einem Extra Kapitel behandeln.

5.3 Python wofuer steht es?

Python als Sprache wurde wie schon erwähnt als nachfolger für eine Lehrsprache entwickelt, und somit so das sie leicht zu erlernen ist. Sie wurde so entwickelt das sie einfach und übersichtlich ist. Ein Punkt dieser Übersichtlichkeit ist, bei der Benutzung der Shell, das automatische einrücken von Codeteilen. Die Shell sagt sofort was sie stört und sorgt dafür das der Nutzer einen ordentlichen Stil pflegt. Um Schlagworte zu nutzen steht Python für folgende Programmier Paradigmen:

- prozedurale Programmierung
  - steht für
- objektorientiere Programmierung
  - steht für eine totale objektorientierung der Sprache. Alles ist ein Obejekt selbst die Klassen.
- funktionale Programmierung
  - steht für einen großen Anteil an schon integrierten Funktionen zu allen möglichen Gebieten. Funktionen die zum Beispiel

5.3.1 Typisierung und Plattformunabhängigkeit

Python steht vor allem für Typisierung. Zum einen herrscht in Python eine starke Typisierung, was bedeutet das hier ein Objekt immer genau von einem Typ ist. Ein String ist immer ein String und ein Int ist immer ein Int. Es lässt sich nicht wie in anderen Sprachen umhercasten.

Zum anderen beherrscht Python genauso eine dynamische Typisierung. Das bedeutet das keine Variablen im vorherein deklariert werden müssen. Wenn eine Variable als zum Beispiel als Int gebraucht wird wird die Variable kurzerhand zu einem Integer. Wird die Variable später für einen anderen Datentyp gebraucht so kann sie diesem zugewiesen werden. Erst zur Laufzeit werden die Eigenschaften eines Objektes untersucht.

5.3.2 Plattformunabhängig

Letzten Endes ist Python auch noch Plattformunabhängig - wie Java. Der Code wird in einen Zwischencode umgesetzt und dann erst ausgeführt.

5.4 Beispiele zur Programmierung

Sämtliche Beispiele zur Programmierung in Python werden auf dem Python Interpreter ausgeführt - welcher wie eine Shell funktioniert. Diese Shell trägt den Namen IDLE. Durch die Arbeit auf dem Interpreter wird bei Anfängern [2] dafür

[2] wie auch allen anderen

gesorgt das Einrückungen gnadenlos eingehalten werden. Im weiteren Verlauf dieses Abschnitts werde ich mehrere Beispiele für Programmierung unter Python vorstellen und an diesen auch bestimmte Merkmale der Sprache aufzeigen.

5.4.1 Hello World in Python

Das einfachste Beispiel das zu bringen ist als erstes.

```
1 >>>Hello World
2
3 Hello World
```

Auf dem Interpreter ausgeführt wird diese Eingabe sofort wieder ausgeführt.

5.4.2 Wertzuweisungen

Hier ein Beispiel für eine Wertzuweisung unter Python als Beispiel für die Dynamische Typisierung.

```
1 >>> x = 4          # Anlegen einer Variable, die den
    Integer-
2                    # Wert 4 erhält
3 >>> abc = 'abc'    # Anlegen einer Zeichenkette
4 >>> y = z = 5      # 2 neue Variablen werden eingeführt
     und
5                    # mit dem rechts stehenden Wert befü
                       llt
6 >>> x, y = y, x+y  # Mehrfachzuweisung für die
    Variablen.
7                    # (danach gilt x== 5, y == 9)
8 >>> del x          # x ist jetzt nicht mehr definiert.
```

Weiterhin lässt sich mit dem Befehl Type der Momentane type(objekt) einer Variablen erfragen. Dies kann man benutzen um den Typ zu bestimmen und mit anderen Objekten zu vergleichen. Zuerst eine kleine Liste von verschiedenen Typen in Python.

Als Numerische Datentypen wären das:

- integer
  Entspricht dem Typ Long in C
- long integer
  Ist in der Genauigkeit unbeschränkt. Um long integer von ingeter zu unterscheiden wird an die Zahl ein 'L' oder 'l' angehängt.
- floating point
  In floating point werden Gleitkommazahlen abgespeichert, wobei die Genauigkeit Implementierungs abhängig ist.
- complex
  Um komplexe Zahlen zu modellieren, werden zwei floating points im neuen Typ complex gekapselt. Dabei wird der Imaginärteil immer durch das Suffix 'J' bzw. 'j' gekennzeichnet.

Als nächstes wäre da der Datentyp String.
String ist eine Folge einzelner Zeichen. Den Datentyp Char gibt es in Python nicht, womit jedes Zeichen ein String ist. Eine Zeichenfolge kann in verschiedene Mengen von Ausführungszeichen geklammert werden. In Einfache ('TEXT'), in Doppelte("TEXT")und sogar in Dreifache ("'TEXT"'), wobei die Einfachen und die Doppelten äquivalent sind, während in der 3er Form Strings verwendet werden können die sich auch über mehrere Zeilen fortsetzen.

Hier nun einige Beispiele für die Arbeit mit Strings:

```
1  >>> myString = "Test\n Das sollte Zeile 2 sein 2\t und
       etwas Freiraum"
2  >>> print myString
3  Test
4  Das sollte Zeile 2 sein 2 und etwas Freiraum
5  >>> myString = 'Man sieht:\ndoppelte " müssen nicht
      mit einem \\ maskiert werden,\nwenn der String in
      einfachen \' steht.'
6  >>> print myString
7  Man sieht:
8  doppelte " müssen nicht mit einem \ maskiert werden,
9  wenn der String in einfachen ' steht.
10 >>> myString = """Das ist ein
11 mehrzeiliger String ohne \\n"""
12 >>> print myString
13 Das ist ein
14 mehrzeiliger String ohne \n
15 >>> myString = "Pyt" + "hon"
16 >>> print myString
17 Python
18 >>> print (myString + ' ')*3
19 Python Python Python
```

Nun noch ein kleines Beispiel zum vergleich von 2 Objekten verschiedenen Typs.

```
1  >>> v = 100              #wir übergeben v eine Zahl
2  >>> v
3  100
4  >>> n = "basteln"        #wir übergeben n einen String
5  >>> n
6  'basteln'
7  >>> type(v)              #der Typ von v wird mit type()
       erfragt
8  <type 'int'>             #und ist vom Typ int
9  >>> type(n)
10 <type 'str'>             #n ist vom Typ String
11 >>> type(v)==type(n)     #Bei einem Vergleich der
      beiden Objekte
12 False                    #Sieht man das die beiden
      Typen nicht vom gleichen Typ sind
```

5.4.3 Arbeiten mit Listen

```
>>> l = ["c", "b", "a", 5]
>>> l.sort()              # Standardsortieralgorithmus
>>> l                     # Liste ausgeben
[5, 'a', 'b', 'c']
>>> l.index('b')          # wo ist 'b' jetzt?
2
```

5.4.4 GUI eine schnelle Oberfläche

5.5 Weitere Informationen

An weiteren Informationen lässt sich die Aktuelle Version von Python nennen. Version 2.5.1 vom 18. April 2007

Weiterhin eine kleine Liste von Firmen und Werken bei denen Python mit benutzt wurde.

Eine der größeren Entwicklungen in den kommenden Jahren wird Python 3000 (Py3K), welche nicht mehr kompatiebel zu den früheren Versionen von Python sein wird. Eine alpha Version von Py3K wurde Ende August 2007 veröffentlicht. In diesem sollen Probleme von früheren Python Versionen gelöst werden.

Kapitel 6

Haskell (Andreas Textor)

6.1 Einleitung

Dieses Dokument soll einen Überblick über die grundlegenden Eigenschaften und Fähigkeiten von Haskell bieten. Es ist nicht als Schritt-für-Schritt-Anleitung zu verstehen, um Haskell zu lernen. Haskell ist eine pure, funktionale Programmiersprache, die als Auswertungsstrategie die Bedarfsauswertung (engl. *lazy evaluation*) verwendet. In einer *funktionalen Programmiersprache* werden Funktionen nicht im sonst in der Informatik üblichen Sinne als Unterprogramme, sondern vielmehr als mathematische Funktionen verstanden. Ein Programm wird also ausgewertet wie eine (mathematische) Funktion, und nicht wie eine Reihe von Rechenanweisungen. *Pur* bedeutet, dass es in normalen Funktionen keine Seiteneffekte gibt, das heißt eine Funktion ändert nichts an der „Welt“ (zum Beispiel Ein- und Ausgaben oder Verändern von globalen Variablen). Es kann überprüft werden, ob eine Funktion pur ist, oder nicht, indem wir die Frage stellen: „Liefert diese Funktion für die selben Parameter immer das selbe Resultat?“. Da Ein- und Ausgaben essentiell sind für die meisten Programme, bietet Haskell auch dafür eine Möglichkeit, allerdings sauber getrennt vom Rest des Programmes, das nur unabhängige Berechnungen durchführt. *Bedarfsauswertung* bedeutet, dass ein Ausdruck erst dann ausgewertet wird, wenn er wirklich benötigt wird. Das kann auch bedeuten, dass ein Ausdruck überhaupt nicht ausgewertet wird, wenn er für die Ausführung des Programmes nicht benötigt wird. Durch die Bedarfsauswertung vereinfachen sich einige Algorithmen sehr. Das Gegenteil der Bedarfsauswertung ist die strikte Auswertung (engl. *strict evaluation*), die zum Beispiel in C und Java verwendet wird.
Was diese Eigenschaften für die Programmierung in der Praxis bedeuten und wie sie im Einzelnen funktionieren, soll im Laufe des Textes näher erläutert werden.

6.2 Haskellcompiler und -Interpreter

Doch bevor wir weiter auf die Sprache eingehen, sollen hier zunächst die verfügbaren Haskell-Pakete kurz vorgestellt werden. Haskell-Code wird in der Regel zu nativen ausführbaren Dateien kompiliert, es gibt aber auch Interpreter, die speziell zur Entwicklungszeit hilfreich sind. Es gibt drei größere Haskell-Pakete: Den „Glasgow Haskell Compiler" (GHC), Hugs und nhc98.

GHC Der GHC ist die am weitesten verbreitete Haskellsoftware. Das Paket umfasst den Compiler `ghc` und den Interpreter `ghci` und eine beträchtliche Liste von Spracherweiterungen. Der GHC wird für die Beispiele in diesem Text verwendet.
Sobald das Paket installiert ist, kann mit dem Befehl `ghci` der Interpreter gestartet werden, der mit dem Prompt `Prelude>` auf Benutzereingaben wartet. Hier können einfache Konstrukte zum Testen direkt eingegeben werden, oder mittels `:l Datei.hs` (l steht für load) externe Quelldateien geladen werden.
Bezugsquelle: `http://haskell.org/ghc`

Hugs Hugs liefert keinen Compiler, sondern nur einen Interpreter. Dieser lädt geringfügig schneller als der `ghci` und verfügt über eine eingebaute Grafik-Bibliothek (auf die hier jedoch nicht eingegangen wird).
Bezugsquelle: `http://haskell.org/hugs`

nhc98 Der nhc98 liefert nur einen Compiler, aber keinen Interpreter. Er ist speziell darauf ausgelegt, möglichst kleine ausführbare Dateien zu erzeugen, die auch zur Laufzeit wenig Speicher benötigen. Darüber hinaus läuft der Compiler selbst schneller als der `ghc`, die erzeugten Programme laufen jedoch langsamer.
Bezugsquelle: `http://haskell.org/nhc98`

6.3 Die Grundeigenschaften am Beispiel

Haskell ist *funktional*, *pure* und *lazy*. Was heißt das anschaulich? Wir können uns vorstellen, dass alle Berechnungen, die durchgeführt werden, mathematischen Funktionen entsprechen: $f(x) = \ldots$, für eine bestimmte Eingabe gibt es also eine bestimmte Ausgabe. Prinzipiell arbeiten imperative Sprachen auf den ersten Blick ähnlich, wenn Unterprogramme („Funktionen"') verwendet werden; betrachten wir daher ein (häufig zur Demonstration verwendetes) Beispiel, die Berechnung der Fakultät. Die Fakultät bezeichnet für eine natürliche Zahl n das Produkt der natürlichen Zahlen kleiner gleich n:

$$n! = 1 * 2 * 3 * \cdots * (n-1) * n$$

Die obige Formel hilft für das Verständnis, was genau berechnet werden soll, sie ist jedoch keine beschreibende Definition der Fakultätsfunktion. Eine Funktion ist laut Definition[Pap01] „eine Vorschrift, die jedem Element [. . .] genau ein Element [. . .] zuordnet", was wir benötigen, ist eine eindeutige Zuordnung. Diese kann für die Fakultätsfunktion folgendermaßen beschrieben werden:

$$n! = \begin{cases} 1, & n = 1 \\ n * (n-1)!, & sonst \end{cases}$$

Die Definition kann gelesen werden als: „Falls $n = 1$ ist, so ist $n! = 1$, andernfalls ist $n! = n * (n-1)!$“, also werden hier zwei Zuordnungen[1] getroffen:

$$\begin{array}{rcl} 1 & \mapsto & 1 \\ n \; mit \; n \neq 1 & \mapsto & n * (n-1)! \end{array}$$

Auch wenn diese Definition auf den ersten Blick verwirrender aussieht, als die erste Formel - vor allem, weil hier in der Berechnungsvorschrift der Funktion auf die Funktion selbst zurückgegriffen wird - so bedeutet sie doch das selbe. Wollen wir nun betrachten, wie die Fakultätsfunktion in Haskell formuliert werden kann. Die Funktion kommt ohne Strukturen wie einen Block aus geschweiften Klammern aus, die Zusammengehörigkeit der einzelnen Zeilen ist hier über den Namen der Funktion gegeben.

```
1 fac :: (Num t) => t -> t
2 fac 1 = 1
3 fac n = n * fac (n - 1)
```

Die erste Zeile gibt hier die Signatur[2] der Funktion an: `fac` ist der Name der Funktion, `(Num t) =>` bedeutet: Der im folgenden verwendete Typ `t` muss eine Nummer sein, `t -> t` bedeutet: Für einen Eingabewert vom Typ `t` erzeugt die Funktion einen Ausgabewert vom Typ `t`. In der zweiten Zeile wird der Funktionswert der Funktion für den Eingabewert 1 definiert, nämlich 1, und die dritte Zeile enthält den Funktionswert für alle anderen Eingabewerte in Form eines rekursiven Aufrufes.

6.3.1 Eine „funktionale“ Sprache

So kurz dieses erste Beispiel auch ist, so zeigt es doch einige wichtige Eigenschaften von Haskell. Als erstes sehen wir, wie die Berechnung durchgeführt wird: Im Gegensatz zu einer imperativen Sprache, in der die Berechnung der Fakultät unter Umständen durch eine `for`-Schleife gelöst werden würde (was der Umsetztung der ersten Formel entspräche), bedienen wir uns hier der Rekursion. Dies ist vermutlich der größte und für Benutzer von imperativen Sprachen wie C auch gewöhnungsbedürftigste Unterschied: In Haskell entspricht die Zuweisung eines Wertes an eine Variable der Deklaration einer Konstanten - es ist nicht möglich, im Nachhinein einer Variable einen neuen Wert zuzuweisen. Warum das so ist, läßt sich verstehen, wenn wir auf die folgenden beiden Zuweisungen schauen: $x = 4, x = 5$. In einer Programmiersprache, die ihr Programm als Liste von Rechenanweisungen versteht, könnten diese Zuweisungen hintereinander sinnvoll sein, betrachten wir sie jedoch *mathematisch*, so bedeutet $x = 4, x = 5$ auch $4 = 5$, was einen Widerspruch darstellt. In diesem Sinne wird eine Zuweisung auch in Haskell verstanden. Diese Eigenschaft verhindert allerdings auch Schleifenkonstrukte, da hier eine Zählvariable verändert werden müsste. Der Programmierer ist gezwungen, darüber nachzudenken, was ein Algorithmus eigentlich leisten soll, und weniger, wie dies bewerkstelligt werden soll, was im Allgemeinen zu leichter nachvollziehbarem und kürzerem Programmcode führt.

[1] Das Symbol „$\mapsto$“ wird gelesen als „wird abgebildet auf“ oder als „wird zugeordnet zu“.

[2] Die Signatur gibt an, welchen Namen eine Funktion hat, welchen Rückgabetyp, sowie welche Parameter und ihre Typen. In C können wir die Signatur auch durch den Prototypen einer Funktion ausdrücken.

Wie wir mit dieser scheinbaren Einschränkung trotzdem sinnvoll programmieren können, werden wir später sehen.

6.3.2 Verschiedene Schreibweisen für Funktionen

Die zweite wichtige Eigenschaft, die das Fakultäts-Beispiel zeigt, ist die Art, mit der Funktionen in Haskell notiert werden können. Die Fakultätsfunktion verwendet das sogenannte *Pattern Matching*; wie wir sehen, ist die Funktion für verschiedene Eingabewerte definiert, ohne dass wir eine Verzweigung mit einem `if` benötigen würden. Beim Aufruf der Funktion wird also zunächst überprüft, ob der übergebene Parameter 1 ist (Zeile 2); Der Funktionswert ist dann mit 1 gegeben und die dritte Zeile der Funktion wird überhaupt nicht ausgewertet. Falls der Parameter nicht 1 ist, rutscht die Überprüfung in die nächste Zeile (Zeile 3), wo der Parameter auf das n zutrifft. Das n wird an dieser Stelle als der Parameter der Funktion interpretiert und kann als solcher in der Berechnungsvorschrift für den Funktionswert verwendet werden.
Neben der Notation mit Pattern Matching gibt es noch eine weitere deklarative Möglichkeit, eine Funktion zu formulieren, die sogenannten Guards. Die Guards können wir als eine Erweiterung des Pattern Matching ansehen, die es erlaubt, beliebige boolsche Ausdrücke zur Auswahl der einzelnen Teile einer Funktionsdefinition zu verwenden. Die Fakultätsfunktion in der Guard-Notation würde so aussehen:

```
1 fac2 :: (Num t) => t -> t
2 fac2 n | n == 1 = 1
3        | otherwise = n * fac2(n - 1)
```

Der senkrechte Strich „|“ kann gelesen werden als „unter der Bedingung, dass“. Zeile 2 definiert also die Funktion `fac2` für einen Wert n unter der Bedingung, dass n gleich 1 ist, und der Funktionswert ist 1 (`==` ist wie in C der Vergleichsoperator, `=` ist eine Zuweisung). Zeile 3 defniert für alle anderen Fälle („otherwise“) den Funktionswert[3]. Hier wird der Funktionsname mit Parametern nicht erneut angegeben, der Compiler erkennt anhand der senkrechten Striche - der Guards - dass die einzelnen Zeilen zur selben Definition gehören. Die deklarative Schreibweise mit Pattern Matching oder Guards ist eine von zwei möglichen Schreibweisen, die Haskell erlaubt. Stattdessen ist es auch möglich, die Funktion in Form eines einzelnen Ausdrucks zu schreiben:

```
1 fac3 :: (Num t) => t -> t
2 fac3 n = if n == 1 then 1 else n * fac3(n - 1)
```

In Haskell ist zu jedem `if` das `else` zwingend erforderlich, andernfalls könnte ein Wert undefiniert bleiben.
Ob er die deklarative oder die Ausdruck-Schreibweise verwenden will, ist dem Programmierer freigestellt. Freigestellt ist dem Programmierer auch, ob er Funktionen in der *infix*-Notation aufrufen will oder nicht - ein Beispiel dafür sind Operatoren wie „+“ und „−“, die in der Mitte des Ausdruckes verwendet werden.

[3] `otherwise` ist einfach definiert als `true` und matcht auf alle Ausdrücke, die bis zu dieser Stelle durchgefallen sind.

Wir können statt der Infix-Schreibweise `fac(n - 1)` auch schreiben `fac((-)n 1)`. Was wir hier tun, ist den Minus-Operator (der auch nur eine Funktion mit zwei Argumenten ist) einzuklammern, und die Argumente dahinter zu schreiben. In der Gegenrichtung können wir auch eine Funktion, die normalerweise in Präfix-Notation aufgerufen wird, in die Infix-Notation überführen, indem wir den Funktionsnamen beim Aufruf in rückwärts geneigte Hochkommas einschließen. So wird aus dem Aufruf der Modulo-Funktion in Präfix-Notation `mod 7 2` der Ausdruck in Infix-Notation: ``7 `mod` 2``.

6.3.3 Typklassen

Für die dritte wichtige Haskell-Eigenschaft der gezeigten Fakultätsfunktion müssen wir die erste Programmzeile, die Funktionssignatur, noch einmal näher betrachten. Zunächst sei hier angemerkt, dass eine solche Signatur nicht zwingend erforderlich ist, sie zu einer Funktion zu schreiben gehört aber zum guten Stil und hilft in vielen Fällen, leichter zu verstehen, was eine Funktion leisten soll. Die bemerkenswerte Eigenschaft der Zeile ist, dass `Num` kein Typ ist, sondern eine *Typklasse*. Für den Begriff der Klasse sollten wir hier alles vergessen, was wir aus der objektorientierten Programmierung kennen, gemeint ist hier eine Eigenschaft, die ein Typ haben kann. Ein Typ, der einer Typklasse t angehört ist vergleichbar mit einer Java-Klasse, die ein Interface t implementiert; Interface und Typklasse definieren eine Obermenge von Eigenschaften. Die konkreten Typen `Integer` und `Double` gehören der Typklasse `Num` an, beide lassen sich durch literale Zahlen im Quellcode darstellen. Beide gehören auch der Klasse `Ord` an, die angibt, dass ein Datentyp sortierbar ist[4], sowie der Klasse `Show`, die angibt, dass ein Datentyp in einen String konvertierbar ist (ähnlich der `toString()`-Methode in Java). Es gibt noch einige weitere Typklassen. Wie wir sehen, ist die Fakultätsfunktion nicht nur für Integerzahlen definiert, sondern für alle Datentypen, die der `Num`-Klasse angehören, und wirklich können wir die Funktion sowohl mit einer Integerzahl, als auch einer Doublezahl als Parameter aufrufen (Die Doublezahl wird als solche definiert, weil der Dezimalpunkt verwendet wird, ähnlich wie in C):

```
1 *Main> fac 5
2 120
3 *Main> fac 5.0
4 120.0
5 *Main>
```

Wir sehen hier einen Ausschnitt aus einem Interpreterlauf, bei dem alles hinter dem `>` in einer Zeile vom Benutzer eingegeben wurde, die Ausgaben in der Folgezeile sind die Ergebnisse. Ohne es zu merken, haben wir hier generische Programmierung angewandt. Bei Funktionen, deren Einsatzzweck nicht im Vorfeld bereits in Stein gemeißelt werden muss, kann der Typ so generisch gehalten werden wie möglich; die Wiederverwendbarkeit kann dadurch gesteigert werden. Ein Beispiel hierfür sind Funktionen die Datenmengen sortieren - sobald sich für den Typ der Elemente einer Menge der Vergleichsoperator „<“ anwenden

[4] Num und Ord sind deshalb zwei getrennte Klassen, weil es auch Datentypen geben kann, die zwar sortierbar sind, aber keine Zahl.

lässt (zum Beispiel natürliche Zahlen), ist die Menge sortierbar, egal, um welchen *konkreten* Typ es sich tatsächlich handelt. Benötigen wir stattdessen eine Funktion, die sehr spezifisch auf einen bestimmen Zweck ausgelegt ist, können wir den Typ konkret festlegen. Wissen wir, dass unsere Faktorisierungsfunktion nur mit Ganzzahlen genutzt werden soll, so können wir die Funktionssignatur auch festlegen wie folgt:

```
1 fac :: Int -> Int
```

Das bedeutet, für einen Eingabewert vom Typ `Int` soll ein Ausgabewert vom Typ `Int` erzeugt werden. Dies hilft dem Compiler, weniger komplizierten und unter Umständen schnelleren Code zu erzeugen. Außerdem kann die Einschränkung der Typen helfen, schon zur Compilezeit eine strengere Typprüfung durchführen zu lassen (so dass wir beispielsweise nicht die Fakultätsfunktion mit einem Double-Argument aufrufen, wenn wir eigentlich nur Integers zulassen wollen). Wie bereits erwähnt, ist die Funktionssignatur nicht zwingend erforderlich. Der Haskell-Compiler kann für eine Funktion automatisch den Typ bestimmen. Wir könnten auch die Fakultätsfunktion ohne die Funktionssignatur definieren, und den `ghci` fragen, welchen Typ die Funktion hat, indem wir das Kommando `:t` (wie *type*) verwenden:

```
1 *Main> :t fac
2 fac :: (Num t) => t -> t
```

Bei dem Ergebnis wird „`::`“ so gelesen wie „hat den Typ“. Der Compiler entscheidet sich für den generischsten Typ, der für eine Funktion möglich ist. In unserem Fall entspricht das dem Typ, den wir im ersten Codebeispiel angegeben haben.

6.4 Datentypen und ihre Manipulation

Nach diesem ersten Einblick wollen wir die Möglichkeiten, die uns Haskell bietet, genauer untersuchen.

6.4.1 Typsystem und Grundlegende Datentypen

Mit den Typklassen im vorigen Abschnitt haben wir bereits auf ein fortgeschritteneres Konzept vorgegriffen. Wenn wir das Typsystem von Haskell verstehen wollen, fangen wir dazu noch einmal von unten an: In Haskell gibt es einen statischen Typcheck, das heißt, jeder Ausdruck hat einen ganz bestimmten Typ. Wird eine Funktion, die einen Parameter eines bestimmten Types erwartet, mit dem falschen Typ aufgerufen, so resultiert das in einem Fehler zur Compilezeit, das Programm kompiliert nicht. Typen werden außerdem nicht implizit in andere verwandelt, wie das zum Beispiel in C der Fall ist (In C kann ohne Weiteres eine Funktion, die einen `int`-Parameter erwartet, mit einem `double`-Wert aufgerufen werden). Dieses Verhalten kann viel Ärger sparen, und die Anzahl der Fehler in einem Programm im Vorfeld stark reduzieren. Darüber hinaus verwendet Haskell *Typinferenz*, das bedeutet, der Typ eines Ausdruckes wird anhand seines Kontextes erfasst, er muss nicht explizit durch den Programmierer angegeben werden. Dies ermöglicht überhaupt erst, dass wir im vorigen

Abschnitt den Interpreter mit dem `:t`-Kommando nach dem Typ der Funktion fragen konnten. Dass das auch mit den übrigen Datentypen funktioniert, sehen wir hier:

```
1 *Main> :t 'a'
2 'a' :: Char
3 *Main> :t 1 == 2
4 1 == 2 :: Bool
5 *Main> :t "Hallo"
6 "Hallo" :: [Char]
7 *Main>
```

6.4.2 Listen

Was bedeutet das Ergebnis `[Char]`? Der Ausdruck `"Hallo"` ist vom Typ einer *Liste* von `Char`s (einzelnen Zeichen). Eine Liste kann in Haskell für Elemente jeden Types angelegt werden, die Elemente einer Liste müssen jedoch alle vom selben Typ sein. Die leere Liste wird durch „`[]`" definiert, soll die Liste Werte enthalten, werden diese von Kommas getrennt in die eckigen Klammern geschrieben: `[1,2,3]`. Für Zahlen ist es außerdem möglich, einen Zahlenbereich direkt als Liste zu definieren: `[1..5]` entspricht `[1,2,3,4,5]`. Gibt man die ersten beiden Elemente dieser Zahlenreihe an, so wird der Abstand zwischen diesen beiden als der Abstand aller Elemente der Liste angenommen: `[1,3..9]` entspricht `[1,3,5,7,9]`. Der cons-Operator[5] „`:`" ermöglicht es, Elemente am Anfang einer Liste zuzufügen, statt `"Hallo"` können wir also auch schreiben: `'H':'a':'l':'l':'o':[]`. Tatsächlich ist die Schreibweise mit doppelten Anführungszeichen sogenannter „syntaktischer Zucker", also eine (für die Semantik des Programmes irrelevante) Vereinfachung in der Schreibweise. Das abschließende `:[]` ist notwendig, weil der `:`-Operator wirklich nur zu Listen zufügen kann, nicht zu einzelnen Elementen. Listen können mittels `++` aneinandergehängt werden, das heißt `[1, 2] ++ [3, 4]` ergibt die Liste `[1, 2, 3, 4]`; das Element einer Liste am Index n erhalten wir mit dem Index-Operator „`!!`": `[2,4,6] !! 1` liefert den Wert 4. Es gibt einige Listenfunktionen: `head` liefert das erste Element einer Liste, `tail` liefert alle Elemente einer Liste außer dem ersten als neue Liste und `length` liefert die Länge einer Liste. Mit `last` kann das letzte Element einer Liste geliefert werdern, `take` liefert die ersten n Elemente der Liste als neue Liste und `drop` liefert die Liste außer den letzten n Elementen.

Spezielle Listenfunktionen

Interessanter sind die Funktionen `map`, `filter` und `foldr`. Die Abbildungsfunktion `map` bekommt als Argumente eine Funktion und eine Liste, und wendet auf jedes Element der Liste die Funktion an. Ein Aufruf könnte so aussehen:

```
1 *Main> map Char.toUpper "Hallo"
2 "HALLO"
```

Als erstes Argument übergeben wir `map` die Funktion `toUpper`, die einen Kleinbuchstaben in den zugehörigen Großbuchstaben verwandelt. Diese Funktion hat

[5] „cons" von *cons*truction

den Typ `Char -> Char`, denn sie nimmt ein `Char` und liefert dafür auch ein `Char` zurück. Als zweites Argument übergeben wir die Char-Liste `"Hallo"`. Für jedes Element der Liste wird nun separat die Funktion `toUpper` aufgerufen, und das jeweilige Ergebnis der Ergebnisliste angehängt. Das Resultat ist der String[6] `"HALLO"`. `map` funktioniert nicht nur mit `Char`-Listen, sondern mit allen Typen. Die einzige Voraussetzung ist, dass der Typ des Eingabeparameters der übergebenen Funktion der selbe ist, den die Elemente der Liste haben (andernfalls könnten wir die übergebene Funktion nicht auf die Elemente der Liste anwenden).
Die Funktion `filter` kann verwendet werden, um aus einer Liste von Werten nur bestimmte Werte, die ein Kriterium erfüllen, herauszufiltern. `filter` bekommt genauso wie `map` als Argumente eine Funktion und eine Liste, die übergebene Funktion muss aber als Rückgabetyp `Bool` haben. Es wird dann für jedes Element aus der Liste überprüft, ob das Kriterium zutrifft (das heißt die übergebene Funktion liefert `true` für das Element):

```
1 *Main> filter Char.isLower "Hallo Welt"
2 "alloelt"
```

`isLower` hat den Typ `Char -> Bool`, sie bekommt also einen Parameter vom Typ `Char` und liefert genau dann `true`, wenn das Zeichen ein Kleinbuchstabe ist. Auch `filter` kann natürlich für beliebige Datentypen verwendet werden, nicht nur für Characters. Wie bei `map` muss der Typ des Eingabewertes der übergebenen Funktion dem Typ der Listenelemente entsprechen. Die dritte der komplexeren Funktionen, `foldr`, ist ein wenig komplizierter. Die Funktion erhält drei Argumente: Eine Funktion, einen Startwert und eine Liste. Anschaulich können wir uns vorstellen, dass `foldr` bei einer Listenkonstruktion überall den „`:`“-Operator durch die übergebene Funktion, und die leere Liste durch den Startwert ersetzt; für eine Liste `[1,2,3,4]` (geschrieben als Konstruktion: `1:2:3:4:[]`) erhalten wir also bei einem Aufruf von `foldr (+) 0 [1,2,3,4]` anschaulich den Wert $1 + 2 + 3 + 4 + 0$, der tatsächliche Rückgabewert des Aufrufes ist 10. Nun stellt sich die Frage, was passiert, wenn man als Übergabefunktion eine Funktion nutzt, die nicht assoziativ[7] ist, also es nicht egal ist, ob man die entstehende Berechnung von links oder von rechts durchführt. `foldr` geht rechts-assoziativ vor (daher auch das „r“ im Namen der Funktion), die Klammerung würde also so aussehen: $1 + (2 + (3 + (4 + 0)))$. Überprüfen können wir dies, indem wir eine nicht-assoziative Funktion zum Testen benutzen, Minus statt Plus: `foldr (-) 0 [1,2,3,4]` ergibt -2. Wird in der Liste der cons-Operator durch ein Minus ersetzt und die leere Liste durch den Startwert 0, und beachten wir die Rechtsassoziativität von `foldr`, so ergibt sich: $1 - (2 - (3 - (4 - 0))) = 1 - (2 - (-1)) = 1 - 3 = -2$.
Als Gegenstück zu `foldr` gibt es auch `foldl`, das seine Berechnungen linksassoziativ durchführt. Für `foldl (-) 0 [1,2,3,4]` erhalten wir als Ergebnis -10, denn: $(((0 - 1) - 2) - 3) - 4 = ((-3) - 3) - 4 = -6 - 4 = -10$.

[6] Statt `[Char]` kann in Haskell auch direkt `String` geschrieben werden, die Definitionen sind äquivalent.

[7] Eine Funktion $(\circ)$ ist assoziativ, wenn gilt $(a \circ b) \circ c = a \circ (b \circ c)$.

Pattern Matching mit Listen

Pattern Matching lässt sich auch auf zusammengesetzte Ausdrücke anwenden, besonders hilfreich ist dies bei der Verarbeitung von Listen. So können wir beispielsweise die Funktion `length`, die die Länge einer Liste bestimmt, selbst so schreiben:

```
length [] = 0
length (x:xs) = 1 + length xs
```

In der ersten Zeile legen wir die Länge für eine leere Liste mit 0 fest. In der zweiten Zeile definieren wir die Länge einer Liste als 1 für das erste Element plus die Länge der übrigen Elemente. Mit dieser Notation können wir bei der als Parameter übergebenen Liste direkt auf das erste Element als `x`, und auf die danach folgende Liste von übrigen Werten als `xs` zugreifen.

List Comprehensions

Haskell bietet die Möglichkeit, Listen zu definieren, die der mathematischen Definition einer Menge von Elementen aus einer Übermenge und beliebig vielen Kriterien entspricht. Wenn wir eine Menge definieren: $\{f(x)|x \in M \wedge p(x)\}$, in der alle Funktionswerte einer Funktion f enthalten sind, die auf alle Elemente der Menge M angewendet wird, die eine Eigenschaft p haben; dann können wir diese Definition auch in Haskell ähnlich notieren. Definieren wir zum Beispiel die Menge der quadrierten Werte der ungeraden Zahlen zwischen 1 und 10, würden wir das mathematisch so tun: $\{x^2|x \in \mathbb{N}, 1 \leq x \leq 10 \wedge x \textit{ ist ungerade}\}$. Die Entsprechung in Haskell können wir so schreiben:

```
*Main> [x * x | x <- [1..10], x `mod` 2 == 1]
[1,9,25,49,81]
```

Der `mod`-Operator wird verwendet, um die ungeraden Zahlen herauszubekommen (denn eine ungerade Zahl *modulo* 2 ist immer 1), und diese Bedingung wird auf die Liste der Zahlen von 1 bis 10 angewendet. Auf jedes Element der resultierenden Liste wenden wir die Funktion `x * x` an, was letztendlich die in Zeile 2 ausgegebene Liste als Ergebnis hat. Diese Technik wird in Haskell eine *List Comprehension* genannt. Die Schreibweise ist ebenfalls nur syntaktischer Zucker, die so geschriebene Liste wird auf Funktionen `map` und `filter` abgebildet. Ohne anonyme Funktionen (siehe Abschnitt „Funktionale Programmierung") ist es ebenso gut möglich, `quadriere` und `ungerade` als Hilfsfunktionen zu definieren, und dann die obige Listendefinition von Hand zu, das heißt unter der Bentzung von `map` und `filter`, zu schreiben:

```
*Main> let quadriere x = x * x
*Main> let ungerade x = x `mod` 2 == 1
*Main> map quadriere (filter ungerade [1..10])
[1,9,25,49,81]
```

6.4.3 Tupel

Eine zweite Möglichkeit, mehrere Werte zusammenzufassen, bieten die *Tupel*: Hierbei werden mehrere Werte, auch unterschiedlichen Types, zusammengruppiert. Die Notation für Tupel in Haskell ist ähnlich der der Listen, es werden aber runde statt eckigen Klammern verwendet. Auch für ein Tupel können wir den Interpreter nach seinem Typ fragen:

```
*Main> :t ("Hallo", 'a')
("Hallo", 'a') :: ([Char], Char)
```

Wie wir sehen, können auch Listen Elemente eines Tupels sein. Umgekehrt können wir auch eine Liste von Tupel vereinbaren:

```
*Main> :t [("x", 'y'), ("a", 'b')]
[("x", 'y'), ("a", 'b')] :: [([Char], Char)]
```

Dabei müssen wir wiederum nur sicherstellen, dass alle Tupel innerhalb der Liste die selben Typen enthalten, weil die Kombination aus diesen den Typ des Tupels selbst ausmacht, und die Liste nur Elemente gleichen Types enthalten darf. Eine Liste kann nur gleichartige Tupel enthalten, `[('a', 1), (1, 'a')]` wäre also unzulässig. Die Tupelfunktionen `fst` („first“) und `snd` („second“) sind nur für Tupel mit genau zwei Elementen (Wert-Paare) definiert und liefern den ersten beziehungsweise den zweiten Wert des Tupels. Anders als bei Listen kann ein Tupel nicht programmatisch zusammengesetzt werden (das heißt, es gibt keinen cons-Operator für Tupel). Tupel können zum Beispiel dazu verwendet werden, eine Funktion zu schreiben, die logisch mehr als einen Rückgabewert hat. Tatsächlich hat eine Funktion genau einen Rückgabewert, dieser kann aber ein Tupel aus mehreren verschiedenen anderen Werten sein.

6.4.4 Selbstdefinierte Datentypen

Einfache selbstdefinierte Datentypen

Ein mächtiges Konzept in Haskell sind selbstdefinierte Datentypen. Diese unterscheiden sich von Strukturen aus C oder Java, deshalb wollen wir zunächst ein Beispiel betrachten:

```
data Farbe = Rot
           | Gelb
           | Blau
           | Eigene Int Int Int
```

Hier wird mittels des Schlüsselwortes `data` ein neuer Datentyp namens „Farbe“ definiert. Dem Gleichheitszeichen folgen, durch den senkrechten Balken getrennt, mehrere sogenannte Konstruktoren. Die ersten drei; Rot, Gelb und Blau, haben keine Parameter. Wir können uns vorstellen, dass diese Werte in etwa den Elementen einer Enumeration aus Java entsprechen. Für die Farbe „Eigene“ sind drei Int-Parameter vorgesehen (für Rot-, Grün- und Blau-Anteil der Farbe), die beim Erzeugen eines Wertes vom Farbe-Typ mit angegeben werden. Wenn nun eine Variable vom Typ Farbe definiert wird, kann diese die Werte Rot, Gelb, Blau oder Eigene (mit drei Int-Parametern) enthalten.

```
*Main> :t Rot
Rot :: Farbe
*Main> :t Eigene
Eigene :: Int -> Int -> Int -> Farbe
*Main> :t Eigene 71 150 145
Eigene 71 150 145 :: Farbe
```

Wir sehen, dass „Rot“ vom Typ Farbe ist. Für „Eigene“ ohne Parameter sieht die Typsignatur so aus, wie für eine Funktion, die drei Int-Parameter erwartet, und Farbe als Rückgabetyp hat. Sobald wir drei solche Werte als Argumente übergeben, ist der Typ des resultierenden Ausdruckes wieder eine Farbe. Wie die Verwendung dieses Datentyps in der Praxis aussieht, soll die folgende Funktion demonstrieren: Sie wandelt einen „Farbe“-Wert um in ein Dreiertupel, das die jeweiligen RGB-Werte enthält.

```
farbeNachRGB :: Farbe -> (Int, Int, Int)
farbeNachRGB Rot = (255, 0, 0)
farbeNachRGB Gelb = (255, 255, 0)
farbeNachRGB Blau = (0, 0, 255)
farbeNachRGB (Eigene r g b) = (r, g, b)
```

Die Signatur zeigt an, dass die Funktion `farbeNachRGB` einen Eingabewert vom Typ Farbe in einen Ausgabewert eines Dreiertupels mit drei Int-Komponenten verwandelt. Wir verwenden für die Funktion hier wieder Pattern Matching: Für Rot, Gelb und Blau wird der Rückgabewert direkt als Tupel definiert, für Eigene matchen wir dessen drei Parameter direkt auf die Bezeichner r, g, b. In der Zeile 5 muss Eigene r g b geklammert werden, damit klar ist, dass die drei Parameter zu dem Konstruktor Eigene gehören, und keine separaten Parameter der Funtion `farbeNachRGB` sind. An dieser Stelle können wir einen Einschub machen, und eine spezielle Möglichkeit des Pattern Matching zeigen: Angenommen, wir möchten nun eine Funktion schreiben, die uns sagt, ob ein Wert vom Typ Farbe eine „Eigene“ Farbe ist, können wir das folgendermaßen tun:

```
istEigene :: Farbe -> Bool
istEigene (Eigene _ _ _) = True
istEigene _ = False
```

Der Unterstrich „`_`“ ist ein Wildcard, der auf beliebige Werte matcht. Da wir die als Argumente der Funktion übergebenen Werte in der Berechnungsvorschrift überhaupt nicht verwenden, brauchen wir ihnen beim Pattern Matching auch keine Namen zu geben. Die Signatur der Funktion stellt sicher, dass das Argument vom Typ `Farbe` sein muss.

Rekursive selbstdefinierte Datentypen

Datentypen können auch rekursiv definiert sein. Als zweites Beispiel soll hierfür die Datenstruktur für einen Binärbaum definiert werden. Ein Binärbaum ist eine Baumstruktur, bei der jeder Knoten 0 oder 2 Kindknoten hat. Hat ein Knoten keine Kinder, so wird er Ast genannt. Da ein Binärbaum auch Werte in den Knoten enthalten soll, wir aber dafür den Typ nicht im Vorfeld festlegen

wollen, werden wir die Definition der Datenstruktur für einen generischen Typ `a` vornehmen:

```
1 data BinaerBaum a = Blatt a
2                   | Ast (BinaerBaum a) a (BinaerBaum a
                      )
```

Das `a` auf der linken Seite des Gleichheitszeichen legt fest, dass alle Konstruktoren der Datenstruktur sich auf den selben Typ a beziehen. Damit stellen wir sicher, dass alle Werte innerhalb des Baumes den selben Typ haben. Das Blatt hat genau einen Parameter vom Typ a, der Ast hat außerdem auch einen linken und rechten Kindknoten, jeweils vom Typ eines Binärbaumes mit Typ `a`. Auch hier soll die Verwendung der Struktur demonstriert werden, wir verwenden dazu die folgende Funktion, die die Größe eines Binärbaumes (die Anzahl der Knoten) berechnet.

```
1 baumGroesse (Blatt x) = 1
2 baumGroesse (Ast links x rechts) = 1 + baumGroesse
     links
3    + baumGroesse rechts
```

Für ein Blatt ist die Größe 1, für einen Ast 1 (für den Knoten selbst) plus die Größe seine beiden Kindknoten.

Der `Maybe`-Typ

Nachdem wir nun gesehen haben, wie Datentypen definiert und verwendet werden, können wir uns dem Problem zuwenden, das auftritt, wenn eine Funktion manchmal keinen Rückgabewert haben soll. Es könnte beispielsweise eine Funktion `findElement` geben, die eine Liste anhand einer bestimmten Eigenschaft, die ein Element haben kann, durchsucht, und das erste Element, das die Eigenschaft erfüllt zurückliefert. Was passiert, wenn kein Element der Liste die gewünschte Eigenschaft erfüllt? In Java würde in so einer Situation meistens einfach `null` zurückgegeben werden - was aber auch eine häufige Fehlerquelle darstellt. Sobald auf eine Methode des vermeintlichen Objektes zugegriffen wird, ist das Resultat eine `NullPointerException`. Die Lösung für dieses Problem in Haskell besteht aus dem `Maybe`-Typ:

```
1 data Maybe a = Nothing
2              | Just a
```

Wir definieren hiermit einen Typ[8], der „optional leer“ sein kann. Die `findElement` -Funktion würde also nicht den Typ `a` als Rückgabetyp haben, sondern den Typ `Maybe a`. Die Funktion könnte folgendermaßen definiert werden:

```
1 findElement :: (a -> Bool) -> [a] -> Maybe a
2 findElement e [] = Nothing
3 findElement e (x:xs) | e x = Just x
4 findElement e (x:xs) | otherwise = findElement e xs
```

[8]Tatsächlich ist der `Maybe`-Typ bereits in der Startbibliothek von Haskell definiert, weil er so häufig benötigt wird.

Die Signatur sieht etwas verwirrender aus, als die bisher gezeigten, daher soll sie noch einmal erklärt werden. Der erste Parameter `(a -> Bool)` muss eine Funktion sein, die einen Wert vom Typ `a` als Parameter bekommt, und einen `Bool`-Wert zurückliefert; der zweite Paramter ist eine Liste von Werten des Types `a` und der Rückgabetyp der Funktion ist ein Wert vom Typ `Maybe a`. Wird `findElement` nun außer der Eigenschaftsfunktion eine leere Liste übergeben (Pattern Matching trifft auf Zeile 2 zu), so ist das Ergebnis nur `Nothing`. Da `Nothing` ein Konstruktor des `Maybe`-Types ist, können wir diesen Wert an dieser Stelle zurückliefern. Zeile 3 trifft dann zu, wenn die Liste nicht leer ist, und die Eigenschaftsfunktion `e` mit dem ersten Element der Liste als Parameter `true` ergibt, das heißt wir haben das erste Element der Liste gefunden, das die gesuchte Eigenschaft besitzt. Das Ergebnis von `findElement` ist dann genau dieses Element, was wir als `Just x` als Rückgabewert liefern. Falls das erste Element der Liste nicht die gesuchte Eigenschaft besitzt, überprüfen wir mit Zeile 4 den Rest der Liste, `xs`, mit einem rekursiven Aufruf. Der Vorteil des `Maybe`-Types gegenüber der Rückgabe von `null` wie in Java liegt auf der Hand: Während `Just a` einen tatsächlichen Rückgabewert darstellt und `Nothing` für „keinen Rückgabewert“ steht, wird explizit zwischen den beiden Ergebnissen unterschieden. Soll nun das Ergebnis der Funktion `findElement` an eine weitere Funktion übergeben werden, so wird dort das Pattern Matching nur explizit für einen der Konstruktoren (das heißt `Nothing` oder `Just`) angewandt, nicht für den `Maybe`-Typ selbst. Es kann also nicht vorkommen, dass man versucht auf einem „nicht vorhandenen“ Wert zu operieren. Wird der `Maybe`-Typ verwendet, so kann schon anhand der Signatur einer Funktion abgelesen werden, dass die Funktion in manchen Fällen keinen (das heißt `Nothing` als) Funktionswert liefert.

6.5 Funktionale Programmierung

6.5.1 Verzögerte Auswertung

Obwohl *Laziness*, deutsch: verzögerte Auswertung, kein Feature jeder funktionalen Programmiersprache ist, so ist sie doch ein Feature von Haskell. Es findet keine Auswertung statt, bevor das Ergebnis der Auswertung nicht wirklich benötigt wird. Schauen wir dazu auf folgendes Beispiel: In einer imperativen Sprache wollen wir eine unendliche Liste erzeugen, die an jeder Stelle den Wert 1 hat. Dieses Beispiel verwendet der Einfachheit halber Pseudocode: [DI]

```
1 List makeList() {
2   List current = new List();
3   current.value = 1;
4   current.next = makeList();
5   return current;
6 }
```

Wir sehen, wie die Funktion arbeiten soll, aber zum `return` wird die Funktion nie kommen: Der Aufruf in der vierten Zeile erzeugt eine Endlosrekursion. In Haskell könnten wir die Funktion so schreiben:

```
1 makeList = 1 : makeList
```

Der deklarative Ansatz und die verzögerte Auswerung von Haskell ermöglichen die Funktion leicht. Berechnet wird der Rückgabewert der Funktion erst, wenn darauf zugegriffen wird:

```
Prelude> take 10 makeList
[1,1,1,1,1,1,1,1,1,1]
```

Wenn wir die ersten 10 Elemente der Liste verarbeiten, ist es nicht wichtig, dass die Liste per Definition unendlich ist. Wollen wir die Menge aller natürlichen Zahlen als Liste in Haskell definieren, so können wir dies auch tun, indem wir schreiben: `[1..]`. Laziness macht es möglich, dass in einem Programm Erzeugung und Selektion von Datenmengen getrennt definiert werden können. Beispielsweise könnte ein Schach-Algorithmus zuerst alle möglichen Züge sammeln als umfangreiche Baumstruktur, und sie in einem zweiten Schritt auswerten. Die Äste des Baumes werden erst dann wirklich berechnet, wenn die Traversierungsfunktion den entsprechenden Weg tatsächlich einschlägt.

6.5.2 Lambda-Kalkül

Um zu verstehen, wie Funktionen in Haskell wirklich funktionieren und was funktionale Programmierung eigentlich bedeutet, müssen wir einen kleinen Abstecher in die theoretische Informatik machen. Dazu soll hier zunächst der sogenannte *Lambda-Kalkül* besprochen werden. Dabei handelt es sich um ein System zur Repräsentation von Funktionen und ihren Parametern, sowie der Regelung zum Auswerten dieser Parameter. Wir wollen das anhand eines Beispieles leichter verständlich machen. Die Funktion zur Berechnung einer Quadratzahl $f(x) = x * x$ können wir im Lambda-Kalkül wie folgt notieren: $\lambda x.x * x$. Dabei bedeutet $\lambda x.$, dass wir eine Funktion definieren, die einen Parameter namens x erhält, das darauf folgende $x * x$ ist die zugehörige Rechenvorschrift. Die dazugehörige Funktionsanwendung (wir setzen einen konkreten Wert für x ein) $f(2)$ wird im Lambda-Kalkül so geschrieben: $(\lambda x.x * x)\ 2$. Die Zwei wird für jedes Vorkommen des Parameters x eingesetzt und das Lambda wird entfernt. Das Resultat ist $2 * 2$. Ein solches Lambda kann nur einen Parameter haben, wenn wir also eine Funktion mit zwei Parametern $f(x, y) = 2 * x + y$ schreiben wollen, so notieren wir dies als $\lambda x \lambda y.2 * x + y$. Bei der Funktionsanwendung $(\lambda x \lambda y.2 * x + y)\ 2\ 3$ wandeln wir diese entstandene einzelne Funktion mit mehreren Parametern in mehrere Funktionen mit jeweils einem Parameter um, wir reduzieren das Problem also auf den vorher bereits besprochenen Fall. Diese Umwandlung wird *Currying* genannt, benannt nach dem amerikanischen Mathematiker Haskell Brooks Curry - nach dem auch die Programmiersprache Haskell benannt ist. Um den Ausdruck auszuwerten, entfernen wir das äußerste Lambda und setzen den am weitesten links stehenden Parameter-Wert überall da ein, wo im Ausdruck der Parameter des Lambdas steht:

$$\begin{aligned} & (\lambda x \lambda y.2 * x + y)\ 2\ 3 \\ \Rightarrow\ & (\lambda y.2 * 2 + y)\ 3 \\ \Rightarrow\ & (2 * 2 + 3) \end{aligned}$$

Haskell macht starken Gebrauch vom Lambda-Kalkül, und es ist auch möglich, direkt solche Ausdrücke in Haskell zu schreiben. Dazu formen wir lediglich die

Syntax ein wenig um: Aus dem Lambda „λ“ wird ein Backslash „\“, den Punkt „.“ verwandeln wir in einen Pfeil „->“. Außerdem brauchen wir, wenn der Ausdruck mehr als einen Parameter hat, nur den vordersten Backslash hinzuschreiben. Aus $\lambda x \lambda y.2 * x + y$ wird also `\x y -> 2 * x + y`. Wir können diesen Ausdruck als anonyme (unbenannte) Funktion direkt verwenden:

```
1 *Main> (\x y -> 2 * x + y) 2 3
2 7
```

6.5.3 Funktionen höherer Ordnung

Das erlaubt uns, ohne benannte separate Funktionen zu definieren, einen solchen Ausdruck als Parameter für einen Funktionsaufruf zu verwenden. Wir erinnern uns, dass die Funktion `map` eine Abbildungsfunktion und eine Liste nimmt, und dann die Funktion auf jedes Element der Liste anwendet. Nun können wir beispielsweise die `map`-Funktion folgendermaßen aufrufen:

```
1 *Main> map (\x -> x*x) [1..5]
2 [1,4,9,16,25]
```

Für jeden Wert von 1 bis 5 wenden wir unsere anonyme Lambda-Funktion an, die den Wert quadriert. Das Ergebnis ist die in Zeile 2 ausgegebene Liste. Genauso können wir die im Abschnitt „Listen“ vorgestellte Mengendefinition `[ x * x | x <- [1..10], x ‘mod‘ 2 == 1]` jetzt auch mittels `map` und `filter` konstruieren, ohne dazu die Hilfsfunktionen zu benötigen:

```
1 *Main> map (\x -> x * x) (filter (\x -> x ‘mod
    ‘ 2 == 1) [1..10])
2 [1,9,25,49,81]
```

`map` und `filter` sind sogenannte *Funktionen höherer Ordnung*, denn sie erhalten als einen Parameter eine weitere Funktion. Jede Funktion, die eine andere Funktion als Parameter erhält, oder deren Rückgabewert eine weitere Funktion ist, wird als eine Funktion höherer Ordnung bezeichnet. Sobald eine Programmiersprache Funktionen höherer Ordnung unterstützt, spricht man von einer *funktionalen Programmiersprache*. Eine Funktion als Rückgabewert einer Funktion klingt seltsam, macht aber durchaus Sinn, nachdem wir den Lambda-Kalkül gesehen haben. Sobald wir eine Funktion mit mindestens einem Parameter haben, können wir eine neue Funktion basierend auf der ersten definieren, die weniger Parameter benötigt, weil wir ihr schon einen oder mehrere Werte mitgeben:

```
1 *Main> (+) 1 2
2 3
3 *Main> let addiereEins = (+) 1
4 *Main> addiereEins 2
5 3
```

In Zeile 1 rufen wir zur Erinnerung die Additionsfunktion in Präfix-Notation mit den Argumenten 1 und 2 auf. Wir verwenden in Zeile 2 das Schlüsselwort `let` um im Interpreter die neue Funktion `addiereEins` zu definieren als die

Additionsfunktion, die bereits den Wert 1 als ersten Parameter fest verdrahtet hat. Diese neue Funktion erwartet dann nur noch einen Parameter. Wird die Funktion mit einem Wert als Parameter aufgerufen, so wird dieser Wert der zweite Parameter der Additionsfunktion, einer Funktion höherer Ordnung. Was wir hier tun, nennen wir „teilweise Funktionsanwendung“. Wie flexibel diese Eigenschaft die Programmierung macht, und wie wir sie ausnutzen können, soll das folgende Beispiel zeigen.

Beispiel zu Funktionen höherer Ordnung

Im Abschnitt über die Grundeigenschaften von Haskell stießen wir auf die Tatsache, dass es keine Schleifen in Haskell gibt, weil Zählschleifen nicht möglich sind, da Werte unveränderlich sind. Mit Hilfe von Funktionen höherer Ordnung wollen wir einen Schritt weiter gehen, als in der Sprache eingebaute Schleifenkonstrukte, und uns eine `for`-Schleife selbst als Funktion definieren. Die `for`-Funktion aus C hat die Syntax: `for(<Initialisierung>; <Schleifenbedingung>; < Inkrementierung>)`, also könnte eine `for`-Funktion in Haskell so aufgerufen werden: `for 1 (<=5) (+1) print`. Dabei soll 1 der Startwert sein, `(<=5)` die Schleifenbedingung, `(+1)` die Inkrementierungsfunktion und `print` die Funktion, die auf den jeweiligen Zählstand angewendet werden soll, also die Funktion, die den Schleifenkörper darstellt. Jede der drei übergebenen Funktionen (Bedingung, Inkrement und Schleifenkörper) soll jeweils einen `Int`-Parameter haben, für den beim jeweiligen Aufruf der aktuelle Zählerstand übergeben wird. Dieser konrete Aufruf soll die Zahlen von 1 bis 5 ausgeben, die `for`-Funktion soll aber so generisch sein, dass sie mit beliebigen Startwerten, Schleifenbedingungen und Inkrementierungs- und Schleifenkörperfunktionen funktioniert. Die `for`-Funktion hat an sich keinen logischen Rückgabewert, deswegen wählen wir hierfür den sogenannten „Unit“-Typ `()`, der in etwa dem `void` aus C entspricht (wollten wir ihn selbst definieren, so würde die Definition so aussehen: `data () = ()`).

```
1 for start cond inc job | cond start = do
2     job start
3     for (inc start) cond inc job
4 for start cond inc job | otherwise = return ()
```

Mit dem Guard in der ersten Zeile stellen wir sicher, dass für unsere kopfgesteuerte Schleife die Schleifenbedingung erfüllt ist, indem wir die Bedingungsfunktion `cond` mit dem Startwert als Parameter aufrufen. Beim ersten Durchlauf wird an dieser Stelle also der Ausdruck `(<=5) 1` ausgewertet, was `1 <= 5` entspricht und `True` wird. Die im folgenden verwendete `do`-Notation erlaubt uns, die folgenden Befehle in sequentieller Abarbeitung auszuführen. Zeile 2 und 3 sind eingerückt und zeigen so dem Compiler, dass sie zu dieser `do`-Sequenz gehören. In Zeile 2 führen wir lediglich die Schleifenkörper-Funktion aus, also den eigentlichen Arbeiter der Schleife, und in Zeile 3 starten wir den nächsten rekursiven Aufruf. Dabei wird direkt die Inkrementierungsfunktion mit dem Startwert als Argument aufgerufen. Die letzte Zeile definiert den Schleifenabbruch, sobald die Schleifenbedingung nicht mehr erfüllt ist. Das Ergebnis funktioniert wie erwartet:

```
*Main> for 1 (<=5) (+1) print
1
2
3
4
5
```

6.5.4 Funktionskomposition

Was wäre, wenn wir nun mehr als eine Sache innerhalb der `for`-Schleife mit dem Zählwert machen wollten? Wir könnten mehr als einen `job`-Parameter übergeben, oder eine Liste solcher Parameter, und diese im Schleifenkörper abarbeiten. Das wäre aber sehr umständlich, und auch nicht sonderlich praktisch. Was wir eigentlich wollen, ist eine Funktion übergeben, die mehrere Sachen macht. Mit der Funktionskomposition können wir genau das tun. In der Mathematik gibt es hierfür die Notation $f \circ g$, gelesen als „f folgt g". Das bedeutet nur $(f \circ g)(x) = f(g(x))$, es wird also zuerst g angewendet auf x, und auf dieses Ergebnis dann die Funktion f. In Haskell schreiben wir statt des Kreises $\circ$ einen Punkt `.`.

```
*Main> for 1 (<=5) (+1) (putStr . show)
12345
```

Wir verwenden jetzt nicht mehr die Funktion `print`, die als Parameter einen beliebigen Wert der `Show`-Klasse bekommt (siehe Abschnitt zu Typklassen), sondern die Funktion `putStr`. Da `putStr` als Eingabe nur Strings erlaubt, unser Zähler aber eine Zahl ist, müssen wir den Zähler erst mit Hilfe der `show`-Methode in einen String konvertieren. Beides verbinden wir durch die Funktionskomposition, und übergeben das Resultat als neue Schleifenkörperfunktion. Bei jedem Schleifendurchlauf wird nun zuerst `show` auf den Zähler angewendet, was daraus einen String macht, und dieser String wird dann von `putStr` ausgegeben.

6.6 Monaden

6.6.1 Das Problem mit Seiteneffekten

Bis jetzt haben wir Funktionen betrachtet, die andere Funktionen aufrufen und in ihrer Welt der Berechnungen leben. Auf Ein- und Ausgaben sind wir bis jetzt nicht wirklich eingegangen, abgesehen von dem Beispiel im letzten Abschnitt. Nicht einmal ein einfaches „Hallo Welt"-Programm wurde bis jetzt vorgestellt. Wo liegt das Problem? Haskell ist, wie in der Einleitung erwähnt, eine *pure* Sprache. Funktionen haben keine Seiteneffekte, denn sie berechnen wirklich nur ihr Ergebnis. Diese Eigenschaft hat enorme Vorteile für die Wiederverwendbarkeit und die Strukturierung von Programmen: Wenn wir beispielsweise in C oder Java eine Funktion bzw. Methode x schreiben, und das Programm, das diese Funktion verwendet, funktioniert, so kann es aber trotzdem passieren, dass wir in dem Programm einen neuen Funktionsaufruf auf eine Funktion y vor dem Aufruf von x einfügen, und plötzlich schlägt x fehl. Was passiert ist: y kann Werte verändern und in einen ungültigen Zustand überführen, auf denen x arbeitet.

In Haskell ist das nicht möglich: Jede Funktion kann nur auf ihren Parametern arbeiten und einen Rückgabewert liefern. Es ist implizit sichergestellt, dass zwei korrekt arbeitende Funktionen sich nicht „ins Gehege“ kommen können, weil sie unabhängig voneinander arbeiten. Das Problem daran ist nun, dass wir aber auch Seiteneffekte *benötigen* - wir wollen Text ein- und ausgeben, Dateien lesen und schreiben und Zustände speichern können. Dafür gibt es in Haskell das aus der Mathematik, genauer aus der Kategorientheorie, entlehnte Konzept der *Monaden*. Eine Monade ist ein Konstrukt zur Strukturierung von Berechnungen in Form von Werten und Sequenzen von Berechnungen, die diese Werte verwenden. Mit Hilfe von Monaden können wir Berechnungen aus sequenziellen Bausteinen aufbauen, die wiederum aus sequenziellen Berechnungen bestehen. Die Monaden bestimmen, wie aus zwei bestimmten aufeinanderfolgenden Berechnungen eine neue, größere Berechnung aufgebaut sein soll. Wir wollen diese sehr abstrakte Beschreibung anhand eines Beispieles klarer machen.

6.6.2 Ein- und Ausgabe

Dazu nehmen wir uns ein Programm vor, das den Namen des Benutzters von der Tastatur liest, und ihn dann wieder ausgibt. Der Einstiegspunkt in ein Programm ist in Haskell eine Funktion namens `main`, wie in C.

```
1 main = do
2   putStr "Wie ist Ihr Name? "
3   name <- getLine
4   putStrLn ("Hallo " ++ name ++ ".")
```

Die `do`-Notation haben wir schon im letzten Abschnitt kennengelernt, auch `putStr` haben wir schon gesehen[9]. In Zeile 3 lesen wir mit `getLine` eine Zeichenkette von der Tastatur ein und weisen sie der Variable `name` zu; seltsam ist hier, dass wir einen Pfeil `<-` verwenden statt des Zuweisungsoperators - wir werden sehen warum. Welchen Typ haben die Aufrufe `getLine` und `putStr`? `getLine` hat keine Parameter und liefert einen String, der Typ kann aber nicht `getLine :: String` sein - denn eine Funktion ist es nicht, weil das Ergebnis nicht immer das selbe ist. Auch bei `putStr` ist der Fall nicht ganz klar - der Parameter ist eindeutig ein `String`, aber was ist der Rückgabetyp? Fragen wir einfach den Interpreter:

```
1 *Main> :t getLine
2 getLine :: IO String
3 *Main> :t putStr
4 putStr :: String -> IO ()
```

Der Rückgabetyp für `putStr` ist `IO ()`, was bedeutet, dass `putStr` eine *Action* ist, die bei der Ausführung den Unit-Typ `()` zurückliefert. Eine Action können wir uns vorstellen als ein Objekt, das eine Sequenz von Befehlen darstellt (technisch gesehen ist `putStr` eine Action innerhalb der IO-Monade). `getLine` ist eine Action, die bei der Ausführung einen String zurückliefert. Da es keine Funktion *ist*, müssen wir dem Compiler mitteilen, dass er diese Action ausführen soll, und wir das *Resultat* der Ausführung, das ein String ist, verwenden wollen - das tun

[9] `putStrLn` ist wie `putStr`, außer dass noch ein Zeilenumbruch mit ausgegeben wird.

wir, in dem wir den Pfeil `<-` verwenden. Ein Programm selbst ist eine einzelne große Action, deshalb ist der Typ der `main`-Funktion auch `main :: IO ()`. Die `do`-Notation ist ein Weg, eine Sequenz von Actions zu kombinieren in eine neue Action.

6.6.3 Kombination von Monaden

Monaden existieren in Haskell nicht nur für Ein- und Ausgaben, sondern für eine ganze Reihe von Anwendungszwecken: Fehlerbehandlung, Zustandsspeicherung, Nebenläufigkeit und vieles mehr. Eine Monade hat eine Verbindungs-Funktion, die angibt, was der Rückgabewert sein wird, wenn sie mit einer anderen Monade zu einem größeren Baustein zusammengesetzt wird. In einem Beispielszenario soll ein Username anhand seiner ID aus einer Datenbank bezogen werden, und dann soll der Username in Großbuchstaben verwandelt werden. Das Resultat der Datenbankanfrage ist vom Typ `Maybe String`, denn wir wissen nicht, ob die gesuchte ID auch wirklich ein Ergebnis liefert. Da wir das Resultat direkt in die Umwandlungsfunktion füttern wollen, muss diese auch einen Parameter vom Typ `Maybe String` haben. Ist das Resultat der Anfrage `Nothing`, dann soll auch das Resultat der Umwandlungsfunktion `Nothing` sein. Wenn der Rückgabetyp einer Action `Maybe a` ist, und zwei solcher Actions sollen nacheinander ausgeführt werden, so soll das Gesamtergebnis `Nothing` sein, sobald auch nur eine der beiden Actions als Ergebnis `Nothing` hat. Das Vorgehen wird festgelegt von der Verbindungsfunktion der `Maybe`-Monade (`Maybe` ist nämlich nicht nur ein Typ, sondern auch eine Monade, für genau diesen Zweck). Tatsächlich ist die `do`-Notation syntaktischer Zucker, der die Verbindungsfunktionen der einzelnen Operationen nach bestimmten Regeln ausführt. Diese Regeln sorgen dafür, dass das resultierende Verhalten einer erwarteten sequenzieller Abarbeitung der Operationen entspricht, zum Beispiel, dass eine Wertzuweisung wie `name <- getLine` bewirkt, dass die folgenden Operationen auch `name` kennen.

6.7 Entwicklung von Haskell und sonstige Features

Am Ende des Überblickes über die grundlegenden Eigenschaften und Fähigkeiten von Haskell soll die vergangene Entwicklung und der gegenwärtige Stand von Haskell noch aufgeführt werden. Außerdem soll hier ein Überblick gegeben werden über die zum Teil fortgeschrittenen Techniken, die in Haskell zur Verfügung stehen, die hier aber nicht besprochen wurden.

Im September 1987 fand die Konferenz „Functional Programming and Computer Architecture“ statt, wo sich Entwickler funktionaler Programmiersprachen trafen, um sich auf eine gemeinsame Sprache zu einigen, die die besten Eigenschaften der bis dahin existierenden Sprachen vereinigen sollte. Die Ziele der Sprache waren:

- Geeignet für Lehre, Forschung und Anwendung
- Formal beschriebene Syntax und Semantik
- Frei verfügbar

- Allgemein vertretene Ideen vereinheitlich
- Unnötige Sprachvielfalt vermeiden

Im April 1990 wurde Haskell 1.0 fertig gestellt, das nach weiterer Entwicklung zu der im Jahr 1999 veröffentlichten Version der Sprachdefinition, Haskell 98 führte. Diese wurde 2002 überarbeitet und ist bis heute der aktuelle Standard von Haskell. Parallel zu Haskell 98 wird in verschiedenen Implementierungen an Erweiterungen von Haskell gearbeitet, hauptsächlich beim Glasgow Haskell Compiler. Eine pure, funktionale Sprache wie Haskell bietet sich an für parallelisiertes Rechnen; wenn Nebeneffekte ausgeschaltet sind, ist sichergestellt, dass parallele Rechnungen korrekt ausgeführt werden können. Für die Nebenläufigkeit gibt es die Haskell-Erweiterungen „Concurrent Haskell“, die Threading erlaubt und „Software Transactional Memory“, die Speicher-Transaktionen zur Synchronisation von Nebenläufigkeit nutzt und ohne Busy Waiting auskommt, und bei der keine Deadlocks auftreten können. Durch eine weitere Erweiterung wird „nested data parallelism“ unterstützt, das es erlaubt, die Nebenläufigkeit auf multicore-CPUs zu übertragen.

Es gibt noch weitere Features in Haskell, die hier nicht vorgestellt wurden: Verschachtelte Funktionen (Unterfunktionen in Funktionen), das Modul-System von Haskell, Exceptions und Fehlerbehandlung, Nebenläufigkeit, Reguläre Ausdrücke, in-Place-Datenstrukturen wie Arrays und wie wir Typklassen und Monaden selbst schreiben können. Darüber hinaus gibt es ein Binding an andere Programmiersprachen, mit der Beispielsweise C-Libraries eingebunden werden können. Haskell liefert eine Library für Modultests mit, und auf der Haskell-Webseite `http://www.haskell.org` befindet sich eine eine umfangreiche Liste von Libraries: Von GUI über Datenbankanbindung bis Netzwerk ist hier alles enthalten.

In der Praxis eingesetzt wird Haskell in vielerlei Hinsicht. Haskell.org listet Firmen wie *Credit Suisse Global Modelling and Analytics Group*, *Nokia* und *ABN AMRO* als Nutzer, Anwendung findet die Sprache in so unterschiedlicher Software wie dem verteilten Versionsverwaltungssystem *Darcs*, dem 3d-Actionspiel *Frag* und der führenden Implementierung eines PERL6-Compilers, *Pugs*.

6.7.1 Fazit

Haskell ist eine mächtige und inzwischen sehr ausgereifte Sprache. Durch die Funktionen ohne Seiteneffekte lassen sich Programme leichter aus kleinen Teilstücken aufbauen als in anderen Sprachen; der Programmierer kann sich immer auf die aktuelle Funktion alleine konzentrieren und muss nicht überlegen, welche Nebenbedingungen es gibt. Die meisten Bugs, die in Software auftreten können, die in anderen Sprachen geschrieben wurde, gibt es in Haskell einfach nicht, die strenge Typprüfung und das Fehlen eines veränderlichen Zustandes im Programmfluss vernichten die meisten Fehlerquellen schon im Vorfeld.

Allerdings gibt es auch eine Kehrseite: Für Programmierer, die aus der C- oder Java-Welt kommen, hat Haskell eine gewöhnungsbedürftige Syntax. Die Lernkurve ist bei Haskell viel steiler als in anderen Sprachen: Man muss viele der kompliziert wirkenden grundlegenden Konzepte erst komplett verstehen, bevor man relative einfache Programme schreiben kann. Ist diese Hürde allerdings erst einmal überwunden, lassen sich die tatsächlich komplizierteren Techniken relativ

leicht erlernen. Die gewonnenen Erfahrungen beim Umdenken auf die Haskell-Konzepte lassen sich auch in viele Haskell-unähnliche Sprachen übertragen.

Kapitel 7

Ada
(Alexey Korobov)

7.1 Einleitung

Ada – ein außergewöhnlich schöner und seltener Name für eine Programmiersprache. Denkt man spontan an eine elegante und geheimnisvolle Frau, so täuscht das Bauchgefühl nicht: benannt nach der „ersten“ Programmiererin Lady Ada Lovelace (1815-1852) und entwickelt in den 80en, dominiert Ada nach wie vor die Softwareentwicklung in sicherheitskritischen Bereichen. Sie ist die erste international standardisierte Hochsprache und zeichnet sich durch ihre Eleganz, Vielfalt und Effizienz aus. Diese legendäre Sprache möchten wir Ihnen im vorliegenden Dokument näher bringen und danken Ihnen bereits an dieser Stelle für Ihr Interesse!

7.2 Exkurs in die Geschichte

Nach der Softwarekrise in den 60en rückte Softwareentwicklung in den Vordergrund und machte das Modell der sog. strukturierten Programmierung populär. Als das amerikanische Verteidigungsministerium (Department of Defence, DoD) sich daraufhin Sorgen über die stets wachsende Anzahl der in Projekten verwendeten Programmiersprachen zu machen begann, wurde eine Arbeitsgruppe ins Leben gerufen mit dem Ziel, einen Ausweg daraus zu finden und die „perfekte“ Programmiersprache zu identifizieren. Doch keine der damals bekannten Sprachen erfüllte die hohen Anforderungen und es kam zu einer Ausschreibung. Aus vier Kandidaten entschied man sich schließlich für den Entwurf von Jean Ichbiah. Als am 10.12.1980 die ursprüngliche Beschreibung der Sprache gebilligt wurde, war es der Jahrestag von Lady Ada (Byron) Lovelace, die aufgrund ihrer umfassenden Kommentare zur mechanischen Rechenmaschine (sog. „Analytical Engine“) des Mathematikers Charles Babbage als erste Programmiererin bezeichnet wird. Der erste Standard Ada83 trägt daher den offiziellen Namen ANSI/MIL-STD 1815. ISO übernahm diesen Standard im Jahr 1987.

Nach Einführung des Standards wurde vom DoD die Verwendung von Ada zur

Pflicht für alle Softwareprojekte mit mehr als 30% neuem Code. Dadurch nahm die Anzahl der verwendeten Sprachen dramatisch ab: von ursprünglich 430 blieben lediglich 37 übrig. Ähnliche Regeln wurden später in anderen Staaten der NATO eingeführt. In Deutschland dagegen fand Ada eine extrem geringe Akzeptanz. Dies liegt primär an dem militärischen Beigeschmack von Ada, der mit den damaligen gesellschaftlich-politischen Einstellungen im Widerspruch stand.

1995 wurde die Sprache überarbeitet und um die modernen Modelle der objektorientierten sowie generischen Programmierung erweitert, was in einem gemeinsamen ANSI/ISO Standard Ada95 resultierte. 1998 wurde die überwachende Organisation AJPO (Ada Joint Program Office) aufgelöst und die Vorschrift zur Verwendung von Ada in Softwareprojekten innerhalb des DoD aufgehoben. Die amerikanische Luftwaffe finanzierte zum Schluss die Entwicklung eines kostenfreien GNAT Compilers, wodurch Ada erstmals in der Geschichte für die breite Anwenderschicht freigegeben wurde.

Mit dem Wegfall der Unterstützung wurde der Sprache ein Aussterben prophezeit. Doch Ada überlebte und dominiert nach wie vor in sicherheitskritischen Bereichen. Der aktuelle Stand heißt Ada2005 und entstand vollständig aus eigener Kraft der Community.

7.3 Tools & Co.

Bis 1998 galten für Ada-Compiler strenge Vorschriften und hohe Anforderungen. Sie mussten in regelmäßigen Abständen einen Validierungsprozess durchlaufen, um die Konformität mit dem Standart zu beweisen. Dies führte dazu, dass für den professionellen Einsatz auf teuere kommerzielle Compiler und Werkzeuge zurückgegriffen werden musste. Ada blieb dadurch eine lange Zeit isoliert. Erst mit der Auflösung der AJPO und der Entstehung des freien GNAT Compilers fiel der Vorhang. Es brauchte seine Zeit, bis die Community in den Genuss von Ada kommen konnte und heute – nach Einführung von Ada2005 – hofft man auf die Renaissance der Sprache.

7.3.1 Reference Manual

Die Grundlage eines jeden Ada-Entwicklers bleibt nach wie vor die Reference Manual, welche als Teil des Standards im WWW frei verfügbar ist und in dieser Hinsicht eine außergewöhnliche Kombination darstellt. Einige Quellen verweisen auf den Urvater von Ada - Jean Ichbiah, nach dessen Wunsch die Reference Manual zum legalisierten Dokument wurde und damit ein hohes Maß an Qualität gesichert war.

7.3.2 Compiler & Editoren

Wer jedoch nach einer komfortablen, mächtigen und zugleich frei zugänglichen Entwicklungsumgebung für Ada sucht, wird schnell verzweifeln. Denn diese sind meist kommerzieller Natur und zudem teuer (z.B. AdaCore). Eine alternative stellt der kostenfreie GNAT Compiler, mit dem sich komfortable Entwicklungsumgebungen in Form eines Eclipse-Plug-Ins durchaus kombinieren lassen. Zu

nennen sind dabei AonixADT und Hibachi. Letzteres soll allerdings erst Ende 2008 verfügbar sein.

TODO - screenshot von AonixADT oder Hibachi Timeline

7.4 Ausführungsmodell

Ada wurde hauptsächlich für den sicherheitskritischen Bereich der embedded- und Echtzeitsysteme konzipiert und zählt zu den strukturierten Sprachen - einem in den 70en Jahren durch die Softwarekrise populär gewordenen Programmierparadigma, das eine Codekapselung durch Prozeduren vorsieht und einen weitgehenden Verzicht auf GOTO Anweisungen durch Einführung von Kontrollstrukturen wie Schleifen, Sequenzen und Verzweigungen verlangt. Dieser Ansatz baut auf dem imperativen Ausführungsmodell auf und zählt heute zur Grundausstattung aller modernen Sprachen. Die durch imperative Programmierung bedingte Nähe zum Modell des Von-Neumann-Rechners erklärt auch die Effizienz von Ada und ihre Eignung für embedded-Systeme.

Zudem ist Ada eine sog. Wirth'sche Sprache, was auf Niklaus Wirth – den Urvater von Pascal – zurückgeht und eine „besser zu lesende“ Programmiersprache bedeutet. Angemerkt sei an dieser Stelle, dass Lesbarkeit zu den Hauptzielen bei der Entwicklung von Ada zählte und man entschied sich explizit für die Syntax der Sprache Pascal, die hauptsächlich als Lehrsprache konzipiert wurde in Punkto Lesbarkeit bis heute als Vorbild gilt.

Mit dem zweiten Standard Ada95 trat Ada der Liga von objektorientierten sowie generischen Programmierung bei und kann dadurch mit den modernen Sprachen wie Java oder C++ mithalten.

TODO - Stammbaum der Sprachen, Bild.

7.5 Besondere Sprachmerkmale

Kommen wir nun endlich zum spannenden Teil - dem Code. Falls Sie Pascal beherrschen, werden Sie eine verblüffende Ähnlichkeit in der Syntax feststellen. Anderenfalls werden Sie positiv überrascht sein, denn Ada bietet bei weitem mehr als man aus Sicht eines Java- oder C-Programmierers zu erwarten vermag. Sehen wir uns zunächst das traditionelle, jedoch ein wenig modifizierte “Hello World”-Programm an:

```
1 with Ada.Text_io; -- importiere Bibliothek Text_io
2 use Ada.Text_io;  -- und bilde einen Namensraum darauf
3
4 procedure HelloWorldX is
5 begin
```

```
6     for i in 1..3 loop
7        put_line("I'm a second nr. " & Integer'Image(i))
            ;
8        delay(1.0);
9     end loop;
10 end HelloWorldX;
```

Hinweis: der Ausdruck `'Image()` ist ein sog. Attribut und bewirkt eine Konvertierung zum String. Der Operator \& dient zur Konkatekierung.

An diesem kurzen Bespiel erkennen wir sofort die Stärke von Ada in Punkto Lesbarkeit - der Code wirkt eher einfach und selbsterklärend als kryptisch und verwirrend. Wie Sie bereits erwartet haben, erscheinen beim Ausführen des Programms drei Zeilen `I'm a second nr. N` auf der Konsole, mit jeweils einer Sekunde Verzögerung.

7.5.1 Strenge statische Typisierung auf angenehme Art

Ada ist eine statisch getypte Sprache und geht sehr streng mit den Datentypen um. Entgegen aller Bedenken schränkt diese Eigenschaft jedoch kaum ein, im Gegenteil - Ada lässt dem Programmierer dank ihrer reicher Syntax so viel Spielraum und Flexibilität, dass selbst manche modernere Sprachen wie z.B. Java in den Schatten gestellt werden.

code; enums

7.5.2 Skalare Typen

TODO: graph dazu

7.5.3 Syntaktische Highlights

** ; loops; 'Image (attribute); etc.

7.5.4 Überladung

code

7.5.5 Zeiger & Co.

die dunkle seite; code

7.5.6 OOP à la Ada

allgemeines; code

Mehrfachvererbung

kurz, wie in C++

Garbage Collection

i.d.r. keins, aber im standard

7.5.7 Generische Programmierung

code, kurz

7.5.8 Nebenläufigkeit

Randevouz, bild

7.5.9 Anschluß an andere Sprachen

kurz, evtl code?

7.5.10 Pragmas

kurz, etwas code

7.5.11 Compiler und -Laufzeittests

ariane5 fiasko beim jungfernflug 1996 :)

7.5.12 Beispielprogramm

TODO

7.6 Anwendungsgebiete

Wir erwähnten bereits, dass bei der Entwicklung von Ada großer Wert auf Effizienz, Wartbarkeit und Zuverlässigkeit gelegt wurde. Dies hatte zur Folge, dass Ada hauptsächlich in sicherheitskritischen Bereichen zum Einsatz kam und dort teilweise zur Pflicht wurde. Denn überall dort, wo Softwarefehler mit extrem hohen Kosten oder gar Menschenleben verbunden sind, kann kaum eine andere Programmiersprache mithalten. Beispiele für die Einsatzgebiete sind sehr umfangreich:

- Militär (von Panzern bis hin Atomraketen)
- Medizintechnik
- Raumfahrt (z.B. GPS, Ariane Raketen)
- Luftfahrt (Boeing, Airbus, Flugsicherung)
- Verkerssteuerung und –Überwachung
- Bahn (z.B. TGV)

7.7 Fazit

Ada hat eine außergewöhnliche Geschichte und wurde nach hohen Maßstäben mit sehr viel Sorgfalt konzipiert. Ihre Schwerpunkte sind Wartbarkeit, Effizienz und Zuverlässigkeit. Trotz der strengen Typisierung und Vielfalt bietet Ada genug Spielraum. Sie zeichnet sich durch Eleganz und Lesbarkeit aus und vereint fast alle Ansätze der modernen Programmiertechniken in sich. Ada überlebte den bürokratischen Korsett und feiert seit Kurzem ihren Einzug in die Welt der freien Softwareentwicklung.

7.8 Quellenangaben

TODO

Kapitel 8

D
(Alexander Stolz)

8.1 Einleitung

Gibt es nicht schon genug Programmiersprachen? Warum eine systemnahe Programmiersprache, wo C und C++ vermeintlich gute Dienste leisten? Nun ja, C ist 35 Jahre alt, C++ 28, immer wieder hinzugefügte Features und Erweiterungen, bei denen auf Rückwärtskompatibilität geachtet werden musste, haben ihre Spuren hinterlassen. Der C Standard ist inzwischen über 500 Seiten dick, bei C++ sind es 750 Seiten. C++ Programmierung ist aufwendig, teuer und die Programmierer von heute schreiben zwar Plattform- aber nicht Entwicklerunabhängigen Code, da jeder seinen eigenen Pool von Bibliotheken hat, die doch wieder die selben Funktionen implementieren.

Hier kommt D ins Spiel, es sieht vom Äußeren C/C++ sehr ähnlich und macht den Einstieg für Programmierer dieser und ähnlicher Sprachen sehr einfach. Darüber hinaus implementiert D *design by contract*, *Unit Tests*, *Garbage Collection*, *dynamische Arrays*, *innere Funktionen* und eine umgestaltete Template-Syntax, um nur einige Features zu nennen.

Dieser Artikel einen schnellen Einstieg in D ermöglichen und wird einige, jedoch nicht alle Features dieser sehr mächtigen Sprache behandeln.

Die Hauptziele von D sind:

- Das Reduzieren von Entwicklungskosten um mindestens 10% durch Features die sich als produktivitätssteigernd rausgestellt haben
- Vereinfachte Portierung zwischen verschiedenen Compilern, Computern und Betriebssystemen
- Unterstützung geläufiger Programmierparadigmen (imperative, strukturierte, objektorientierte, funktionale und generische Programmierung)
- Low-Level-Zugriffe auf Maschinencode-Ebene bereitstellen
- Vereinfachte Compilerimplementierung

- Einfache Anbindung von C-Bibliotheken
- Vereinfachte Codedokumentierung

8.2 Geschichte

Walter Bright, der den ersten nativen C++ Compiler schrieb (den Zortech C++) und auch in Verbindung mit einem der ersten Strategiespiele (Empire) gebracht werden kann, hatte schon im Jahre 1988 die Idee einer Nachfolgersprache für C. 1999 begann er die Sprache D zu entwickeln und veröffentlichte 2003 den ersten Compiler. Seitdem wird D ständig weiterentwickelt und ist heute in der Version 2.008 vorhanden.

8.3 Compiler

Für D sind zwei Compiler vorhanden, im weiteren Kapitel werden diese vorgestellt.

8.3.1 DMD

Dmd ist der von Walter Bright selbst geschriebene und erste Compiler für D; er wurde unter Windows entwickelt, was seiner Linux kompatibilität jedoch nicht im Weg stand. Die erste Version(0.63) des dmd wurde am 10. Mai 2003 veröffentlicht.

Installation unter Linux

Die aktuellste Version des dmd ist unter *http://www.digitalmars.com/d/changelog.html* zu finden. Hat man das Packet runtergeladen, kann man es in einen beliebigen Ordner entpacken, beispielsweise .d/ im Homeverzeichnis. Als nächstes passt man den *DFLAGS* Eintrag in der Datei

```
~/.d/dmd/bin/dmd.conf
```

mit

```
DFLAGS=-I/home/dein_nutzername/.d/dmd/src/phobos
```

an und kopiert die Datei nach */etc/*. Danach passt man die Rechte für die Compiler-Binaries in */.d/dmd/bin* wie folgt an:

```
chmod u+x dmd obj2asm dumpobj
```

und setzt symbolische links in das */usr/local/bin* Verzeichnis

```
ln -sv ~/.d/dmd/bin/dmd /usr/local/bin/dmd
ln -sv ~/.d/dmd/bin/obj2asm /usr/local/bin/obj2asm
ln -sv ~/.d/dmd/bin/dumpobj /usr/local/bin/dumpobj
```

Damit sollte der Compiler einsatzbereit und mit dmd aufrufbar sein.

Installation unter Windows

Für die Installation in Windows benötigt man den Compiler und einen gesonderten Linker, die beide unter folgenden Adressen zu beziehen sind:
Compiler: *ftp://ftp.digitalmars.com/dmd.zip*
Linker: *ftp://ftp.digitalmars.com/dmc.zip*
Dann entpackt man beide in ein Verzeichnis, der keine Leerzeichen im Namen enthält zum Beispiel *C:*
und kann das erste Programm über

```
C:\dmd\bin\dmd prog.d
```

starten.

8.3.2 GDC

Der *gdc* ist ein gcc Frontend, das erstmals von Ben Hinkle implementiert und am 02. Januar 2004 veröffentlicht wurde, dieses Frontend erreichte die Version 0.21 und wurde von Ben Hinkle aufgegeben. Fast zur selben Zeit wie Ben Hinkle (22.März 2004) stellte auch David Friedmann sein D Frontend vor, dieses unterstützt heute fast alle Features wie der dmd von Walter Bright. Eine Windowsversion des *gdc* ist nicht vorhanden, jedoch ist es möglich, den Linux-gdc unter der *Cygwin* Umgebung zu nutzen.

Installation unter Linux

Zuerst lädt man sich die neuste Version von *http://dgcc.sourceforge.net/* herunter. Dann entpackt man das Paket zum Beispiel im Homeverzeichnis:

```
tar -xvjf dc-0.24-i686-linux-gcc-4.0.3.tar.bz2
```

und kopiert die für die Ausführung benötigten Dateien in die Systemverzeichnisse:

```
cp -r ~/gdc/include/d/ /usr/local/include/
cp -r ~/gdc/bin/gdc /usr/local/bin
cp -r ~/gdc/bin/gdmd /usr/local/bin
cp -r ~/gdc/lib/* /usr/local/lib
cp -r ~/gdc/libexec/* /usr/local/libexec
cp -r ~/gdc/man/man1/* /usr/share/man/man1
```

Der gdc ist somit installiert und mit *gdc* zu erreichen.

8.3.3 Compilerbenutzung

Die Benutzung der Compiler erfolgt über die jeweiligen Ausführungsdateien dmd und gdc. Beim dmd reicht ein

```
dmd test.d
```

zum Compilieren und Linken eines Programms. Will man hingegen nur Compilieren benutzt man

```
dmd -c test.d
```

Da der gdc ein Frontend des gcc ist, ist auch die Parametersyntax ähnlich zum gcc. Zum Compilieren und Linken benutzt man

```
gdc -o test test.d
```

Nur zum Compilieren

```
gdc -c test.d
```

Diese und mehr Compilerbefehle können in den Manpages nachgeschlagen werden.

8.4 Sprachfeatures

8.4.1 Primitivie Datentypen

Um zu C kompatibel zu sein muss D natürlich alle C Datentypen unterstützen, darüber hinaus werden aber auch neue Datentypen eingeführt, die eine noch effizientere Speichernutzung erlauben sollen die zum Beispiel im Embedded-Bereich sehr wichtig ist. Hier ist eine Tabelle mit allen primitiven Datentypen:

Datentyp	**Speicherbedarf in Bit**	**Min**	**Max**	**Kommentar**
void				Hat keinen Typ
bit	1	0	1	
byte	8	-128	+128	
ubyte	8	0	255	Vorzeichenlos
short	16	-32768	32768	
ushort	16		65535	
int	32	-2147483648	2147483648	
uint	32	0	4294967295	
long	64	-9223372036854775808	9223372036854775807	
ulong	64	0	18446744073709551616	
cent	128			Reserviert
ucent	128			Reserviert
float	32	1.17549e-38	3.40282e+38	Fließkommazahlen
double	64	2.22507e-308f	1.79769e+308f	Fließkommazahlen %lf r. %lgf
real	80	3.3621e-4932	1.18973e+4932	80 bit für Intel CPU's
ireal	80	3.3621e-4932	1.18973e+4932	
ifloat	32	1.47256e-4932	1.18973e+4932	
idouble	64	2.22507e-308	1.79769e+308	
cfloat	64	2.84809e-306	1.40445e+306	
cdouble				
creal	24			
char	8	0	255(0xFF)	Vorzeichenlos 8 bit UTF-8
wchar	16	0	65535(0xFFFF)	Vorzeichenlos 16 Bit UTF-16
dchar	32	0	1114111(0x0000FFFF)	Vorzeichenlos 32 Bit UTF-32

Tabelle 8.1: *Primitive Datentypen*

8.4.2 Objektorientierung

Klassen

D ist eine objektorientierte Sprache, so darf die Klassenfunktionalität natürlich nicht fehlen. Anders als bei C++ werden bei D keine zwei gesonderten Dateien für die Klassendefinition benötigt. Bei den Klassenelementen ist es dem Programmierer überlassen, ob er die alte C++ schreibweise *visibility:* nutzt oder die Sichtbarkeit vor jedes Feld oder jede Methode setzt, ausserdem wird der Konstruktor mit dem Identifier *this* deklariert und nicht wie bei C++ und anderen Sprachen mit dem Klassennamen. Felder werden vor dem Konstruktoraufruf automatisch mit einem typspezifischen Wert initialisiert (zum Beispiel int = 0, float = NAN), dies soll die Produktivität steigern, indem eine ganze Klasse von obskuren Fehlern durch nicht initialisierte Felder von vornherein ausgeschlossen wird.

```
1  class Point{
2
3          private:
4                  //Initialisierung auf 0 von Hand
5                  int x = 0;
6                  // Automatische Initialisierung auf 0
7                  int y;
8          public this(int x, int y) {
9                  this.x = x;
10                 this.y = y;
11         }
12
13         public:
14                 this() {}
15
16                 void setX(int x) {
17                         this.x = x;
18                 }
19
20                 int getX() {
21                         return this.x;
22                 }
23
24                 void setY(int y) {
25                         this.y = y;
26                 }
27
28                 int getY() {
29                         return this.y;
30                 }
31 }
```

Vererbung

Bei der Implementierung der Vererbung haben sich die Designer von D dazu entschieden, die fehleranfällige und schwer Implementierbare Mehrfachvererbung durch ein sauberes Vererbungsmodell zu ersetzen. D unterstützt einfache Vererbung und Mehrfachvererbung von Interfaces. Interfaces werden mit dem Schlüsselwort interface deklariert und können voneinander erben, indem man einen Doppelpunkt nach dem Interfacenamen setzt. Bei Klassenvererbung wird nach dem Klassennamen und dem Doppelpnkt zuerst die abzuleitende Klasse hingeschrieben und dann mit dem Komma getrennt beliebig viele Interfaces. Der Konstruktor der jeweiligen Oberklasse kann mit dem schlüsselwort *super()* aufgerufen werden und ist nicht auf die erste Anweisung des Konstruktors begrenzt wie zum Beispiel in Java. Wenn man sichergehen will, dass eine Methode auch wirklich eine Methode von einer Oberklasse überschreibt, kann man das Schlüsselwort *override* benutzen, das einen Compilerfehler verursacht, wenn keine Methode zum Überschreiben vorhanden ist.

```
1  interface A{
2          void doStuffA();
3  }
4
5  class B{
6          this() {
7          //B Initialisierung
8          }
9          void doStuffB(){}
10 }
11
12 class C :B,A{
13         this() {
14                 ...
15                 super();
16                 ...
17         }
18
19         override void doStuffA(){}
20 }
```

Operatorüberladung

D bietet die Möglichkeit, bestimmte *Operatoren* für eigene Klassen zu überladen, hierbei gibt es für jeden überladbaren Operator eine spezielle Methode die implementiert werden kann. Hier ist eine Tabelle mit den überladbaren Unären Operatoren und ein Beispiel:

Operator	Funktion
-e	opNeg
+e	opPos
e	opCom
*e	opStar

Operator	Funktion
e++	opPostInc
e–	opPostDec
cast(type)e	opCast

Tabelle 8.2: *Unäre Operatoren*

Im Folgenden wird der *++* Operator überschrieben, dafür überschreibt man die *opPostInc-Methode* und baut die Funktionalität ein, die man gerne möchte. Hierbei spielt es keine Rolle, welchen Typ der Rückgabewert hat, man kann ihn nach Belieben wählen.

```
class Point {
        public int x;
        public int y;
        void opPostInc() {
                x++;
                y++;
        }
        void print() {
                printf("[%d,%d]",x,y);
        }
}

void main(char[][] args) {
        Point p1= new Point;
        p1.print();
        p1++;
        p1++;
        p1.print();
}
-> [0,0]
-> [2,2]
```

Das Überschreiben der Binären, Funktionsaufruf- und Array-Operatoren funktioniert auf dem gleichen Weg, hierfür kann die Onlinereferenz hinzugezogen werden *http://www.digitalmars.com/d/operatoroverloading.html*.

8.4.3 Arrays

Statische Arrays

Statische Arrays sind einfache Arrays, die auch in C bekannt sind, ihre Größe kann nach dem Anlegen nicht verändert werden. Statische Array können in Pre- und Postfix-Notation angegeben werden, wobei die Postfix-Notation eher für C/C++-Umsteiger gedacht ist, empfohlen wird die Präfix-Notation. Deklaration:

```
 int [3] arrA; //Präfix
 int arrB [3]; //Postfix
```

Will man das gerade deklarierte Array auch gleich initialisieren, trifft man auf einen Bug der Sprache, der hoffentlich in den nächsten Versionen behoben wird. Hier ist es nämlich nötig, vor die Arraydeklaration das Schlüsselwort *static* zu setzen, sonst gibt es bei der Ausführung einen *Segmentation Fault*.

```
static int [3] arrA = [2,3,5];
```

D verfügt über eine spezielle Syntax, um bestimmte Felder eines Arrays bei der Deklaration zu initialisieren und andere nicht. Beim folgenden Beispiel werden das nullte und das zweite Feld mit den Werten zwei und fünf initialisiert, das erste Feld wird durch die automatische Initialisierung auf null gesetzt.

```
static int [3] arrA = [0:2,2:5];
```

Dynamische Arrays

Dynamische Arrays enthalten in D zusätzlich zu den Daten die Länge des Arrays und einen Pointer für die *Gargabe Collection*. Die Deklaration ist ähnlich zu den statischen Arrays, man läßt einfach die Größe des Arrays weg.

```
int [] arrA = [1,2,3,4,5];
```

So hat dieser Array die größe fünf, was man auch nachprüfen kann, ohne über den ganzen Array iterieren zu müssen:

```
writefln(arrA.length);
-> 5
```

Um einen dynamischen Array zu vergrössern, gibt es in D zwei Möglichkeiten, entweder man fügt einfach ein neues Element an das Ende des aktuellen Arrays an mit dem Tilde-Operator:

```
int [] arrA = [1,2,3,4,5];
writefln(arrA.length);
-> 5

arrA ~= 6;
writefln(arrA.length);
-> 6
```

Da die Reallocation des Speichers eine eher aufwendige Operation ist, sollte damit nicht all zu leichtfertig umgegangen werden. Hierfür bietet D die Möglichkeit den Array gleich um einen ganzen Bereich zu vergrössern.

```
int [] arrA = [1,2,3,4,5];
writefln(arrA.length);
-> 5

arrA.length=10;
writefln(arrA.length);
-> 10
```

Die neu erstellten Felder werden wie gewohnt automatisch mit null initialisiert. Verkleinern ist natürlich auch möglich, hier setzt man einfach die gewünschte Größe und der Garbage Collector gibt den nicht mehr benötigten Speicher frei.

Assoziative Arrays

Assoziative Arrays werden in den meisten Sprachen als *Hashtables* bezeichnet, diese sind in D dynamische Arrays die statt einem einfachen int-Index über Keys von verschiedenen Datentypen indizieren und über spezifische Hilfsfunktionen verfügen. Im nachfolgenden Beispiel wird ein assoziativer Array mit Strings als Keys und Daten zum Speichern von verschiedensprachigen Begrüßungen für Benutzer einer Software angelegt:

```
1 string [string] begr;
2 begr["en"] = "Good day to you.";
3 begr["de"] = "Guten Tag.";
4 begr["fr"] = "Bonjour.";
```

Die Key- und Datentypen können hierbei von jedem Typ sein, auch selbst erstellte Klassentypen können verwendet werden. Ist so ein Array erstellt, bietet D verschiede Funktionen und Eigenschaften, um dem Programmierer das Leben zu erleichtern, so ist es zum Beispiel möglich über das Array zu iterieren, wie das folgende Beispiel zeigt:

```
1 foreach (string i; begr.keys) {
2         writefln(i ~ " " ~ begr[i]);
3 }
4 -> de Guten Tag.
5 -> en Good day to you.
6 -> fr Bonjour.
```

Ausserdem ist es möglich zu prüfen, ob ein Eintrag in einem Array existiert ohne von Hand den Array durchgehen zu müssen:

```
1 if ("en" in begr) {
2         begr["en"] = "Hello";
3 }
```

Wie bereits erwähnt stellen assoziative Arrays verschiedene Methoden bereit, diese werden in den folgenden zwei Beispielen erklärt. Die *remove-Funktion* entfernt einen Eintrag aus dem Array:

```
1 begr.remove("fr");
```

Die sort-Funktion sortiert das Array alphabetisch nach den Keys:

```
1 begr.sort();
```

8.4.4 Funktionen/Methoden

Funktionen und Methoden sind bis auf ein paar Spezialfälle, die in diesem Kapitel behandelt werden, mit C/C++ identisch. Im folgenden Beispiel ist eine einfache Funktion deklariert, die einen String auf der Konsole ausgibt. In der main-Funktion wird diese Funktion mit einem Stringparameter aufgerufen und der String wird ausgegeben. Ausserdem erfolgt ein zweiter Aufruf, den es so nicht in den meisten Sprachen gibt, die Funktion wird auf dem String aufgerufen, dies ist nur syntaktischer Zucker und auch eher unüblich, funktioniert jedoch in D.

```
1  void print(string str) {
2          writefln(str);
3  }
4
5  void main(char[][]args) {
6          string teststr = "Hallo Welt";
7          print(teststr);
8          teststr.print();
9  }
10 -> Hallo Welt
11 -> Hallo Welt
```

Variable Parameterlisten

D bietet die Möglichkeit variable Parameterlisten zu verwenden und anders als in C/C++ ist das auch ohne einen benötigten Parameter am Anfang möglich.

```
1  void print(...) {}// So nicht in C möglich
```

Um auf die Elemente der Parameterliste zuzugreifen, bietet D einen void-Pointer *_argptr* auf das erste Element der Liste. Um durch die Liste durchzuiterieren kommt man nicht an Pointerarithmetik vorbei und muss den Pointer von Hand weitersetzen.

```
1  void print(...) {
2          string s1 = *cast(string*)_argptr;
3          _argptr += string.sizeof;
4          string s2 = *cast(string*)_argptr;
5          writefln(s1 ~ " " ~ s2);
6  }
7  -> String1 String2
```

Wie man sich denken kann, ist das ganze sehr fehleranfällig, da das Programm gnadenlos abstürzt sobald man einen falschen Typen übergibt oder die Grösse des Pointers übertritt. Dafür gibt es den dynamischen *_arguments* Array, dieser enthält die Anzahl der Parameter in der Parameterliste und die jeweiligen Typen.

```
1  void print(...) {
2  string outp = "";
```

```
for (int i = 0; i < _arguments.length; i++) {
        if (_arguments[i] == typeid(string)) {
                outp = outp
                        ~ " " ~
                        *cast(string*)_argptr;
        _argptr += string.sizeof;
        }
}
writefln(outp);
}
int main(char[][]args) {
        string teststr = "Hallo Welt";
        string teststr2 = "Wie gehts";
        print(teststr,teststr2);
        return 0;
```

Um sich den ganzen Ärger zu sparen, wenn man wirklich nur eine variable Parameterliste von einem Typen will, gibt es eine es auch *typensichere Parameterlisten*:

```
int sum(int[] ar...) {
        int erg;
        foreach (int x; ar) {
                erg += x;
        }
return erg;
}

int main(char[][]args) {
        printf("%d",sum(1,2,3,4,5,6,7,8));
return 0;
}
-> 36
```

Funktionsparameter

D bietet eine ganze Reihe von Funktionsparamtern an, die eine sicherere und eindeutigere Programmierung erlauben sollen.

in

Der in Parameter ist equivalent zu *final const* wobei weder die Referenz noch der Wert verändert werden können. Das folgende Beispiel wird nicht kompilieren:

```
void foo(in int x) {
        x++;
}
```

out

Out Parameter sind *Call by Reference* und werden standardmäßig mit null initialisiert:

```
void func(out int i) {}
int x = 5;
print("%d",func(x));
-> 0
```

```
void func(out int i) {
        i=5;
}
int x = 10;
print("%d",func(x));
-> 5
```

inout/ ref
Parameter die mit dem *ref* oder *inout* Schlüsselwort anotiert sind, werden per *Call by Reference* übergeben, es ist in Zukunft geplant das inout Schlüsselwort aus der Sprache zu entfernen:

```
void func(ref int i) {
        i++;
}
int x = 10;
print("%d",func(x));
-> 11
```

Ein weiterer Einsatzort dieser Schlüsselwörter sind foreach Schleifen wie im folgenden Beispiel, hier ist es möglich beim *Durchiterieren* durch Arrays oder Listen den geholten Wert an Ort und Stelle zu verändern:

```
int main(char[][]args) {
        static int [5] b= [1,2,3,4,5];
        foreach (ref int a;b) {
                a+=5;
        }

        foreach (int a;b) {
                printf("%d ",a);
        }
return 0;
}
-> 6 7 8 9 10
```

final
Final Parameter haben eine feste *Referenz* die nicht geändert werden kann.

const
Const Parameter schützen den Wert eines Parameters, so dass er nicht verändert werden kann.

lazy
Lazy Parameter werden nicht beim Funktionsaufruf sondern erst beim Aufruf des Parameters ausgewertet, so kann ein lazy paramater null bis n-mal ausgeführt werden. Einem *lazy void* Parameter können Funktionen und Werte von jedem Typ übergeben werden wie in folgendem Beispiel:

```
 void foo(int n, lazy void exp) {
        while(n--) exp();
}

int main(char[][]args) {
int x=10;
        void add() {x++;}
        foo(3,3);//macht 3 mal nichts
        foo(3,x--);//subtrahiert 3
        foo(6,add);//addiert 6
        printf("%d",x);

return 0;
}
-> 13
```

invariant
Invariante Parameter sind das Gleiche wie *final const* und somit weder auf Referenz- noch auf Werteebene editierbar.

Innere Funktionen

Funktionspointer und Delegates

Funktionspointer und *Delegates* haben in D eine fast identische Schreibweise und sollen in Zukunft vereinigt werden. Im Moment nutzt man Funktionspointer für statische Funktionen und Delegates für nicht statische Funktionen und Methoden. Im folgenden Beispiel wird einer Funktion ein Wert und ein Funktionspointer übergeben, diese Funktion führt die übergebene Funktion auf dem Wert aus. Da äussere Funktionen immer statisch sind, funktioniert dieses Beispiel, würde man das selbe mit Delegates probieren würde es nicht compilieren.

```
void add(ref int x) {
        x++;
}

void sub(ref int x) {
        --x;
}
void op(ref int g, void function(ref int x) fp2) {
        fp2(g);
}

int main(char[][]args) {
```

```
13          int f = 10;
14          op(f, &sub);
15          op(f, &add);
16          op(f, &add);
17          printf("%d",f);
18  return 0;
19  }
20  -> 11
```

Delegates funktionieren in D nur mit inneren Funktion und Methoden. Das zweite Beispiel demonstriert den Einsatz von Delegates, dem Delegate wird erst eine innere Funktion und dann eine andere übergeben und der Delegate ruft die jeweilige zugewiesene Funktion auf. So können mit dem selben Funktionsaufruf verschiedene Sortierverfahren verwendet werden und dynamisch ausgetauscht werden. Hier werden ein naives Mischverfahren und das FisherYates Verfahren, was auch als perfektes Mischverfahren bezeichnet wird[Knu69], implementiert und über den Delegate aufgerufen.

```
1   /*Die Delegate-Deklaration*/
2   int[] delegate() randomArray;
3
4   int main(char[][]args) {
5       /*Seed setzung*/
6       srand(cast(uint) time(null));
7
8       /*Das naive Verfahren*/
9       int[] naiveRandomArray(){
10              int [] er= [1,2,3,4,5];
11          for (int i=0; i< er.length; i++) {
12                  swap(er[i], er[rand() % er.length]);
13          }
14          return er;
15      }
16
17      /*Das durchdachte verfahren*/
18      int[] fisherYatesRandomArray() {
19              int [] er= [1,2,3,4,5];
20              for (int i = er.length - 1; i>0; i--) {
21                   swap(er[i], er[rand() % (i + 1)]);
22              }
23      return er;
24      }
25
26      /*Delegate Zuweisung*/
27      randomArray = &naiveRandomArray;
28      foreach (int i; dg()) {
29              printf("%d ",i);
30      }
31
32      printf("\n");
```

```
33
34      /*Delegate Zuweisung*/
35      randomArray = &fisherYatesRandomArray;
36      foreach (int i; dg()) {
37             printf("%d ",i);
38      }
```

8.4.5 Templates

Generische Programmierung ist ein Verfahren, um Software möglichst allgemein zu entwerfen und mit verschiedenen Datentypen und Datenstrukturen wiederverwendbar zu machen, hierfür bietet D *Templates* für verschiedene Sprachkonstrukte wie Funktionen, Variablen, Structs und Klassen an und erlaubt hierbei auch Mixins die im Verlauf dieses Kapitels besprochen werden. Eine Beschränkung von Templates in D ist, dass man sie nicht auf Nicht-Statische Methoden oder innerhalb von Funktionen deklarieren kann.

Template Funktionen

Templatefunktion werden in einem speziellen template-Block geschachtelt dieser ermöglicht es mehrere Funktionen auf einmal generisch zu implementieren. Der Block wird mit template(TYPE) erstellt, wobei auch mehrere Typen mit Kommata getrennt möglich sind. Der Aufruf der Funktionen erfolgt über einer dem Casting ähnliche Syntax *TEMPLATENAME!(TYPE,..).funktion()*.

```
1   template Gen(T) {
2          void quad(ref T a) {
3                  a *= a;
4          }
5
6          T sum(T x, T y) {
7                  return x+y;
8          }
9  }
10
11 int main(char[][]args) {
12         int iX = 22;
13         double dX = 12.2;
14         Gen!(int).quad(iX);
15         Gen!(double).quad(dX);
16         printf("%d %lf \n", iX,dX);
17
18         printf("%d \n",Gen!(int).sum(5,5));
19         printf("%lf \n",Gen!(double).sum(34,36));
20
21 return 0;
22 }
23
```

Template Klassen/Structs

Die Templatesyntax für *Structs* und *Klassen* ist identisch, man schreibt ein *(TYPE)* nach dem jeweiligen Konstruktnamen wie im folgenden Beispiel zu sehen:

```
1
2  class Point(T) {
3          public T x;
4          public T y;
5
6          public this(T x, T y) {
7                  this.x = x;
8                  this.y = y;
9          }
10
11 }
12
13
14
15 int main(char[][]args) {
16         Point!(int) p1 = new Point!(int)(2,3);
17         Point!(double) p2 = new Point!(double)(4.4, 63.3);
18
19         printf("%d %d \n",p1.x, p1.y);
20         printf("%lf %lf \n",p2.x,p2.y);
21 return 0;
22 }
23 -> 2 3
24 -> 4.40000 63.30000
```

Mixins

Mixins sind Templatefunktionen und -Klassen die man in vorhandenen Code einbindet, wobei diese dann die Umgebung sehen und Variablen verändern können. Sie sind ein gutes Mittel um das Copy/Pasten von gleichem Code zu vermeiden. Bei dem folgenden Beispiel wird ein *Mixin* verwendet um eine Variable zurückzugeben, dabei kann man gut sehen wie zwischen lokalem und globalem Kontext unterschieden wird.

```
1  string x="global";
2
3  template Gen(T) {
4          T doSmt() {
5                  return x;
6          }
7  }
8
9  int main(char[][]args) {
10         string x = "lokal";
11         mixin Gen!(string);
```

```
12          /*Aufruf in lokalem Kontext*/
13          writefln(doSmt());
14          /*Aufruf in globalem Kontext*/
15          writef(Gen!(string).doSmt());
16
17  return 0;
18  }
19  -> lokal
20  -> global
```

Mixins auf Klassenebene können verwendet werden, um Methoden in bestehende Klassen einzubinden. Bei dem foldendem Beispiel wird die *print()* Funktion in die Klasse Foo eingebunden, die diese ganz normal als Methode benutzen kann. Es besteht sogar die Möglichkeit Aliase zu benutzen, da man die Variablennamen des lokalen Kontexts nicht immer kennt und sie auch nicht bei jeder Anwendung gleich sind.

```
1   template Gen(alias x) {
2           void print() {writefln(x);}
3   }
4
5   class Foo{
6           private int var1;
7           mixin Gen!(var1);
8   }
9   int main(char[][]args) {
10          Foo f= new Foo;
11          f.var1=22;
12          f.print();
13  return 0;
14  }
15  -> 22
```

Implicit Function Template Instantiation

Implicit Function Template Instantiation oder kurz *IFTI* ist eine neu eingeführte Schreibweise um die Templatesyntax von Funktionen zu verkürzen, wobei man den Templateblock komplett wegläßt und stattdessen die schon von Klassen und Structs bekannt Syntax nutzt. Beim Aufrufen der Methoden ist auch der Cast nicht mehr nötig, hier übernimmt der Compiler die Bestimmung der Typen.

```
1   T quad(T)(T x) {
2           return x * x;
3   }
4
5   int main(char[][]args) {
6           printf("%d \n", quad(2));
7           printf("%lf \n", quad(2.23));
8   return 0;
9   }
```

```
-> 4
-> 4.97290
```

8.4.6 Produktivität

Dokumentation

Da D besonders viel Wert auf die Erhöhung der Produktivität und die Senkung der Produktionszeit legt, hat man auch die Erstellung der Dokumentation nicht dem Zufall überlassen. Bei D wird die Dokumentation gleich in den Code eingebunden und muss nicht doppelt geschrieben werden, was vor allem der Konsistenz der Dokumentation beim Schreiben der Dokumentation und eventuell bei Änderungen durch die Wartung des Codes zu Gute kommt. Die genauen Ziele von D für die Dokumentation sind:

- Gutes Aussehen der Dokumentation und nicht erst nach der Extraktion durch ein Tool
- Simple und einfach zu schreibende Syntax
- Einfache Unterscheidung vom Code, so dass es keine Verwechslungen gibt
- Nutzung von vorhandenen Dokumentationstools wie zum Beispiel *Doxgen* sollte möglich sein

D Kommentare werden ähnlich zu anderen Sprachen mit

```
 ///Hallo ich bin ein Kommentar
```

oder mit

```
 /**
        Die ist ein Kommentarabschnitt
*/
```

angegeben. D hat ausserdem noch eine weitere Notation, die es erlaubt Kommentare zu schachteln

```
 /++
        Kommentar1
        /**
                Kommentar2
        */
+/
```

Bei D kümmert sich der Compiler selbst um die Verarbeitung von Kommentaren, so werden sie im Sourcecode nicht einfach ignoriert, sondern mitgeparsed und an verschiedene Codekonstrukte drangehängt. Hierbei gibt es bestimmte Regeln, die es zu beachten gilt, um eine korrekte Zuordnung hinzubekommen. Kommentare die vor der Moduldeklaration stehen, gelten für das gesamte Modul, wenn ein Kommentar in der gleichen Zeile rechts von einem Codekonstrukt steht, bezieht er sich auf dieses Codekonstrukt. Wenn ein Kommentar nur aus

dem Schlüsselword *ditto* besteht dann wird der zuletzt aufgeführte Kommentar verwendet.

```
int a;  /// Kommentar für a; b hat keinen Kommentar
int b;

/** Kommentare für c und d */
/** Kommentare für c und d */
int c;
/** ditto */
int d;

/** Kommentare für e und f */
int e;
int f;        /// ditto

/** Kommentar für g*/
int g; /// ditto

/// Kommentar für C und D
class C
{
    int x;    /// Kommentar für C.x

    /** Kommentar für C.y und C.z */
    int y;
    int z;    /// ditto
}

/// ditto
class D
{
}
```

Um bestimmte Informationen standardisiert an den Compiler oder Tools zu geben nutzt man in D sogenannte *Sektionen* die zum Beispiel auch aus *JavaDoc* bekannt sind.

```
/**
   Author: Alexander Stolz, stolz.alexander@gmail.com
   Bugs: Funktion Foo wirft Exception
   Date: 30.Dezember 2007
*/
```

```
/**
   Deprecated: Neuere Funktion Foo2 benutzen
*/
deprecated void Foo() {}
```

Alle möglichen Sectionen sind in der Onlinereferenz dokumentiert und können dort nachgeschlagen werden *http://www.digitalmars.com/d/ddoc.html*. Ausser-

dem wird dort noch tiefer auf die Dokumentierungmöglichkeiten wie zum Beispiel HTML-Einbettung oder Makros eingegangen.

Garbage Collection

D ist eine *managed* Sprache, das heisst es ist nie nötig Speicherplatz von Hand freizugeben, der *Garbage Collector* von D übernimmt das für den Programmierer, dieser alloziiert je nach Bedarf und man muss sich um Nichts weiter kümmern. *Gemanagete Software* wird allgemein als langsam betrachtet, aber dies ist eine Annahme die nicht ganz der Wahrheit entspricht.

- Gemanagete Software hat keine Speicherlecks und kann viel länger laufen ohne dabei immer langsamer zu werden.
- Der Code der zum Managen des Speichers benötigt wird, kann bei grossen Projekten enorm groß werden und mehr Code bedeutet mehr Paging im Cache was das Programm langsamer macht.
- Moderne Garbage Collectoren unterstützen Heapkomprimierung, was die Anzahl der referenzierten Seiten reduziert und zu mehr Treffern im Cache führt.
- Gemanagete Software hat weniger schwer zu findende Pointer Bugs was die Produktionszeit eines Programms extrem verkürzt.

8.4.7 Zuverlässigkeit

Contracts

Contract Programming ist eine Programmiertechnik die Code sicherer, leichter zu Debuggen und übersichtlicher macht. Das Contractskonzept stützt sich auf vier Contractfälle: *Vorbedingungen*, *Nachbedingungen*, *Fehler* und *Invarianten.*

Asserts

Asserts sind in D etwas moderater gestaltet als in C/C++ und können im normalen Programm eingesetzt werden und nicht nur zum Debuggen. Der enscheidende Vorteil von *D asserts* zu *C/C++ asserts* ist, dass sie das Programm bei einer negativen Aussage nicht einfach Beenden sondern eine *AssertError-Exception* werfen, die gefangen und weiterverarbeitet werden kann.

```
1  double sqr(double i) {
2          try{
3          assert(i >= 0);
4          } catch (Exception e) {
5                  i *= -1;
6          }
7          return sqrt(i);
8  }
9
10 int main(char[][]args) {
11         writef(sqr(-25.0));
12 return 0;
13 }
```

Pre und Post Conditions

Pre und Post Conditions werden genutzt, um die übergebenen Parameter zu testen *(Pre)* und um den Rückgabewert zu prüfen *(Post)*, im *body* block wird der eigentliche Code reingeschrieben. Die *in* und *out* Blöcke sind optional und können unabhängig voneinander genutzt werden.

```
long square_root(long x)
    in
    {
        assert(x >= 0);
    }
    out (result)    {
        assert((result * result) <= x && (result+1) * (result+1) >= x);    }
    body
    {
        return cast(long)sqrt(cast(real)x);
    }
```

Klasseninvarianten

Klasseninvarianten sind Bedingungen einer Klasse die immer wahr sein müssen, ausser innerhalb eines Methodenaufrufs. Beispielsweise in einer Datumsklasse wie im folgenden Beispiel:

```
class Datum {
        private int tag;
        private int monat;
        private int jahr;

        invariant(){
                assert(tag >= 1 && tag<=31);
                assert(monat >=1 && monat <= 12);
        }
}
```

Unit Tests

Bei Unittests geht D einen Schritt weiter als andere Sprachen, hier werden keine extra Tools benötigt und es müssen nicht einmal extra Dateien angelegt werden, man testet da wo es am sinnvollsten ist, in den Klassen selbst. Die Unittests werden nur mit der Compilerweiche *-unittest* mit Compiliert, dann werden sie noch vor der main-Funktion ausgeführt.

```
class Mathe{

        int sum(int x, int y) {return x+y;}
        int dif(int x, int y) {return x-y;}

        unittest{
                Mathe m= new Mathe;
```

```
8                 assert(m.sum(2,3) == 5);
9                 assert(m.dif(15,6) == 9);
10        }
11  }
12
```

Exceptions

Exceptions werden bei D mit den try-catch-finally Blöcken behandelt, wobei das finally nach der Abarbeitung des try Blocks immer aufgerufen wird, egal ob eine Exception geworfen bzw. gefangen wurde oder nicht.

```
1   void operation() {
2           try{
3                   writefln("try");
4                   assert(false);
5           }catch(Exception e){
6                   writefln("catch");
7           }finally{
8                   writefln("finally");
9           }
10  }
11  -> try
12  -> catch
13  -> finally
```

8.5 Fazit

D ist eine mächtige und dabei sehr flexible und angenehm zu Programmierende Sprache. D implementiert die besten features aus C++ und hat ausserdem Einflüsse von anderen Sprachen wie *Java*, *C#* oder *Eiffel*, welche sie zu einer Highlevelsprache machen ohne ihr die Möglichkeiten des Lowlevel Programmierens zu nehmen.

Allerdings gibt es auch Stimmen die D kritisieren, so wird zum Beispiel die Standardbibliothek von D die manchen C++ Programmierern noch zu fehleranfällig und unausgereift ist. Oder der noch unfertige Support für dinamische Bibliotheken.

Insgesammt ist D eine Sprache von der man in Zukunft noch hören könnte, wenn die Ungereimtheiten und Bugs aus der Sprache verschwinden.

Kapitel 9

objective C (Marco Rancinger)

9.1 Einleitung

Im folgenden Dokument soll ein grober Überblick über die Programmiersprache Objective C gegeben werden. In diesem Zusammenhang werden wir uns mit geschichtlichen wie aktuellen Aspekten über diese Sprache beschäftigen, ihre Vor- und Nachteile im Bezug auf die aktuellen Main-Stream-Sprachen diskutieren, die Grundlagen ihrer Syntax kennenlernen und zu verstehen versuchen wie die Ausführung der Programme und die Speicherverwaltung funktionieren.

9.1.1 Wer sollte das hier lesen

Dieser Artikel ist gedacht für Studenten oder interessiere Fachleute mit Vorkenntnissen in objektorientierter Programmierung, die sich in die Sprache Objective C einlesen wollen. Hierbei soll keinesfalls der Anspruch der Vollständigkeit erfüllt werden, da eine Umfassende Einführung in Objective C den Rahmen dieses Artikels deutlich sprengen würde. Stattdessen soll mit diesem Artikel eine allgemeine Kurzeinführung, in deutscher Sprache, geschaffen werden, mit deren Hilfe es möglich sein soll einen schnellen Einstieg in die Sprache zu finden und schnell zu ersten Erfolgserlebnissen zu gelangen.

Für einen tieferen Einblick in die funktionalen Möglichkeiten der Sprache soll bereits an dieser Stelle auf die offizielle Dokumentation, die Sie auf den Apple Developer Seiten[1] finden, verwiesen werden. Auf dieser Seite befindet sich, neben einer ausführlichen Darstellung aller Sprachelemente von Objective C, auch eine umfangreiche Sammlung an Code-Beispielen mit deren Hilfe auch komplexere Anwendungen mit Leichtigkeit erstellt werden können.

9.1.2 Warum Objective C?

Natürlich fragt sich der Leser an dieser Stelle, warum man sich mit Sprachen wie Objective C, die ja offensichtlich ein Schattendasein führen und kaum eine

[1] Zu finden unter www.developer.apple.com

praktische Anwendung finden, beschäftigen soll. Die Antwort die mir spontan auf diese Frage einfällt ist: *Weil das vielleicht nicht immer so bleibt!*

Selbstverständlich ist es richtig, dass die Sprache Objective C sich nie wirklich durchgesetzt hat. Dennoch muss man zugeben, und ich hoffe dies im weiteren Verlauf dieses Artikels auch noch klar herausstellen zu können, dass Objective C einige sehr innovative Sprachelemente enthält, und sie in Sachen Laufzeitverhalten und in ihrem reichhaltigen Angebot an vordefinierten Bibliotheken, jederzeit mit Main-Stream-Sprachen wie C++ und Java mithalten kann.

Diesen großen Vorteil hat auch die Firma NeXT vor vielen Jahren erkannt und Objective C zur Basissprache für ihr Betriebssystem NeXTStep gemacht. Inzwischen heißt die Firma NeXT Apple und NeXTStep wurde zu MAC OS X, doch an der Basissprache Objective C hat sich nichts geändert, was deutlich macht wie erfolgreich man mit ihr arbeiten kann. So werden bis heute noch fast alle Programme für Mac OS X in Objective C geschrieben, was uns einen weiteren wichtigen Grund bietet uns etwas eingehender mit dieser Sprache zu beschäftigen. In den letzten Jahren schafft Apple es nämlich immer mehr an Marktanteil zu gewinnen, was unter anderem daran liegt, dass Apple's Betriebssystem Mac OS X bei den Useren so gut ankommt. Wenn dieser Trend sich auch zukünftig durchsetzt ist es für einen Programmierer auf jeden Fall von Vorteil Kenntnisse in der Basissprache dieses Betriebssystems aufweisen zu können. Denn je mehr Marktanteil Apple-Systeme einnehmen desto größer wird auch der Bedarf an Programmen für diese Systeme werden.

Vorteile Objektorientierter Programmierung

Will man Argumente für die Verwendung von Objective C finden so beginnt man am Besten damit zu erwähnen, dass es sich um eine sehr gut durchdachte objektorientierte Sprache handelt. Objektorientiere Sprachen wie Objective C, C++ oder Java haben im Gegensatz zu prozeduralen Sprachen wie z.B. C einige deutliche Vorteile, die hier nun kurz beleuchtet werden sollen.

- Die Verwendung objektorientierter Programmiersprachen ermöglicht es dem Entwickler seinen Code nicht nur funktional sondern auch logisch sehr sauber zu Modularisieren.
- Dies verbessert zum einen wesentlich die Übersichtlichkeit und Wartbarkeit des Codes und verkürzt in aller Regel drastisch die Entwicklungszeit der Programme.
- Dies ermöglicht es in vielen Fäller überhaupt erst, umfangreiche Software-Projekte zu realisieren.

Der Lauf um die Wette

Vergleichen wir Objective C im Bezug auf das Laufzeitverhalten mit anderen Objektorientierten Sprachen, so müssen wir feststellen, dass es sehr gut abschneidet. So schlägt es z.B. um Längen die immer populärer werdende Sprach Java. Deren großes Manko ist die Tatsache, dass Java-Programme auf einer virtuellen Maschine interpretiert werden. Dies ist für Java zwar unerlässlich, da sie als plattformunabhängige Sprache konzipiert wurde, schlägt sich allerdings deutlich in der Laufzeit der Programme nieder. Spannender ist da schon der

Vergleich mit C++. Wie ihr großer Bruder C werden Programme in C++ und Objective C kompiliert und dann direkt vom Betriebssystem bzw. auf der Hardware ausgeführt. Tatsächlich, ist die Laufzeit fast identisch.

Darf's ein bisschen mehr sein?

Der wohl beeindruckendste Vorteil von Objective C ist die Vielfalt an Bibliotheken die schon bereit stehen. Das Cocoa Framework, auf dem Objective C basiert, und welches im weiteren Verlauf dieses Artikels noch eingehender beleuchtet wird, bietet Klassen und Funktionen für fast jeden Anlass. Es ist egal ob sie schnell mal eben eine graphische Benutzeroberfläche bauen, über ein Netzwerk kommunizieren oder einfach eine kleine Konsolenapplikation schreiben wollen, mit Objective C und Cocoa ist alles was sie brauchen schon da. Dies ist ein Luxus der in manch anderer Sprache fehlt. Zwar gibt es für C++ inzwischen (es ist ja auch schon lange genug auf dem Markt) auch eine Menge Bibliotheken, diese muss man sich i.d.R. allerdings mühsam zusammensuchen. In Objective C genügt ein einfacher *import* Befehl um die gewünschte Funktionalität zu ergänzen.

Man kann, um diesen Abschnitt damit abzuschließen, also gerosst behaupten, dass Objective C eine echte Alternative zu den aktuellen objektorientierten Main-Stream-Sprachen C++ und Java ist. Es vereinigt einige der größten Vorteile beider Sprachen miteinander. Das ausgezeichnete Laufzeitverhalten von C++ und die funktionale Vielfalt von Java. Wenn man dann noch berücksichtigt, dass die Syntax der Sprache wesentlich konsequenter ist, als die von C++, so muss man sich wirklich die Frage stellen, wie sein konnte, dass sich aus zwei Sprachen die zeitlich fast synchron entwickelt wurden, ausgerechnet diejenige durchgesetzt hat, die technisch eigentlich weniger innovativ ist.

9.2 Geschichte der Sprache

Bevor wir uns näher mit der Programmerstellung unter Obective C befassen wollen wir in diesem Abschnitt zunächst einmal einen Blick auf die Wurzeln der Sprache werfen. Hierbei wollen wir klären wann, wo, wie und warum Obective C entstanden ist und wie es dann mit dieser Sprache weiter ging.

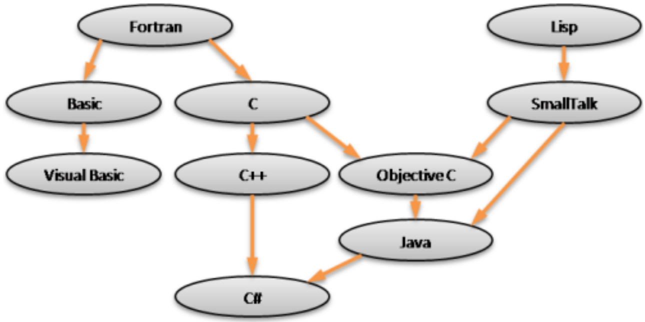


Abbildung 9.1: Entwicklung der Programmiersprachen
Entnommen Lit. [??]

9.2.1 Warum so viele C's?

In den 1970er Jahren erblickte eine Programmiersprache mit dem Namen C das Licht der Welt. Diese stellte für damalige Verhältnisse einen geradezu beeindruckend innovativen Stand der Technik dar.

Da sich die Welt allerdings weiter drehte und die ersten objektorientierten Sprachen wie z.B. *SmallTalk* drohten C zu verdrängen, wurde der Ruf nach einer objektorientierten Erweiterung von C laut. Diesen Ruf hörte, das ist in der Welt der Informatik hinreichend bekannt, der Entwickler Bjarne Stroustrup und schuf ab 1979 bei den Bell Laboratories eine Sprach mit dem Namen *„C mit Klassen"*. Da es sich hierbei, wie man sich beim Lesen des Namens leicht vorstellen kann, nicht unbedingt um einen sehr praktikablen Namen handelte, wurde die Sprache nach relativ kurzer Zeit in C++ umgetauft. Der Name C++ ist hierbei eine Anspielung auf den Iterations-Operator von C und soll bedeuten, dass es sich hierbei um die nächste Version von C handeln sollte.

Weniger bekannt ist allerdings, dass, etwa zur selben Zeit zu der auch C++ entwickelt wurde, auch ein anderer junger Entwickler Namens Brad Cox den Ruf nach einer objektorientierten Version von C vernahm. Brad Cox machte sich nun daran einen völlig eigenen Ansatz zur Erweiterung von C zu schaffen. Hierbei baute Cox auf den ANSI-C Standard auf und erweiterte diesen um die Möglichkeit der objektorientierten Programmierung. Er tat dies allerdings in Anlehnung an die, zu seiner Zeit populärste objektorientierte Sprach SmallTalk.

So kommt es zu Stande, dass die Syntax von Objective C im ersten Moment leicht inkonsistent wirkt. Dieser Eindruck verfliegt allerdings recht schnell, wenn man sich etwas eingehender mit der Sprache beschäftigt. So erleben wir nichts anderes als eine bewusste und klare Trennung von prozeduralen und objektorientierten Sprachelementen.

9.2.2 Nach der Entwicklung

Nachdem Cox die Entwicklung der Sprache abgeschlossen hatte, zeigte die 1985 von Steve Jobs gegründete Firma NeXT interesse daran. Sie integrierte Objective C in die *GNU C Compiler Collection*. Die Motivation hierbei bestand darin, dass NeXT Objective C als Basissprache für ihr neues Betriebssystem NeXTStep verwenden wollten. 1996 wurde NeXT dann von Apple aufgekauft, und NeXTStep wurde zur Basis von Mac OS X. So fand Objective C seinen Einzug in OS X, wo es bis heute als Standartsprache Verwendung findet.

9.3 Die Entwicklungsumgebung

Bei der Entwicklung von Programmen geht man so vor, dass man die gewünschte Funktionalität zunächst in Form eines „Programmcodes" beschriebt. Dieser Code wird in einer Programmiersprache, wie z.B. Objective C, die für Menschen lesbar ist formuliert. Nun besteht allerdings das Problem, dass ein für Menschen lesbarer Code für den Computer in aller Regel nur sehr wenig Sinn ergibt. Damit dieser den Code also auch verstehen kann muss der Code zunächst in ein für den Computer verständliches Format, sprich in Maschinen-Code, konvertiert werden. Diese Aufgabe erledigt ein so genannter Compiler. Abbildung 9.2 zeigt, grob vereinfacht, noch einmal anschaulich wie dies funktioniert.

Wir erkennen also, dass uns die beste Programmiersprache nicht das geringste nützt, wenn wir keinen Compiler haben um die Programme, die wir schreiben, in Maschinencode zu übersetzten. Daher wollen wir uns in diesem Abschnitt kurz mit den Entwicklungswerkzeugen beschäftigen, die man benötigt um Programme in Objective C zu schreiben.

9.3.1 Tools unter Mac OS X

Wie weiter oben bereits eingehend beschrieben wurde, ist Objective C die Standartsprache unter Mac OS X. So ist es also auch nicht weiter verwunderlich, dass die Firma Apple Eine Vielzahl von Tools herausgibt, mit denen Entwickler unter Mac OS X Programme in Objective C erstellen können. Die beiden wichtigsten, *XCode* und den *Interfacebuilder*, wollen wir uns daher in diesem Abschnitt einmal anschauen.

XCode

XCode ist eine Entwicklungsumgebung unter Mac OS X, die neben Objective C auch eine Vielzahl von anderen Programmiersprachen, darunter Java, C, C++, und auch Skriptsprachen wie z.B. XML unterstützt. Hierbei ist XCode nicht nur ein einfacher Compiler, der die Programme übersetzt, es enthält neben einem sehr komfortablen Texteditor auch Möglichkeiten zur Projektverwaltung. Kurzgesagt, alles was das Entwicklerherz begehrt ist in der Graphischen Entwicklungsumgebung XCode enthalten. Abbildung 9.3 zeigt, dass dies alles in der, gewohnt übersichtlichen und intuitiven, graphischen Umgebung von Mac OS X eingebettet ist, und somit beste Voraussetzungen für schnelle Erfolge bei der Programmerstellung bietet.

Zwar ist XCode nicht in der Standartinstallation von Mac OS X enthalten, kann allerdings ohne Schwierigkeiten nachinstalliert werden. Erhältlich ist XCode im Downloadangebot auf den Apple Developer Seiten[2] oder, für User von Mac OS X Tiger, direkt von der Installations-DVD.

Interface-Builder

Neben XCode ist auch noch der Interface-Builder eine Erwähnung wert. Dies ist ein sehr komfortables Tool, zur Entwicklung von graphischen Benutzeroberflächen. Hierbei werden GUIs einfach mittels drack&drop zusammengebaut und können per Mausklick mit Funktionen versehen werden. Zwar eignet sich dieses Vorgehen in der Regel nicht für größere Projekte, für kleinere Anwendungen ist es allerdings eine sehr praktische Lösung.

9.3.2 Tools unter Windows

So einfach die Beschaffung der Entwicklungsumgebung unter OS X ist, so schwierig ist sie unter Windows. Da sich Objective C vornehmlich unter den Systemen von Apple durchgesetzt hat, werden die gängigen Entwicklungsumgebungen entsprechend auch nur für den Mac angeboten. Unter Windows bleibt dem Entwickler nichts anderes übrig als sich die Entwicklungsumgebung selbst zu bauen.

[2] http://developer.apple.com/

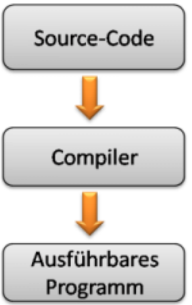


Abbildung 9.2: Der Weg vom Code zum Programm

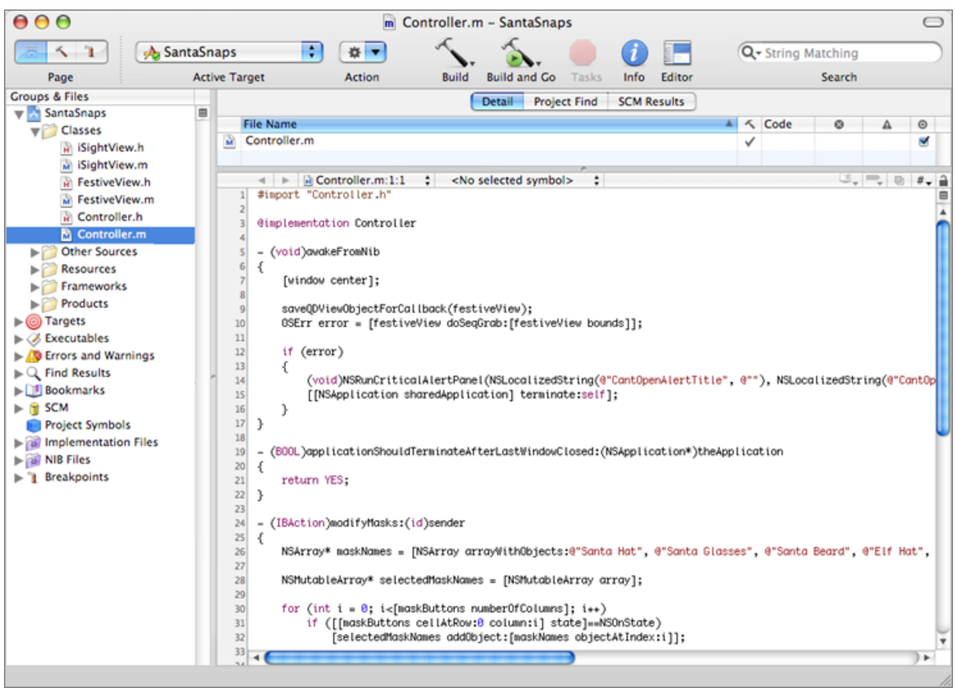


Abbildung 9.3: Die Entwicklungsumgebung unter Mac OS X

GNUStep

Bei GNUStep handelt es sich im Grunde um das Cocoa Framework. Der Name ist eine Anspielung auf NEXTStep, das Betriebssystem der Firma Next, welches als Vorgänger von Mac OS X gilt. GNUStep enthält neben dem Cocoa-Framework auch einen gcc-Compiler und bietet damit, in Verbindung mit einem beliebigen Texteditor, theoretisch alle nötigen Werkzeuge um Objective C Programme zu schreiben. Die Installation von GNU-Step auf einem Windows-Rechner gestaltet sich allerdings etwas aufwändig. Da eine detailierte Installationsanleitung, eigentlich einen eigenen Artikel wert wäre, und daher deutlich den Rahmen dieser Einführung sprengen würde, sei zu diesem Zweck auf die offiziellen GNUStep-Seiten[3] im Web verwiesen.

9.4 DasCocoaFramework

An verschiedenen Stellen in diesem Artikel ist bereits vom Cocoa-Framework gesprochen worden. Der geneigte Leser hat also vermutlich bereits erkannt, dass dieses Framework für Objective c wohl in irgendeiner Form wichtig sein muss. Dass dies tatsächlich so ist, und was es sonst noch so mit diesem ominösen Framework auf sich hat wollen wir daher in diesem Abschnitt kurz erklären. Dabei werden wir das Framework keines Wegs bis ins kleinste Detail beleuchten, sondern uns statt dessen darauf beschränken, einen generellen Überblick darüber zu erhalten in wie weit Cocoa für Objective C von Bedeutung ist.

9.4.1 Mac OS X

Um zu begreifen welche Funktion Cocoa hat sollte man es sich am besten im Gesamtzusammenhang mit Mac OS X ansehen. Abbildung 9.4 Zeigt den schematischen Aufbau Eines Mac OS X Systems.

Aqua		
Cocoa	Carbon	Java
Quartz	OpenGL	QuickTime
Darwin		

Abbildung 9.4: Das Schichtmodell von Mac OS X
Entnommen Lit. [reflit:1]

Als unterste Schicht sehen wir hier *Darwin*, den Mac-Systemkern. Darüber sind verschiedene APIs für Multimedia und Graphikverarbeitung angesiedelt. Auf diesen wiederrum bauen die drei Frameworks Cocoa, Java und Carbon auf. Carbon ist hierbei ein Framework, dass speziell dazu dient, Software, die unter älteren Versionen von Mac OS X entwickelt wurde, zu integrieren. Java stellt

[3] http://www.gnustep.org/

das Java-Runtime-Enviernment zu Verfügung, dass es ermöglicht, in Java geschriebene Programme auszuführen. Cocoa... ach ja, darum geht es ja in diesem Abschnitt.

Cocoa hat im Grunde zwei Gesichter. Zum einen bildet es eine Schnittstelle zwischen *Aqua*, der Benutzerschnittstelle von Mac OS X, und den Systemfunktionen und APIs der unteren Schichten. Zum Anderen ist es aus Entwicklersicht im Wesentlichen eine Sammlung von Bibliotheken, für die verschiedensten Aktionen. Es enthält alles von Benutzer Ein- und Ausgaben über graphische Oberflächen bis hin zu Multimediaanwendungen und Netzwerkkommunikation. Und genau diese bibliotheken sind es die Objective C verwendet.

Genau genommen besteht das Cocoa-Framework aus zwei, naja wir nennen es mal Packeten. Und zwar sind das:

- Foundation
- Application Kit

Das Foundation Packet enthält im Wesentlichen alle Grundlegenden Bibliotheken, die man zur Erstellung von konsolenbasierten Programmen benötigt. Hier befinden sich unter anderem Bibliotheken für Ein- und Ausgabeverwaltung, Dateioperationen und Speichermanagement. Will man allerdings nicht nur auf der Konsole arbeiten, sondern darüber hinaus auch noch graphische Benutzeroberflächen programmieren, so benötigt man zusätzlich die Bibliotheken des Application Kits. Hier ist von Buttons über Fenster bis hin zu Menüs alles vordefiniert was das Herz begehrt.

Wir sehen also, dass das Cocoa-Framework die Basis von Objective C ist. Ohne die zahlreichen Bibliotheken von Cocoa könnte man Objective C so gut wie nichts anfangen.

9.5 Sprachliche Elemente von Objective C

Nach der bisher doch sehr theoretischen Einführung in die Sprache Objective C wollen wir uns in diesem Abschnitt nun endlich mit ihren sprachlichen Elementen, sprich der Syntax, befassen. Hierbei werden wir feststellen, dass Objective C im Grunde eine Zweischneidige Syntax besitzt. Zum einen wäre da eine starke Tendenz zur Programmiersprache C, von der Objective C abgeleitet ist, zum anderen erkennt man bei Objektoperationen ganz klar die Syntax von SmallTalk. Aber lassen Sie uns das genauer in Augenschein nehmen.

9.5.1 Alte Freunde

Jeder Programmierer, der in seiner Laufbahn schon einmal mit den Sprachen Java, C oder C++ in Kontakt gekommen ist, wird in Objective C eine Menge alter Freunde wieder treffen. So bietet die Sprache alle Standard-Sprachkomponenten in derselben Form an, wie dies auch die anderen C basierten Sprachen tun. Für all jene Leser die mit C basierten Sprachen bisher noch keinen Kontakt hatten, soll an dieser Stelle noch einmal kurz auf das Wesentliche eingegangen werden.

Verzweigungen Die Verzweigung, also die bedingte Auswertung von bestimmten Codesequenzen, wird in Objective C mit dem Schlüsselwort `if` eingeleitet.

Dieses wird von einer Bedingung, welche in runden Klammern steht, gefolgt. Der bedingt auszuführende Code, steht anschließend in geschweiften Klammern. Das Folgende Beispiel soll dies noch einmal verdeutlichen.

```
if(x==2){
   y=3;
}
```

Sofern die Variable x hier den wert zwei hat, soll der Variable y der Wert 3 zugewiesen werden. An dieser Stelle lassen sich nebenbei auch sehr anschaulich der Vergleichs und der Zuweisungsoperator unterscheiden. Ein einfaches Gleichheitszeichen ist eine Zuweisung, das bedeutet, dass der Variable links der Wert, welcher sich rechts vom Gleichheitszeichen befindet zugewiesen wird. Mit einem doppelten Gleichheitszeichen prüft man auf Gleichhiet der beiden Werte. Das Ergebnis einer sollten Vergleichsoperation ist entweder *wahr* (engl. true) oder *falsch* (engl. false).

Fallunterscheidungen Bei einer Vielzahl von möglichen Fällen ist die oben dargestellte Verzweigung eine eher müßige Struktur, die auch schnell unübersichtlich wird. Dies führt gerade bei unerfahrenen Programmierern schnell einmal zu Fehlern, die dann auch nur sehr schwer zu finden sind. Die Fallunterscheidung kann hier Abhilfe schaffen. Man hat hier die Möglichkeit eine bestimme Variable vom Typ *int*, dies ist eine stets eine Ganzzahl, auf ihren Wert hin zu überprüfen. Je nachdem welchen Wert die Variable gerade hat, kann nun eine entsprechende Reaktion erfolgen. Hierbei sind in der Anzahl der zu unterschiedenen Fälle von Seiten der Sprache keine Grenzen gesetzt. Das folgende Codebespiel soll dies noch einmal kurz verdeutlichen.

```
switch(x){
   case 1:
      y = 2;
      break;
   case 2:
      y = 4:
      break;
   default:
      y = 0;
      break;
}
```

Mit `switch(x)` geben wir an welche Variable wir überprüfen wollen. Danach folgt in geschweiften Klammern die Auswertung, oder genauer die verschiedenen Fälle (engl. case) die wir überprüfen wollen. So bedeutet `case 1:`, dass nun die Befehle folgen, welche ausgeführt werden sollen, wenn x, also die Variable die wir betrachten, den Wert 1 hat. In unserem Fall soll dann der Variable y der Wert 2 zugewiesen werden. Das Schlüsselwort `break;` bewirkt, dass die Fallunterscheidung nach dem Ausführen des bedingten Codes verlassen wird. Das heißt, dass wenn eine Bedingung zutrifft, und in ihrem Codeblock, der Befehl `break;` auftaucht, alle anderen Fälle nicht mehr überprüft werden.

Abweisende Schleifen Hiermit wird eine Schleife benannt, welche zu Beginn ihrer Ausführung eine Bedingung überprüft. Nur wenn diese Bedingung bereits vor der ersten Ausführung des Schleifenblocks erfüllt ist, so wird der Code-Block der Schleife überhaupt ausgeführt. Auch hierfür ist die Syntax direkt aus C übernommen, wie folgendes Beispiel zeigt.

```
while(x < 10){
   x++;
}
```

Hierbei wird die Variable x so lange um eins erhöht, bis sie 10 oder größer ist.

Nicht abweisende Schleifen Analog zu den oben vorgestellten abweisenden Schleifen wollen, wir nun die *nicht abweisenden Schleifen* definieren als solche, bei denen die Bedingung erst am Ende der Ausführung überprüft wird. Dies bedeutet, dass zunächst der gesamte Schleifenblock einmal ausgeführt wird, bevor zum ersten mal geprüft wird ob die Bedingung der Schleife erfüllt ist. Auch hierbei wurde auf die C-Syntax zurückgegriffen.

```
do{
   x++;
}while(x < 10);
```

Hierbei wird die Variable x auf jeden Fall einmal um eins erhöht. Erst dann wird geprüft ob ihr Wert noch kleiner ist als 10. Nur wenn das der Fall ist, wird die Schleife fortgesetzt.

For-Schleifen Auch die sehr nützliche for-Schleife aus C ist in Objective C integriert worden. Bei einer for-Schleife handelt es sich im Grunde genommen um eine kompaktere Version der oben bereits vorgestellten abweisenden Schleife. Ihren syntaxtischen Aufbau wollen wir uns an einem kleinen Beispiel betrachten.

```
for(i = 0; i < 10; i++){
   x = x + y;
}
```

Nach dem Schlüsselwort `for` stehen in runden Klammern drei Terme. Der erste ist eine Initialisierung. Hier kann wie in unserem Beispiel ein Schleifenzähler initialisiert werden. Der zweite Term bildet die Schleifenbedingung. Die Schleife wird so lange ausgeführt, bis diese Bedingung nicht mehr zutrifft. Im dritten Term wird festgelegt, was jeweils am Ende der Schleifenausführung getan werden soll. In unserem Beispiel wird hier der Schleifenzähler erhöht. Dies ist wie bereits erwähnt nur eine andere, kompaktere, Schreibweise für eine abweisende Schleife. Man könnte den Sachverhalt auch folgendermaßen darstellen:

```
i = 0;
while(i < 10){
   x = x + y;
   i++;
}
```

Diese beiden Schleifen haben exakt die selbe Funktion, die for-Schleife ist lediglich etwas kompakter.

9.5.2 Klassen

Da es sich bei Objective C um eine objektorientierte Sprache handelt, stehen Klassen natürlich im Mittelpunkt des Interesses. In diesem Abschnitt wollen wir uns daher einmal ansehen wie man eine Klasse in Objective C definiert. Auf die Grundlagen der Objektorientierung werden wir in diesem Zusammenhang nicht mehr weiter eingehen, da dieser Artikel sich mit Objective C befasst und eine eingehende Beschreibung der Grundlagen objektorientierter Programmierung dessen Rahmen daher leicht sprengen würde. Interessierte Leser sein daher auf [??] verwiesen.

Die Klassendefinition

Die Definition einer Klasse besteht in Objective C immer aus zwei Dateien. Einer so genannten Header-Datei (Dateiname endet auf .h), in welcher zunächst die grobe Struktur der Klasse definiert wird. Und einer Implementierungs-Datei (Dateiname endet auf .m), in welcher man die konkrete Funktionalität realisiert. Betrachten wir zunächst einmal den Header. Die Erste Zeile eines solchen Header sollte stets folgendermaßen lauten.

```
#import <Foundation/Foundation.h>
```

Hiermit wird die Funktionalität des in Abschnitt 9.4 vorgestellten Cocoa-Framworks für Konsolenanwendungen eingebunden. Diese Funktionalität wird in Objective C für so gut wie alles benötigt, und sollte daher immer mit eingebunden werden. Will man in der Klasse darüber hinaus auch noch graphische Elemente, wie z.B. Buttons oder Fenster, verwenden so kann man zusätzlich noch das Application Kit einbinden. Dies geschieht mit dem Befehl:

```
#import <AppKit/AppKit.h>
```

Nach den Import-Anweisungen beginnt die eigentliche Deklaration der neuen Klassen mit dem Schlüsselwort `@interface` gefolgt vom Namen der Klasse. Anschließend werden die Instanzvariablen in geschweiften Klammern definiert, dann folgen, nach den Geschweiften Klammern, die Instanz- und Klassenmethoden. Beendet wird die Deklaration der Klasse mit dem Schlüsselwort `@end`. Lassen Sie uns diese verbale und unansehnliche Beschreibung nun einmal etwas strukturierter betrachten indem wir einen Blick auf die allgemeine Syntax werfen.

```
#import <Foundation/Foundation.h>

@interface Klassenname{
   Datentyp _instanzVariable;
}
   + (Rückgabetyp)klassenMethode:(Parametertyp)parameterName;
   - (Rückgabetyp)instanzMethode:(Parametertyp)parameterName;
@end
```

In dieser Darstellung erkennen wir sofort einige Besonderheiten in der Syntax von Objective C. So hat es sich beispielsweise eingebürgert, dass man den Namen einer Instanzvariable mit einem Unterstrich beginnt. Zwar wird dies vom Compiler nicht zwingend verlangt, und es stellt in keinster Weise einen syntaktischen

Fehler dar dies nicht zu tun, es ist allerdings gängiger Standard und wird daher eindringlich empfohlen. Wesentlicch spannender ist da schon die Deklaration einer Methode.

```
+ (Rückgabetyp)klassenMethode:(Parametertyp)parameterName;
- (Rückgabetyp)instanzMethode:(Parametertyp)parameterName;
```

Plus bzw. Minus vor der Methode beschreiben hierbei nicht, wie man leicht annehmen könnte die Sichtbarkeit einer Methode, sondern ordnen sie als Klassen bzw. Instanzmethode ein. In geschweiften Klammern folgt dann der Rückgabetyp der Methode, hierbei kann es sich sowohl um native Datentypen wie *int* und *void* oder auch um selbst definierte Klassen handeln. Der Methodenname sollte laut Konvention mit einem kleinen Buchstaben beginnen und dann in alternierender Schreibweise fortgesetzt werden, d. H. jedes neue Wort wird mit einem großen Buchstaben begonnen (z.B. *gehNachHause* oder *lassMichInRuhe*). Nach dem Methodennamen folgt die Liste der Parameter, wobei jeder Parameter jeweils mit einem Doppelpunkt beginnt. Hinter diesem Doppelpunkt steht dann in runden Klammern der Datentyp des Parameters, dahinter sein Name.

Nachdem diese *Deklaration* in der Header Datei abgeschlossen ist, kann nun damit fortgefahren werden der Klasse Leben einzuhauchen, sprich sie mit der gewünschten Funktionalität zu versehen. Lassen Sie uns auch hierzu einen Blick auf die allgemeine Syntax der Implementierungsdatei werfen.

```
#import "Klassenname.h"

@implementation Klassenname

   +(Rückgabetyp)klassenMethode:(Parametertyp)parameterName{
      //Anweisungsblock
   }

   -(Rückgabetyp)instanzMethode:(Parametertyp)parameterName{
      //Anweisungsblock
   }

@end
```

Wie man sieht benötigen wir auch hier zunächst einmal einen import-Befehl. Bevor wir damit beginnen können die Methoden mit Funktionalität zu füllen müssen wir zunächst einmal den oben definierten Header importieren. Hieran sehen wir auch deutlich, dass ein Import von eigenen Headern sich von dem von Systemeigenen Headern unterscheidet. Eigene Header-Files werden beim importieren in Anführungszeichen geschrieben, native Klassen in spitzen Klammern. Damit wird dem Compiler gezeigt wo er die entsprechenden Dateien zu suchen hat.

Nachdem wir unsere Klassendeklaration eingebunden haben können wir mittels des Schlüsselwortes `@implementation`, gefolgt vom Klassennamen, damit beginnen die Funktionalität der Methoden zu realisieren. Hierzu kopiert man im Grunde noch einmal die Methoden-Deklaration und fügt dann in geschweiften Klammern den Anweisungsblock hinzu den die Methode realisieren soll.

Duch das Schlüsselwort `@end` wird dann auch die Implementierung der Klasse abgeschlossen.

Vererbung

Eines der wichtigsten Werkzeuge der objektorientieren Programmierung ist die Vererbung. Hierbei geht es darum, eine bereits bestehende Klasse um gewisse Funktionalitäten zu erweitern. Man tut dies indem man eine neue Klasse, die wir im folgenden als Unterklasse bezeichnen wollen, definiert und ihr quasi die Funktionalität und Instanzvariablen der Oberklasse vererbt. So Kann man z.B. eine Oberklasse *Kraftfahrzeug* schreiben und von dieser dann die zwei Unterklassen *PKW* und *LKW* ableiten, welche jeweils eigene, erweiterte, Funktionalitäten mit sich bringen.

Syntaktisch ist die Vererbung in Objective C ein Kinderspiel. Alles was zu tun ist um eine Klasse von einer anderen erben zu lassen ist, die Deklaration der Unterklasse (vgl. Abschnitt 9.5.2) wie folgt zu erweitern.

```
#import <Foundation/Foundation.h>
#import "Oberklasse.h"

@interface Klassenname : Oberklasse{

...

@end
```

Überschreiben und Erweitern von Methoden Die Vererbung bietet neben der Möglichkeit einer Klasse zusätzliche Methoden hinzuzufügen auch noch die Option eine Bestehende Methode zu erweitern oder gänzlich zu überschreiben. Will man in Objective C eine Methode der Oberklasse überschreiben, so muss man einfach nur in der Implementierungsdatei eine neue Implementierung der Methode vornehmen. Wird jetzt ein Objekt der Unterklasse erzeugt, und die überschriebene Methode wird aufgerufen, so wird die Version verwendet, welche in der Unterklasse implementiert ist.

Oft ist es allerdings sinnvoll eine Methode nicht gänzlich zu ersetzten. Es kann z.B. sein, dass man in der Oberklasse eine Methode bereits implementiert hat und in der Unterklasse im Grunde die selbe Methode braucht, die nur noch etwas mehr tut. Schreibt man, um bei unserem Beispiel von oben zu bleiben, eine Klasse *Kraftfahrzeug* mit einer Methode *schreibeTechnischeDaten*, welche die technischen Daten des Kraftfahrzeugs ausgibt, so ist es unter Umständen sinnvoll in der Unterklasse *LKW* zusätzlich zu Daten wie Kraftstoffverbrauch und Motorleistung auch noch die Anzahl der Achsen und das zulässige Ladungsgewicht auszugeben. Dies sind Angaben die wiederrum bei der Unterklasse *PKW* nur bedingt sinnvoll wären.

Glücklicherweise bietet uns Objective C nicht nur die Möglichkeit Methoden stur zu überschreiben. Wir können sie auch einfach um die benötigte Funktionalität erweitern. Dies geschieht, indem wir zu Beginn der neuen Implementierung einer Methode die folgende Zeile schreiben.

```
[super methodenName:parameter];
```

Auf die hier dargestellte besondere Syntax werden wir im Abschnitt 9.5.2 noch näher eingehen. Soviel sei aber schon hier verraten. Die Zusätzliche Codezeile tut nichts anderes als die entsprechende Methode der Oberklasse aufzurufen. Bevor wir unsere neue Funktionalität ausführen, rufen wir also noch einmal die alte auf, und sparen uns somit die Arbeit, die identische Funktionalität noch einmal zu zu implementieren.

NSObject - die Mutter aller Klassen Alle Klassen die Objective C mitgeliefert werden, sind, über unterschiedlich viele Generationen, von der Klasse NSObject abgeleitet. Dies ist eine Klasse, die für sich genommen keine wirkliche Funktion hat, allerdings einige grundlegende Methoden und Eigenschaften für die Speicherverwaltung von Objective C mit sich bringt. Daher sollte auch jede Klasse die man selbst schreibt stets von NSObject abgeleitet sein. Mehr zu den Gründen hierfür wird im Abschnitt 9.6.1 behandelt.

Abstrakte Klassen

Ein weiteres wichtiges Mittel der Objektorientierten Programmierung ist die Möglichkeit, sogenannte *abstrakte Klassen* definieren zu können. Bei einer abstrakten Klasse handelt es sich hierbei um eine Klasse, welche mindestens eine *abstrakte Methode* besitzt, d.h. eine Methode, die zwar im Header deklariert ist, für diese Klasse allerdings nicht implementiert wird. Die Idee hierbei ist, die Implementierung auf eine Unterklasse auszulagern, und somit sicher zu gehen, dass diese Unterklasse die entsprechende Methode implementieren muss.

Was Objective C angeht, so heißt es in der offiziellen Dokumentation der Sprache (vlg. [**??**]) zum Thema abstrakte Klassen:

> A Class taht's defined solely so that other Classes can inherit form it. Programs don't use instances of an abstract class, only of its subclasses.

Hierbei handelt es sich allerdings nur um eine reine Konvention. Sprachliche Mittel mit deren Hilfe abstrakte Klassen realisiert werden könnten bieten Objective C leider nicht an. Definiert man in Objective C eine Klasse mit einer abstrakten Methode, so wird man vom Compiler nicht gezwungen deren Implementierung in einer Unterklasse nach zu hohlen. Sobald eine Methode im Header-File deklariert wurde, kann sie auch aufgerufen werden, egal ob sie implementiert wurde oder nicht. Das Ergebnis, dieser Vorgehensweise ist ein Programmabsturz. Man vertraut hier also im Grunde auf die Fähigkeiten der Entwickler, sich untereinander abzusprechen um derartige Fehler zu vermeiden.

Ein Workaround, der sicherstellt, dass eine abstrakte Methode nicht zum Absturz des Programmes führt, wenn diese aufgerufen wird ohne Implementier zu sein, ist die Methode eine *Exception* werfen zu lassen. Man deklariert, die Methode also in der Oberklasse und Implementiert sie dort in der Art, dass sie nichts anderes tut als eine Exception zu werfen. Dies könnte wie folgt aussehen:

```
-(Rückgabetyp) methodenName{
   [NSException raise:@"Abstrakte Methode aufgerufen"
   format:@"Es wurde die abstrakte Methode -methodenName der Klasse
   Klasse aufgerufen."];
}
```

Zwar stellt auch dies keine finale Lösung des Problems dar, allerdings wird so wenigstens der Programmabsturz verhindert.

Objektoperationen

Nachdem wir uns in den vorhergehenden Unterkapiteln nun eingehend damit beschäftigt haben, wie man Klassen definiert, und welche Besonderheiten es dabei zu beachten gilt. Wollen wir uns nun mit der Erzeugung und Verwendung von Objekten befassen. Wir werden hierbei die Frage Klären wie man ein Objekt erzeugt, wie man seine Methoden aufruft und wie man es wieder vernichtet.

Methodenaufruf Das erste was wir uns zum Umgang mit Objekten ansehen wollen, ist der aufruf einer Methode, da dies die häufigste Aktion ist, die man in der Praxis mit Objekten durchführt und wir dies für alle späteren Beispiele benötigen. Die generelle Syntax hierfür lautet:

```
[objektName␣methodenName:parameterName1␣:parameterName2];
```

Der gesamte Aufruf steht in eckigen Klammern. Man beginnt mit dem Namen des Objektes für welches man eine Methode aufrufen will, und fährt nach einem Leerzeichen mit dem Namen dieser Methode fort. Danach können noch, jeweils beginnend mit einem Doppelpunkt, die Parameter für die Methode übergeben werden. Lassen Sie uns hierzu noch einige konkrete Beispiele betrachten:

```
// Aufruf mit einem Parameter
[auto fahre:links];
// Aufruf ohne Parameter
[bombe explodiere];
// Aufruf mit mehreren Parametern
[bank eroeffneKonto:nachName :vorName];
```

Objekterzeugung Zur Erzeugung eines Objektes verwendet man in Objective C eine spezielle Klassenmethode, die man *convenience Allocator* nennt. Dies ist eine Klassenmethode, welche einen Pointer auf ein Objekt der Klasse zurückgibt. In der Deklaration sieht ein solcher convenience Allocator wie folgt aus:

```
+ (KlassenName *)klassenNameWithParameter:(ParameterTyp)parameterName;
```

Die hier verwendete Namensgebung ist dabei zwar nicht verpflichtend, empfiehlt sich allerdings um den Überblick zu behalten. So beginnt der Name des convenience Allocators immer mit dem Namen der Klasse, gefolgt von den Namen der Parameter die ihm übergeben werden.

Darüber hinaus benötigt man für jede Klasse auch einen *Initializer* der die Objekte initialisiert. Hierbei handelt es sich um eine Instanzmethode, die i.d.R. direkt nach der Erzeugung des Objektes vom convenience Allocator aufgerufen wird. In der Deklaration sieht dies folgendermaßen aus:

```
- (id) initKlassenNameWithParameter:(ParameterTyp)parameterName;
```

Was es mit dem Rückgabetyp *id* auf sich hat wird im Abschnitt 9.6.1 genauer erklärt. Ansonsten sieht die Deklaration der Methode der des convenience Allocators sehr ähnlich, es wurde lediglich ein *init* vor den Methodennamen geschrieben. Auch wenn man mehrere convenience Allocators deklariert sollte man stets zu jedem auch eine InitMethode schreiben. Darüber hinaus sollte man auf jeden Fall eine einfache init-Methode ohne Parameter schreiben.

Wesentlich spannender als die bloße Deklaration dieser Methoden ist ihre Implementierung. Beginnen wir hierbei mit dem Initializer in einer schematischen Darstellung.

```
- (id) initKlassenNameWithParameter:(ParameterTyp)parameterName{
   // Zunächst Initialisierung der Superklasse ausführen
   self = [super init];

   // Bei Erfolg...
   if (self){
      // ...eigene Eigenschaften setzten
      [self setInstanzVariable:paramterName]
   }
   return self;
}
```

Das Objekt `self`, dass hier angesprochen wird, bezeichnet in Objective C stets das aktuelle Objekt. `super` bezeichnet die Oberklasse von der die Klasse des Objekts abgleitet ist. Diese Zeile kann in Objective C eigentlich immer so geschrieben werden, da jede Klasse, von NSObject einmal abgesehen, eine Oberklasse hat. Hieran können wir auch sehen, warum es so wichtig ist, immer eine einfache init-Methode ohne Parameter zu schreiben, da eine etwaige Unterklasse diese stets aufrufen wird um sich selbst zu initialisieren. Holen wir dies also nach.

```
- (id) init{
   self = [super init];
   return self;
}
```

Nachdem wir nun auch sicher Gestellt haben, dass man von unserer fiktiven Klassen gefahrlos erben kann, wollen wir uns noch anschauen wie man einen convenience Allocator implementiert.

```
+ (KlassenName *)klassenNameWithParameter:(ParameterTyp)parameterName{
   return [ [ [ self alloc ] initWithParameter:parameterName ] autorelease ];
}
```

Wie wir sehen besteht der Rumpf der Methode aus nur einer einzigen Zeile, aber die hat es in sich. Die Auswertung erfolgt von innen nach außen. Zunächst wird die alloc-Methode des self-Objekts aufgerufen. Dies ist eine Methode, welche die Klasse von NSObject erbt. Hier findet die eigentliche Objekterzeugung statt, indem Speicher für das Objekt reserviert wird. Im nächsten Schritt wird die init-Methode aufgerufen. In diesem Beispiel wird der Vollständigkeit halber diejenige Verwendet die Parameter erwartet und die Instanzvariablen setzt, doch das ist nicht zwingend erforderlich die Methode `init` hätte es genauso getan. Der dritte

schritt der hier gezeigt wird, ist optional. Hier wird eine Methode `autorelease` aufgerufen, die ebenfalls zum NSObject gehört. Was es damit genau auf sich hat soll in Abschnitt 9.6.1 etwas genauer beäugt werden. An dieser Stelle sei nur so viel gesagt, dass dieser Methodenaufruf dafür sorgt, dass der Speicher für unser Objekt wieder frei gegeben wird, sobald wir es nicht mehr benötigen.

9.6 Besonderheiten der Sprache

Nun da wir uns mit den generellen Aspekten von Objective C als objektorientierte Sprache beschäftigt haben, wollen wir uns einige Besonderheiten der Sprache etwas genauer anschauen. Hierbei begutachten wir neben einigen speziellen sprachlichen Komponenten auch das sehr innovative System der Speicherverwaltung von Objective C.

9.6.1 Speicherverwaltung

Beim Lesen dieses Artikels wurden sie schon an verschiedenen Stellen auf diesen Abschnitt verwiesen. Dies war nötig, da die Speicherverwaltung von Cocoa, denn die Speicherverwaltung von Objective C baut komplett auf Cocoa auf, ein recht umfangreiches und vor allem auch wichtiges Thema ist, dass nicht nur so zwischendurch am Rande erklärt werden kann.

Als erstes sollte hierbei gesagt werden, dass diese Speicherverwaltung vollständig dynamisch ist. Dies erkennt man bereits daran, dass Objective C es uns nicht erlaubt, Objekte statisch zu erzeugen. Die folgende Codezeile, zur Erzeugung eines String-Objekts würde einen Compiler-Fehler verursachen:

```
NSString aString;
```

Wir erhalten hier den Fehler

> error: statical allocated instance of Objective-C class 'NSString'

da wir versucht haben statisch eine Instanz der NSString-Klasse zu erzeugen. Tatsächlich müssten wir hier schreiben:

```
NSString* aString;
```

What's your number?

Das Symbol „*“ bedeutet in Objective C sozusagen „ID“. Wir deklarieren also nicht wirklich ein Objekt, sondern definieren eine ID für ein Objekt. Man kann sich diese ID wie eine Art Personalausweisnummer vorstellen, die uns hilft ein Objekt zu identifizieren. Dabei ist die ID selbst kein Objekt (so wie eine Personalausweinummer ja auch kein Mensch ist).

> In Wahrheit handelt es sich bei ID um einen C-Zeiger auf eine Objekt-Beschreibungsstruktur: typedef strct Objc_object *id. Die Identifikationen sind also nichts anderes als C-Zeiger. In Objective C interessiert uns das jedoch von seltenen Ausnahmen abgesehen nicht wirklich. (vgl. Lit. [**??**])

Mit dem oben genannten Aufruf, haben wir nun ein Objekt, bzw. eine ID für ein Objekt, deklariert. D.h. wir haben bekannt gegeben, dass wir ein solchen Objekt erzeugen und nutzen wollen. Erzeugt haben wir es allerdings noch nicht, denn dafür benötigen wir Speicherplatz, den wir noch nicht haben. Daher wollen wir uns nun einmal im Detail ansehen wie genau ein Objekt Erzeugt wird, und was dabei speicherseitig passiert. Um Speicher zu bekommen müssen wir folgendes schreiben:

```
aString = [NSString alloc];
```

Diese Zeile erzeugt nun eigentlich unser Objekt, indem es der oben deklarierten ID einen Speicherbereich zuweist. Dass ist auch genau dass, was bisher unsere convenience Allocator getan haben. Damit haben wir ein Objekt erzeugt. Doch verwenden können wir es immer noch nicht, denn es wurde noch nicht initialisiert. Also ist dies unser nächster Schritt:

```
aString = [aString init];
```

Nachdem wir nun auch dies getan haben, ist unser String-Objekt fertig erzeugt, und initialisiert. Es kann also verwendet werden. Die beiden Letzten Schritte, die wir ausgeführt haben, hätte man allerdings auch noch etwas kompakter schreiben können. Lassen Sie uns diese beiden Methodenaufrufe ineinander verschachteln um den Code etwas übersichtlicher zu halten:

```
aString = [[NSString alloc] init];
```

Dem so erzeugten String können wir nun Werte zuweisen, ihn auf dem Bildschirm ausgeben, seine Zeichen umsortieren usw.. Dann, wenn wir ihn nicht mehr benötigen müssen wir ihn wieder löschen. Genauer gesagt, müssen wir den Speicher, den wir für den String alloclert haben, wieder frei geben. Das Schlüsselwort hierfür lautet in Objective C `release`.

```
[aString release];
```

Damit haben wir den Speicher wieder frei gegeben und alles ist gut. Dies ist tatsächlich der optimalste Fall. Objekt erzeugen, Objekt benutzten, Objekt vernichten. Leider ist es nicht immer so einfach. Das Problem das sich einstellt ist, dass man in großen Projekten nicht unbedingt immer weiß, wann ein Objekt ausgedient hat. So ist es durchaus möglich, dass es an einer Stelle im Programm zwar nicht mehr benötigt wird, an einer anderen Stelle aber schon.

Stellen Sie sich eine Mitarbeiterverwaltung vor, in der Ein Mitarbeiter die Abteilung wechselt. Er wird in der Personalabteilung nicht mehr benötigt, dafür aber in der IT-Abteilung. Und in der Gehaltsabrechnung sollte er auch weiterhin auftauchen. Hier wäre es fatal, wenn er gelöscht würde, nachdem er in der Personalabteilung nicht mehr gebraucht wird.

Wir benötigen also eine etwas andere Herangehensweise. Glücklicherweise stellt Cocoa uns eine zur Verfügung. Denn Cocoa zählt für uns an wie vielen Stellen im Programm unser Objekt gerade verwendet wird. Wenn wir für ein Objekt die Methode `alloc` aufrufen, dann reserviert Cocoa nicht nur Speicherplatz sonder Initialisiert auch einen Zähler mit 1. Diesen Zähler nennt man Retain-Counter. Die Methode `release` tut tatsächlich nichts anderes als diesen

Zähler wieder zu senken. Sobald der Zähler wieder auf 0 ist, gibt Cocoa den reservierten Speicherplatz wieder frei.

Wir müssen also nicht andres tun, als jedes mal wenn wir die ID eines Objektes Speichers, diesen Zähler zu erhöhen. Und jedes mal wenn wir an einer Stelle im Programm mit dem Objekt fertig sind, den Zähler zu verringern. Den Befehl zum verringern kennen wir schon, das ist `release`, der Befehl um, den Zähler für ein Objekt zu erhöhen, lautet `retain`.

Luxus mit Pools

Diese Arbeitsschritte können wir uns sogar noch etwas vereinfachen. Stellen Sie sich einmal folgendes vor. Sie erzeugen am Anfang eines Hauptprogramms viele Objekte und benutzen diese. Jetzt müssen Sie natürlich daran denken, alle diese Objekte wieder frei zu geben, wenn sie diese nicht mehr benötigen. In der Praxis geht das oft schief, da man es schlicht und ergreifend vergisst. Die Lösung für dieses Problem heißt NSAutoreleasePool. Die Objekte dieser Klasse kann man sich im Grunde wie kleine Müllmänner vorstellen, die alles aufsammeln, was so liegengeblieben ist. Man muss sich einfach nur zu Beginn des Hauptprogramms ein solches Objekt erstellen, und dieses am Ende wieder frei geben.

Damit unser Müllmann aber auch weiß, was er einsammeln soll, müssen wir noch eine kleinigkeit machen. Und zwar müssen wir die Objekte bei der Erzeugung für den NSAutoreleasPool „markieren“. Wir tun das durch den Aufruf der Methode `autorelease`. Wenn wir in unserem Beispiel von oben bleiben, so bedeutet dies, dass wir

```
aString = [[NSString alloc] init];
```

zu

```
aString = [[[NSString alloc] init] autorelease];
```

ändern. Indem wir dies, direkt bei der Erzeugung unseres Objektes tun, sorgen wir dafür, dass wir uns keine Gedanken mehr um die Freigabe dieses Objektes machen müssen. Das einzige Objekt, dass wir jetzt am Ende unseres Hauptprogrammes noch freigegeben müssen ist das NSAutreleasePool-Objekt.

9.6.2 Typlose Objekte

Lassen Sie uns nun, da wir auch mit der Speicherverwaltung von Objective C vertraut sind, einen Blick auf eine weitere Spezialität der Sprache werfen. Bereits im vorherigen Abschnitt haben wir gesehen, dass in Objective C Objekte mittels so genannter IDs angesprochen werden. Tatsächlich kennt Objective C sogar einen „Datentyp“ *id*. Wenn wir eine Variable vom Typ id deklarieren, so haben eine Variable in die Objekte von jedem beliebigen Typ packen können. Denn unser „id-Objekt“ ist typlos. Nun stellt sich natürlich die berechtigte Frage, wozu so etwas dienen soll. Die Antwort auf diese Frage erschließt sich uns wahrscheinlich am besten, wenn wir uns ein kleines Beispiel ansehen.

In unserem Beispiel haben wir zwei Programmierer. Der eine befindet sich in Deutschland, der andere in Australien. Nun möchte der Programmierer in Australien, mit einem Objekt namens `Band`, gerne ein Objekt namens `Guitar` des Programmierers aus Deutschland nutzen. Und zwar will er für dieses Objekt

eine Methode `seiteGerissen` aufrufen. Das Problem ist nun, dass der Australier nicht weiß, von welcher Klasse dieses Objekt ist. Natürlich könnte er jetzt einfach den deutschen Programmierer anrufen und Fragen, aber der würde, auf Grund der Zeitverschiebung, wahrscheinlich gerade schlafen und wenig Begeisterung zeigen. Eine mögliche Lösung ist, dass der deutsche Programmierer sein Objekt als vom Typ id deklariert. Wenn er das tut, dann kann der Australier ohne Schwierigkeit, jede beliebige Methode für dieses Objekt aufrufen. Erst zur Laufzeit wird dann geprüft ob das Objekt diese Methode auch wirklich kennt. Dies nennt man *spätes Binden*. An spätestens dieser Stelle sollte jetzt jedem erfahrenen Programmierer ein kalter Schauer über den Rücken laufen. Denn das Objekt `Guitar`, könnte ja alles mögliche sein und Niemand kann garantieren, dass es die Methode `seiteGrissen` kennt. Ein Programmabsturz ist hier wortwörtlich vorprogrammiert. Doch auch für dieses Problem bietet uns Objective C eine Lösung. Sie lautet Protokolle.

Protokolle

Ein Protokoll ist in Objective C im Grunde eine Sammlung von Methoden-Deklarationen, welche nicht an eine bestimmte Klasse gebunden ist. Einer Klasse wiederum kann beigebracht werden ein Protokoll zu verstehen. Die Deklaration für ein solches Protokoll sieht folgender Maßen aus:

```
@protocol ProtokollName
   - (Rückgabetyp)methodenName1;
   - (Rückgabetyp)methodenName2:(Parametertyp)parameterName;
   ...
@end
```

Nach der Definition eines solchen Protokolls, kann man einer Klasse beibringen dieses zu verstehen, indem man die Klassendeklaration wie folgt ändert:

```
...
@interface KlassenName : OberKlasse <ProtokollName>{
...
```

Dann müssen die Methoden des Protokolls nur noch in der Klassen-Implementierung realisiert werden. Wenn man jetzt ein id-Objekt deklariert, kann man die Einschränkung verlangen, dass dieses Variable nur solche Objekte aufnehmen kann, welche das entsprechende Protokoll verstehen.

```
id<ProtokollName> variablenName;
```

Um wieder zu unserm Beispiel zurück zu kommen, würde dies bedeuten, dass sich die beiden Programmierer auf ein Protokoll einigen müssten. Dieses Protokoll könnte dann wie folgt aussehen:

```
@protocol Zerschmettern
   - (bool)seiteGerissen;
   - (void)aufDemBodenZerschmettern;
@end
```

Das Objekt `Guitar` müsste dann folgendermaßen angelegt werden.

```
...
```

```
id<Zerschmettern> Guitar;
...
```

Nachdem dies erfolgt ist, kann der Programmierer der Band-Klasse ohne Probleme folgenden Code schreiben:

```
...
   if([guitar seiteGerissen]){
      [guitar aufDemBodenZerschmettern];
   }
...
```

Zwar weiß der Programmierer in Australien jetzt immer noch nicht von welcher Klasse `Guitar` ist, es kann ihm aber auch egal sein, denn alles was er wissen muss, ist, dass `Guitar` die Methoden, die er aufrufen will, kennt.

9.6.3 Vergleich mit anderen Sprachen

Um diesen Abschnitt abzuschließen, soll an dieser Stelle noch einmal ein kompakter Vergleich von Objective C mit anderen, vergleichbaren, Sprachen hergestellt werden.

Kriterium	**Objective C**	**C++**	**Java**
Binding Time	spät	spät	früh
Operatoren überschreiben	nein	ja	nein
Garbage Collection	nein	nein	ja
Abstrakte Klassen	nein	ja	ja
Mehrfachvererbung	nein	ja	nein
Interfaces / Protokolle	ja	nein	ja
Kategorien	ja	nein	nein
Konstruktor	nein	ja	ja
Destrukttor	nein	ja	nein
Referenzen	nein	ja	ja

Tabelle 9.1: Eigenschafften verschiedener Programmiersprachen
Entnommen Lit. [**??**]

Wenn man diese Tabelle betrachtet, dann sieht es zunächst so aus, dass Objective C nicht sonderlich viel zu bieten hätte. Ich will hier allerdings ausdrücklich darauf hinweisen, dass einige Dinge für die hier ein „nein“ eingetragen wurde, weil die Sprache sie nativ nicht enthält, durchaus leicht nachgebaut werden können. So haben wir z.B. gesehen, dass der Garbage Collector, der hier vermisst wird, durch einen NSAutoreleasePool durchaus ersetzt werden kann. Abstrakte Klassen werden sprachlich zwar nicht direkt angeboten, sind, durch einen kleinen Workaround, aber durchaus verfügbar.

9.7 Beispielapplikation

Im bisherigen Verlauf dieses Artikels habe ich mich stets bemüht die Code-Beispiele möglichst klein und allgemeingültig zu halten. Das Ziel dieser Vorge-

hensweise bestand darin, Ihnen die Möglichkeit zu bieten den Code von Objective C in der reinsten nur möglichen Form zu betrachten, die noch nicht durch Sonderfälle eines konkreten Beispiels verzerrt ist. Da man auf diese Weise zwar sehr schön die generelle Syntax, nicht aber die praktische Programmierung lernen kann, soll in diesem Abschnitt ein vollständiges Programmbeispiel entstehen. Aus naheliegenden Gründen wird auch dieses Beispiel selbstverständlich nicht durch überwältigende Komplexität glänzen, jedoch durchaus die Umsetzung der gängigsten Techniken der objektorientierten Programmierung mittels Objective C verdeutlichen.

9.7.1 Unser "kleines"Beispiel

Wir wollen uns in diesem Beispiel ein kleines Programm schreiben, welches verschiedene Instrumente verwalten kann. Hierbei soll es eine abstrakte Klasse *Instrument* geben, von der wir zwei Subklassen, eine *Gitarre* und ein *Piano*, ableiten wollen. Abbildung 9.6 zeigt die Klassenstruktur unseres Programmes in Form eines UML-Diagramms.

Das Instrument soll der Einfachheit halber nur drei Instanzvariablen haben, einen Namen, einen Preis und ein Alter. Für diese drei Variablen werden wir dann entsprechende Accesoren schreiben, welche die Variablen auslesen oder auf bestimmte Werte setzten. Des weiteren wird Instrument noch eine abstrakte Methode, die das Instrument altern lassen soll, einen Initializer und einen convenience Allocator haben.

Die Gitarre wird vom Instrument erben und keine eigenen Instanzvariablen haben. Somit müssen wir hier nur einen neuen convenience Allocator schreiben.

Dem Piano, wollen wir zusätzlich zu geerbten Instanzvariablen noch eine weitere Hinzufügen, die angeben soll, wie viele Tasten das Piano hat. Daher müssen wir hier, neben dem convenience Allocator, noch die beiden Accsesoren für diese Variable schreiben.

Lassen sie uns mit der Deklarationsdatei der Instrumentenklasse beginnen.

```
// Datei: Instrument.h

#import <Foundation/Foundation.h>

//Wir erben wie immer von NSObject
@interface Instrument : NSObject{

   // Instanzvariablen:
   NSString * _name;      // Name des Instruments
   int _age;              // Alter des Instruments
   int _price;            // Preis des Instruments
}
   // Accessoren:
   -(NSString*)name;
   -(void)setName:(NSString*)name;
```

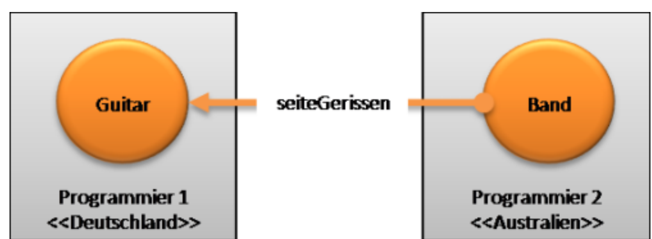


Abbildung 9.5: Ein Alltägliches Problem

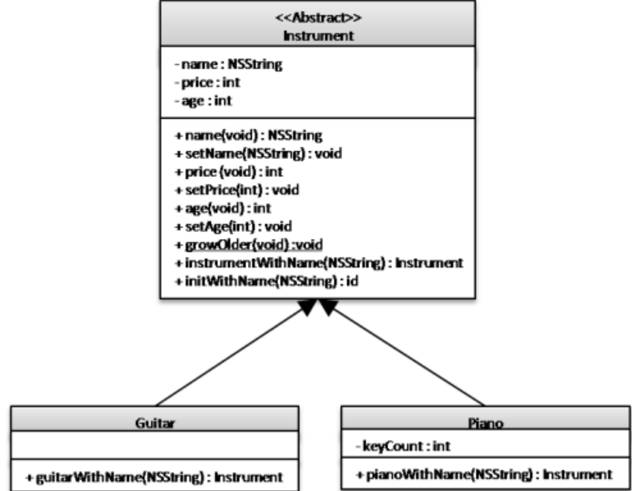


Abbildung 9.6: Klassendiagramm zur Beispielapplikation

```
    -(int)age;
    -(void)setAge:(int)age;

    -(int)price;
    -(void)setPrice:(int)price;

    // Diese Methode soll das Instrument altern lassen.
    -(void)growOlder;

    // Objekterzeugung:
    +(Instrument*)instrumentWithName:(NSString*)name;
    -(id)initWithName:(NSString*)name;

@end
```

Damit wäre die Deklaration unseres Instrumentes bereis abgeschlossen. Nun wollen wir uns an die Implementierung der Methoden machen. Dabei gehen wir hier, aus Gründen der Übersichtlichkeit, in derselben Reihenfolge vor, wie wir sie oben deklariert haben. Dies ist allerdings reine Willkür, es gibt keine Verpflichtung die Methoden in dieser Reihenfolge zu implementieren.

```
// Datei: Instrument.m

#import "Instrument.h"

@implementation Instrument

    // Liest die Variable _name aus
    -(NSString*)name{
        return _name;
    }
    // Weißt der Variable _name einen neuen Wert zu
    -(void)setName:(NSString*)name{
        // Wenn der neu Wert der zugewiesen werden soll nicht gleich dem alten ist...
        if( _name != name){
            // ...gib den Speicher des alten Werts frei...
            [_name release];
            // ...und reserviere Speicher für den neuen bevor du ihn zuweist.
            _name = [name retain];
        }
    }
    // Liest die Variable _age aus
    -(int)age{
        return _age;
    }
    // Weißt der Variable _age einen neuen Wert zu
    -(void)setAge:(int)age{
        _age = age;
    }
    // Liest die Variable _price aus
```

```
    -(int)price{
        return _price;
    }
    // Weist der Variable _price einen neuen Wert zu
    -(void)setPrice:(int)price{
        _price = price;
    }
    // Abstrakte Klasse, die das Instrument altern lassen soll.
    // Die Methode muss von Unterklassen implementiert werden.
    -(void) growOlder{
        /*
         * Wenn jemand diese Methode aufruft ohne, dass sie Implementiert wurde,
         * wird eine Exception geworfen!
         */
        [NSException raise:@"Abstrakte Methode aufgerufen"
        format:@"Es wurde die abstrakte Methode -growOlder der Klasse
        Instrument aufgerufen."];
    }
    // Convenience Allocator, der zur Objekterstellung benoetigt wird.
    +(Instrument*)instrumentWithName:(NSString*)name{
        return [[[self alloc] initWithName:name] autorelease];
    }
    // Initializer, welcher zur Initialisierung der Objekte benoetigt wird.
    -(id)initWithName:(NSString*)name{
        self = [super init];

        if(self){
            [self setName:name];
            [self setPrice:0];
            [self setAge:0];
        }
        return self;
    }

@end
```

Damit haben wir eine fertige Instrumentenklasse definiert. Wie Sie sehen, habe ich bei der abstrakten Methode `growOlder` eine Exception eingebaut, die verhindern soll, dass unser Programm abstürtzt, wenn sie fälschlicher weise für ein Instanzobjekt der Klasse Instrument selbst aufgerufen wird.

Lassen Sie uns nun mit der Deklaration der Gitarren-Klasse fortfahren.

```
// Datei: Guitar.h

#import<Foundation/Foundation.h>
#import "Instrument.h"

// Gitarre ist ein Instrument
```

```
@interface Guitar : Instrument{

}
   // Objekterzeugung:
   +(Guitar *) guitarWithName:(NSString*)name;
@end
```

Wie erwartet, ist diese Deklaration nicht sonderlich Spektakulär. Da die Klasse keine eigenen Instanzvariablen hat muss hier nur klargestellt werden, dass Sie von Instrument abgeleitet ist, und einen convenience Allocator benötigt. Sehen wir uns also die dazugehörende Implementierung an.

```
// Datei: Guitar.m

#import "Guitar.h"

@implementation Guitar

   /*
    * Implementierung der Methode growOlder, welche in der Klasse Instrument
    * als abstrakte Methode definiert wurde.
    */
   -(void) growOlder{
      // Erhoehe das Alter der Gitarre um 1.
      [self setAge:[self age] + 1];

      // Senke den Preis der Gitarre um 10 %
      [self setPrice:[self price] * 0.9];
   }

   // Convenience Allocator, der zur Objekterstellung benoetigt wird.
   +(Guitar *) guitarWithName:(NSString*)name{
      return[[[self alloc] initWithName:name] autorelease];
   }

@end
```

Auch diese Implementierung enthält nicht sehr viel Spannende Elemente. Neben dem coneninece Allocator implementieren nur noch die abstrakte Methode der Instrumenten-Klasse aus. Etwas interessanter wird da schon die Deklaration der Piano Klasse die wie folgt aussieht.

```
// Datei: Piano.h

#import <Foundation/Foundation.h>
#import "Instrument.h"

// Ein Piano ist eine Instrument
@interface Piano : Instrument{
```

```
    // Zusätzliche Instanzvariable
    int _keyCount;                // Anzahl der Tasten des Pianos
}
    // Accsessoren für die neue Instanzvariable
    -(int)keyCount;
    -(void)setKeyCount:(int)keyCount;

    // Objekterzeugung:
    +(Piano*)pianoWithName:(NSString*)name;

@end
```

Wie versprochen haben wir dem Piano, neben drei Instanzvariablen der Oberklasse, noch eine weitere Variable spendiert. Hierfür deklarieren wir auch wieder die entsprechenden Accessoren um die Variable auslesen und auf bestimmte Werte setzten zu können. In der Implementierung sieht die Klasse, dann folgendermaßen aus:

```
// Datei: Piano.m

#import "Piano.h"

@implementation Piano
    // Ließt die Variable _keyCount aus
    -(int)keyCount{
        return _keyCount;
    }

    // Weißt der Variable _keyCount einen neuen Wert zu
    -(void)setKeyCount:(int)keyCount{
        _keyCount = keyCount;
    }

    /*
     * Implementierung der Methode growOlder, welche in der Klasse Instrument
     * als abstrakte Methode definiert wurde.
     */
    -(void)growOlder{
        // Erhoehe das Alter der Gitarre um 1.
        [self setAge:[self age] + 1];

        // Senke den Preis der Gitarre um 15 %
        [self setPrice:[self price] * 0.85];
    }

    // Convenience Allocator, der zur Objekterstellung benoetigt wird.
    +(Piano*)pianoWithName:(NSString*)name{
        return [[[self alloc]initWithName:name] autorelease];
    }
```

```
@end
```

Nun da wir alle Klassen so wie wir sie geplant hatten implementiert haben, wollen wir uns daran machen ein kleines Hauptprogramm zu schreiben. Im Verlauf dieses Hauptprogramms, soll jeweils eine Gitarre und ein Piano erstellt werden, welche wir dann mit ein paar Werten initialisieren und schließlich altern lassen wollen.

```
// Datei: main.m

#import <Foundation/Foundation.h>
#import "Instrument.h"
#import "Guitar.h"
#import "Piano.h"

int main(int argc, const char * argv[]){
   // Gitarre und Piano deklarieren
   Guitar * aGuitar;
   Piano * aPiano;

   NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

   // Hier wird das Gitarrenobjekt erzeugt
   aGuitar = [Guitar guitarWithName:@"Eine Gitarre"];
   NSLog(@"Wir haben: %@", [aGuitar name]);

   // Der Preis der Gitarre wird gesetzt
   [aGuitar setPrice:100];

   // Hier wird das Pianoobjekt erzeugt
   aPiano = [Piano pianoWithName:@"Ein Piano"];
   NSLog(@"Wir haben: %@",[aPiano name]);

   // Der Preis des Pianos wird gesetzt
   [aPiano setPrice:900];

   // Die Anzahl der Tasten des Pianos werden festgelegt
   [aPiano setKeyCount:24];

   // Hier werden zur Kontrolle die aktuellen Eigenschaften der Instrumente ausgegeben
   NSLog(@"Gitarre Preis: %@, Alter: %@",[aGuitar price],[aGuitar age]);
   NSLog(@"Piano Preis: %@, Alter: %@",[aPiano price],[aPiano age]);

   // Jetzt lassen wir die Instrumente altern
   [aGuitar growOlder];
   [aPiano growOlder];

   // Eine erneute Ausgabe zum zu sehen, ob die Alterung erfolgreich war
```

```
    NSLog(@"Gitarre Preis: %@, Alter: %@",[aGuitar price],[aGuitar age]);
    NSLog(@"Piano Preis: %@, Alter: %@",[aPiano price],[aPiano age]);

    [pool release];
    return 0;
}
```

Wenn wir dieses Programm starten, so erhalten wir die folgenden Bildschirmausgabe:

```
Maggo@Laptop>$ Instrument.exe

    Wir haben: Eine Gitarre
    Wir haben: Ein Piano
    Gitarre Preis: 100, Alter: 0
    Piano Preis: 900, Alter: 0
    Gitarre Preis: 90, Alter: 1
    Piano Preis: 765, Alter: 1

Maggo@Laptop>$
```

Hieran können wir erkenn, dass unser Programm wie erwartet funktioniert hat. Nachdem wir die Objekte initialisiert hatten ließen wir sie altern. Hierbei hat das Gitarren-Objekt, völlig korrekt, einen Wertverlust von 10 % erlitten, während das Pinao-Objekt einen Wertverlust von 15 % hinnehmen musste.

Zugegebenermaßen ist dieses Beispiel nicht gerade sehr komplex. Doch es war schließlich auch nicht das Ziel Sie mit einem hochkomplexen Beispiel möglichst stark zu verwirren, sondern Ihnen in kompakter Form die sprachlichen Möglichkeiten von Objective C näher zu bringen.

Kapitel 10

Erlang (Arend Kühle)

10.1 Wie Erlang entstand

1981 richtete die Schwedische Telekommunikationsgesellschaft Erricson das Computer Science Laboratory (CSLab) ein, mit dem Ziel neue Architekturen, Konzepte und Strukturen zur zukünftigen Systementwicklung zu erarbeiten. Dort untersuchte man 20 logische und funktionale sowie nebenläufige Programmiersprachen auf ihre Fähigkeiten in der Telekommunikation um eine zu finden, die man zukünftig zur Entwicklung von Ericsson Produkten verwenden sollte. Man kam zum Schluss, dass es eine Symbolische Hochsprache sein sollte, wie Lisp, Prolog und Parlog. Weitere Experimente mit diesen Sprachen haben gezeigt, dass Lisp und Prolog nicht die in der Telekommunikation benötigte Nebenläufigkeit und Fehlertoleranz erreichen und mit Parlog war man nicht in der Lage einen asynchronen Telefonprozess in nur einem Prozess der Sprache zu erstellen. Eine neue Programmiersprache musste entwickelt werden, mit den gewünschten Eigenschaften der drei Endkandidaten und vor allem ein hohen Grad an Nebenläufigkeit und Fehlertoleranz. Unter der Leitung von Bjarne Däcker, Chef des CSLab, wurde bis 1987 an einer bis dahin geheimen Sprache gearbeitet, die nebenläufige Prozesse zu Prolog hinzufügen sollte. Es wurde zum ersten Mal von Erlang gesprochen.

Der Name Erlang ist keine Zusammensetzung aus den Worten Ericsson und Language, wie man zuerst denken könnte, sonder wurde zu Ehren des dänischen Mathematikers und Ingenieurs Agner Krarup Erlang (1878 - 1929) gewählt. Er war Anfangs des letzten Jahrhunderts der Erste, der sich mit Warteschlangenproblemen in der Telefonie beschäftigt hat. Seine Theorien wurden von vielen Telfongesellschaften aufgegriffen und die Hilfsmaßeinheit für Verkehr in Kommunikationsnetzen, Erlang wurde nach ihm benannt. Noch heute ist die Erlang-Verteilung eine oft verwendetet Methode um die statistische Durchschnittslänge von Telefonverbindungen in Call-Centern zu errechnen.

Zum Ende des Jahres 1987 gab eine in Prolog implementierte erste Version von Erlang, die sich der Prolog Operatoren bediente und noch keine eigene Syntax hatte. 1988 begann das erste wichtige Projekt mit Erlang. Eine Gruppe von Ericsson Ingenieuren wählten die Sprache um damit einen Prototypen des so

genannten ACS/Dunder, eine Architektur für eine Telefonanlage (MD110), zu implementieren. Sie verglichen den zeitlichen Programmieraufwand Typischer Telekommunikationsroutinen in Erlang dem bisher verwendetet PLEX. Damit schaffte Erlang nicht nur den Schritt aus dem Labor, sondern gewann aus diesem Projekt viele Ideen, die die Sprache schließlich formten. 1988 wurde klar, dass Erlang bereit war zum programmieren von Telekommunikationssystem eingesetzt zu werden. Als 1989 ca. 10% der Funktionalität der MD110 in Erlang vorlagen stellte sich heraus, dass die Effizienz von Erlang zehn mal so hoch war wie die von PLEX. Um Erlang für kommerzielle Produkte nutzen zu können musste eine Performance Verbesserung von 70% erreicht werden. Um das Nachzuweisen entwickelte Joe Armstrong JAM, Joe's abstract maschine, indem er die WAM (Warren's abstract maschine, eine Virtuelle Maschine für Prolog) um parallele Prozesse, Nachrichtenübergabe und Fehlererkennung erweiterte. Die erste Implementierung der JAM erfolgte in Prolog, was sehr ineffizient war, da es sehr langsam arbeitete. Es erlaubte aber die Virtuelle Maschine zu testen und zu evaluieren, was es ermöglichte einen Erlang Compiler in der Sprache Erlang zu schreiben. Daraufhin versuchte Armstrong eine Virtuelle Maschine in C zu implementieren, was von Mike Williams schließlich umgesetzt wurde. Zudem arbeitete Robert Virding an den Erlang Libraries. 1991 war die JAM ausgereift und Erlang hatte seine eigene Syntax. Erlang war nun nicht mehr nur ein Dialekt von Prolog, sondern eine eigene Programmiersprache.

1992 wurde bei Ericsson Buesiness Systems die Entscheidung gefällt ein Produkt zu entwickeln, dass auf dem in Erlang implementierten ACS/Dunder basierte. Dieses Produkt und damit die Sprache Erlang etablierten sich auf dem Internationalen Switching Symposium in Japan, woraufhin Ericsson 1993 ein Tochterunternehmen namens Erlang Systems AB gründete. Diese Tochter sollte Erlang an externe Kunden vermarkten und verkaufen, Schulungen und Consulting anbieten, was die erste kommerzielle Version von Erlang zur Folge hatte, die auf heterogenen Systemen lief. Der große Durchbruch der Sprache begann 1995 mit dem Scheitern eines groß angelegten Projekts, der next generation switch-AXE-N Telefonanlage. Um dieses Vorhaben nicht nach 7 Jahren Entwicklung aufgeben zu müssen entschied man sich für einen neuen Versuch unter Verwendung von Erlang. Das Ergebnis war 1998 die AXD301, ein ATM Switch, der mit seinen 1,7 Millionen Zeilen Erlangcode eine Zuverlässigkeit von 99,9999999% (Neun 9er) erreicht. Das entspricht etwa 31 Millisekunden Systemstörung im Jahr, bei 30-40 Millionen Verbindungen in der Woche. Das machte Ericsson nach der Markteinführung dieser Anlage zum Weltmarktführer im Bereich Softswitching. Während dieser Zeit wurden die Erlang Librariers weiterentwickelt und in OTP (The Open Telecom Platform) umbenannt, um den Erlang Anwendern eine stabile Plattform zu geben. Dennoch wurde Anfang 1998 die Sprache aus der Entwicklung verbannt, da Ericsson lieber mit weit verbreiteten Sprachen, wie C++ entwickeln wollte, als eine neue Programmiersprache durchzusetzen. Ende des Jahres wurden dann Erlang und OTP als open source veröffentlicht. Schlagartig verließen 15 Entwickler das Ericsson CSLab und gründeten eine eigene Firma, Bluetail AB, um mit Erlang zuverlässigere Internet Dienste zu entwickeln. Der Trend setzte sich fort und Erlang fand viele neue Anhänger. Schließlich hob Ericsson den Bann und nahm die Sprache wieder in die Entwicklung auf. Viele weitere Produkte unter Verwendung von Erlang machten sich einen Namen, wie z. B. der Alteon SSL Acclerator von Nortel Networks (Nortel schluckte Bluetail) oder GPRS (Ericsson), was zu 76% in Erlang ge-

schrieben wurde. Seit 2001 ist die Erlang/OTP Technologie gut etabliert und findet immer breitere Anwendung, wenn es darum geht hochverfügbare Netzwerkdienste und Steuerungssysteme zu entwickeln.

10.2 Die Entwicklungsumgebung

Erlang benötigt wie Java eine Virtuelle Maschine um Programme in Bytecode zu übersetzen und auszuführen. Man spricht bei diesen VM auch von Erlang-Knoten. Es gibt zwei VM die für Erlang eingesetzt werden, JAM und BEAM. JAM ist eine Weiterentwicklung von WAM und wurde in der frühen Entwicklung von Erlang eingesetzt. Damals versuchte man außerdem noch Erlang in C zu übersetzen, um es dann in Maschinensprache zu kompilieren, hat dabei aber gemerkt, dass es im Vergleich zur VM nur bei kleinen Programmen schneller ist. Heute benutzen fast alle BEAM, was für Bogdan/Björn's Erlang Abstract Machine steht und auch die unterstützte VM der kommerziellen Version von Erlang ist.

Die Virtuelle Maschine BEAM steht auf www.erlang.org in steht's aktuellen Versionen seit 2003 kostenlos zum Download zur Verfügung und ist mit umfangreicher Dokumentation und Bibliotheken zu haben. BEAM gibt es für nahezu alle gängigen Betriebssysteme wie Solaris, Linux, FreeBSD, MacOS und Windows.

Ist die Virtuelle Maschine für Erlang BEAM installiert, kann man den Erlang Interpreter mit dem Befehl `erl` starten. Es begrüßt einen dann die Erlang Shell (EShell) und wartet auf Eingaben. Die Besonderheit dieser Shell ist die Nummerierung des Prompts. Da ein Befehl in Erlang wird immer mit einem Punkt (.) abgeschlossen wird, kann man mehrere Zeilen Code als einen Befehl eingeben.

Zum Schreiben von Erlang Programmen kann man einen beliebeigen Editor nehmen, wobei sich Emacs empfiehlt, da Erlang ein für diesen Editor ein Plugin bereithält. Mit diesem *Erlang Mode for Emacs* sind Syntax Highlighting und viele andere nützliche Funktionen vorhanden, was die Programmierarbeit erleichtert.

Um das Rad der Erlang Programmierung nicht neu erfinden zu müssen hat man mit OTP (Open Telecom Platform) eine große Sammlung von Bibliotheken, generischen Komponenten und Möglichkeiten der Anbindung von C, C++ und Java-Anwendungen. Zu dem auch verschiedene fertig implementierte Komponenten, die man zur Entwicklung seiner Erlang Programme nutzen kann. Diese Komponenten verstehen das Nachrichtensystem von Erlang und lassen sich so gut einbinden. Inets Beispielsweise ist ein Modul, dass HTTP Server und Client sowie FTP Client Funktionalität bereitstellt. Oder Mnesia, ein verteiltes Echtzeit Datenbank Management System, die komplexe Datenstrukturen verwalten und speichern kann.

Für die Entwicklung mit einer sehr umfangreichen Bibliothekensammlung ist CEAN (Comprehensive Erlang Archive Network) eine gute Lösung. Dabei handelt es sich um eine Softwareplattform, die Zugang zu aktuellen Erlangprogrammen für verschieden Systeme gibt.

10.3 Der Code

Erlangs Sprache ist sehr kompakt und recht leicht zu erlernen. Mit Erfahrung in Java, C, oder Pascal etwa braucht man vielleicht eine Woche, bis man in der Lage ist damit kleinere Programme zu schreiben und einen Monat bis man reif ist sich auch an Größeres zu wagen. Erfahrung in ähnlichen Sprachen wie zum Beispiel, wie Prolog, Haskell, oder Lisp ermöglicht es schnell in Erlang fit zu sein. Es kann dennoch eine Weile dauern kann, bis man auch die Fehlertoleranz und Nebenläufigkeit dieser Sprache verinnerlicht hat und nutzen kann.

10.3.1 Variablen und Konstanten

Das Erste was bei den Variablen auffällt ist, dass sie nicht variabel sind! Eine Variable ist nur solange variabel, wie ihr noch kein Wert zugewiesen wurde. Hat sie erst einmal einen Wert ist dieser fest an sie gebunden und kann nicht mehr verändert werden. Man kann es sich vorstellen wie eine Gleichung in der Mathematik. Da kann man zum Beispiel sagen `X = 3`, `X = X` und `3 = 3`, niemand würde auf die Idee kommen zu sagen `X = X + 1`. Da man seine Variable `X` nicht mehr verändern kann erstellt man einfach eine weitere Variable `X1` um einen neuen Wert zu speichern. In Variablen können nicht nur einzelne Werte oder Datenstrukturen gespeichert werden, sondern auch Funktionen und Ausdrücke. Sie beginnen immer mit einem Großbuchstaben.

Konstanten sind in Erlang global und ohne Makrodefinition oder Include Dateien für alle Programmteile bekannt. Sie beginnen mit einem kleinen Buchstabe gefolgt von einer Reihe von alphanumerischen Zeichen und den Sonderzeichen _ und `@`. Sie sehen beispielsweise so aus:

`januar, akueh001@fhw, hans_wurst`

Zudem kann man Konstanten auch mit Apostrophen definieren, um die gesamte Zeichenpaltette zu nutzen wie

`'Montag' '*' 'Konstante mir Leerzeichen'`.

An der Stelle sei angemerkt, dass man sich um das allokieren von Speicher keine Sorgen machen muss, da Erlang das automatisch macht und einen Garbage Collector besitzt, der auch automatisch nicht mehr benötigten Speicher wieder frei gibt.

10.3.2 Datentypen

Zahlenwerte werden sehr dynamisch gehandhabt. Gibt man eine Zahl ohne Kommerstelle an, so erhält man einen Integer Wert. Teilt man diesen, oder besteht auf Nachkommastellen so erhält man einen Fließkommawert. Man muss sich auch keine Gedanken um Überlauffehler machen, im Gegenteil man könnte seine Freunde damit beeindrucken, in Erlang mit riesigen Zahlen zu rechnen.

Eine weitere Rolle spielen Zahlen wenn es um String, also Zeichenketten geht. Ein String in Erlang ist nichts weiter als eine Liste aus Integern[1]. Man kann also ein String anlegen indem man etwas zwischen Anführungszeichen setzt, oder eben eine Liste mit den Codes der einzelnen Zeichen erstellt.

[1] Diese Zahlen entsprechen dem Latin-1 Code (ISO-8859-1)

Tuples

Um Daten strukturiert verwalten zu können kann man Tuples definieren, das entspricht etwa dem Struct in C. Will man nun beispielsweise einen Punkt `P` definieren sieht das so aus: `P = {3, 7}.` Tuples können auch verschachtelt werden, wie in diesem Beispiel mit einem Tuple für eine Gerade `G`, die in dem eben erstellten Punkt `P` enden soll: `G = {{4, 10}, P}.` Um jetzt Werte aus einem Tuple lesen zu können geht man danach, an welcher Stelle diese Werte im Tuple stehen. Die `X`-Werte beider Punkte in der eben erstellen Geraden kann damit auslesen, dass man die Struktur des Tuples mit neuen Variablen (`X1` und `X2`) und _ (Unterstriche) an Stellen die nicht intressieren nachbaut und den Namen (`G`) angibt. `{{X1, _}{X2, _}} = G.` Daraus erhält man zwei neue Variablen, `X1` mit dem festen Wert 4 und `X2` mit dem festen Wert 3. In der Praxis werden Tuples meist mit einem konstanten Namen begonnen, damit man es später besser zuordnen kann. Das Tuple `G` aus diesem Beispiel würde dann so aussehen: `G = {gerade, {punkt, 4, 10}, {punkt, 3, 7}}.`

Listen

Will man nun mehrere Dinge speichern, kann man sie in einer Liste definieren. Das Erstellen einer Liste in Erlang sieht etwa so aus wie das Erstellen eines Array in C. Alles was in die Liste soll wird zwischen eckigen Klammern gepackt und mit Komma getrennt. Da Variablen in Erlang nicht variabel sind kann man auch eine Liste nicht mehr verändern, wenn sie erst erstellt ist. Dennoch kann man schnell mit diesen Listen arbeiten, wenn man das Prinzip von Kopf und Schwanz verinnerlicht und sich rekursiven Funktionen bedient. Beispiel: Wir erstellen eine kleine Liste von Zahlen. `Liste = [2, 4, 7, 10, 3, 8].` In dieser Liste ist 2 der Kopf und die restliche Liste ab dem Wert 4 der Schwanz. Um nun damit arbeiten zu können kann man diese Form nehmen: `[Kopf | Schwanz].` Die Notation gibt Zugriff auf das erste Element und einer Liste mit dem Rest, dem *Schwanz*. Das erste Element zu holen ist einfach, man erstellt eine neue Variable und da man sich nicht um Schwanz kümmert kommt da ein *Dont Care*-Unterstrich in die Notation: `[Erstes | _ ] = Liste.` Die Liste hat sich nicht verändert, lediglich eine neue Variable namens Erstes mit dem Wert 2 wurde kreiert. Will man nun an das dritte Element, muss man Schrittweise der Liste die Glieder abtrennen: `[ _ | Schwanz ] = Liste.`
`[ _ | SchwanzVomSchwanz ] = Schwanz.`
`[ X | _ ] = SchwanzVomSchwanz.` Jetzt haben wir den Wert des dritten Elements in `X` gespeichert und merken, dass man das doch besser mit einer Schleife hätte machen können. Da Erlang eine funktionale Sprache ist, haben wir keine Schleifen, sondern Funktionen die solche Probleme für uns lösen sollen.

10.3.3 Funktionen

Funs

Die einfachste und schnellste Möglichkeit Funktionalität in ein Erlang Programm zu bringen sind so genannte Funs. Das sind anonyme Funktionen, die man Variablen zuweisen kann. Eine Fun, die zwei Zahlen summieren soll, kann folgender Maßen aussehen: `Sum = fun(A, B) -> A*B end.` Zur Erklärung, das

Wort `fun` leitet die Funktion ein, die Werte in der Klammer werden als Eingabe erwartet und nach dem Pfeil beginnt die Berechnung. Nun ist die Fun in der Variable Sum gespeichert und kann verwendet werden indem man diese Variable mit zwei Werten aufruft: `Sum(3, 5).` Dieser Aufruf liefert nun das Ergebnis 15 zurück. Funs können in Erlang als Argumente an Funktionen übergeben werden und Funktionen können auch Funs zurückgeben. Solche Funktionen nennt man dann Funktionen höherer Ordnung.

Module

Module sind der Hauptbestandteil des Codes in Erlang, denn in ihnen befinden sich die Funktionen, die wir schreiben. Module haben die Dateinamensendung .erl und müssen kompiliert werden um angewendet werden zu können. Ein kompiliertes Modul hat dann die Dateinamensendung .beam.

Hier ein Beispiel für ein Modul das Funktionen für mathematische Berechnungen enthält. Es nennt sich math und exportiert[2] die Funktionen pi und fac, welche 0 oder 1 Parameter erwarten.

```
1  -module(math).
2  -export([pi/0, fac/1]).
3
4  pi()                  ->  3,14159265358979323846.
5
6  fac(N) when N > 0     ->  N * fac(N-1);
7  fac(0)                ->  1.
```

Benötigt man Pi, so kann man aus dem Modul `math` die Funktion `pi` aufrufen: `math:pi()` Benötigt man die Fakultät von 3 sieht das so aus: `math:fac(3).` Dabei lässt sich erkennen, das die Berechnung der Fakultät rekursiv stattfindet. Der übergebene Werte wird jeweils mit dem Ergebnis aus der Fakultät der nächst niedrigeren Zahl multipliziert. Um die Rekursion zu stoppen, ist die Funktion `fac` mehrfach implementiert. Einmal für alle Zahlen die Größer sind als 0 und eine eben für die 0. Denn ist die 0 erreicht ist die Berechnung zu ende und das Ergebnis kann zurückgegeben werden. Hier kommt das Pattern Maching ins Spiel. Wird die Funktion `fac` aufgerufen, entscheidet es sich anhand der übergebenen Werte, welche `fac`-Funktion den Wert berechnen wird. Werden mehr oder weniger Werte als nur einen an die Funktion übergeben, so wird kein Pattern gefunden das passt und es kommt zu einer finsteren Fehlermeldung. Bei einem übergebenen Wert, so wie hier gewollt, wird dann von oben nach unten durchprobiert und die erste Funktion genommen die passt. Wird kein passendes Pattern gefunden, wie beispielsweise bei einer negativen Zahl oder einem nicht numerischen Wert, kommt es ebenfalls zu Fehler. Weiter einschränken kann man die Wahl der passenden Funktion durch Guards. Sie geben die Funktion nur frei, wenn man ihre Bedingung erfüllt. In diesem Beispiel findet man einen Guard, der prüft, ob der übergebene Wert größer ist als 0: `when N > 0`

Die Rekursion, das Pattern Maching und die Guards ermöglichen es, bekannte Elemente der sequenziellen Programmierung, wie Schleifen, Fallunterscheidungen und Wertzuweisungen durch die Funktionen in einem Modul zu ersetzen.

[2]Nur exportierte Funktionen können von außerhalb der Funktion aufgerufen werden

10.3.4 Hello, World!

Bei der Vorstellung von Code darf natürlich das "Hello, world!" nicht fehlen!

```
1 -module(say).
2 -export([hello/0]).
3
4 hello()   -> io:fwrite("Hello, world!\n").
```

10.4 Ausführungsmodell

Erlang ist eine nebenläufige, funktionale und verteilte Programmiersprache die gezielt für Anwendungen in der Telekommunikation entwickelt wurde. Hauptsächlich wird Erlang für Netzwerkdienste, Webservices, Überwachungs- und Steuerungssysteme mit den Schwerpunkten auf Parallelität, hoher Verfügbarkeit, Fehlertoleranz und das Wechseln von Modulen zur Laufzeit eingesetzt.

10.4.1 Funktional

Erlang ist eine funktionale Sprache, das bedeutet das anders als bei den weit verbreiteten imperativen Sprachen wie C oder Java nicht sequenziell programmiert wird sondern die Programmiertechnik auf das Aufrufen von Funktionen beruht. Die Programme bestehen nur aus hintereinander geschalteten oder rekursiven Aufrufen von Funktionen einfacher und höherer Ordnung.

10.4.2 Nebenläufigkeit

"Die Welt ist parallel. Wenn wir Programme schreiben wollen, die sich wie Dinge in der realen Welt verhalten, brauchen diese Programme eine nebenläufige Struktur. Benutze eine Sprache, die für nebenläufige Programme gestaltet wurde und die Entwicklung wird viel einfacher. Erlang Programme modellieren, wie wir denken und interagieren."Joe Armstrong, geistiger Vater von Erlang

Nebenläufigkeit ist Teil der menschlichen Natur. Das Gehirn reagiert extrem schnell auf Reize, da bewusstes denken zu langsam wäre. In der Zeit in der sich der Gedanke "Bremsen!"bildet, hat man schon längst die Bremse betätigt. Man verfolgt geistig viele Verkehrsteilnehmer ohne bewusst darüber nachzudenken. Wenn man das nicht könnte wäre man nicht in der Lage ein Auto zu fahren. Dies geschieht nebenläufig. Wir benutzen hauptsächlich sequenzielle Programmiersprachen, mit denen es unnötig schwer ist Programme für eine parallele Welt zu schreiben. Dazu braucht es eine nebenläufige Programmiersprache die so denkt und interagiert wie wir. Wir teilen uns keine Gedanken, Erinnerungen oder Ideen, jeder hat seine Eigenen. Um deine Gedanken zu beeinflussen sende ich dir eine Nachricht, indem ich zu dir rede, oder dir etwas zeige. Du hörst oder siehst meine Botschaft und nimmst sie auf, was deine Gedanken beeinflusst. So kann man sich die Philosophie des Nebenläufigkeitsorientierten Programmierens (Concurrency Oriented Programming) vorstellen, auf der Erlang beruht.

10.4.3 Prozesse

Die Stärke von Erlang liegt darin, Programme in vielen Prozessen zu organisieren. Wird etwas benötigt erstellt man einen neuen Prozess, der sich darum kümmern soll. Hier erkennt man die Wurzeln Erlangs in der Kommunikation, denn für jede Verbindung kümmert sich ein eigener Prozess. Dabei ist es nicht wichtig wo die einzelnen Prozesse laufen, ob lokal oder verteilt auf verschiedenen Manschinen, denn sie sind total voneinander unabhängig. Sie teilen sich keine Daten. Jeder Prozess hat seinen eigenen Speicher, denn das wäre nicht effizient da sie so nicht parallel laufen könnten und auch recht kompliziert zu handhaben, da man sich mit Semaphoren, Mutexe, Locks und so weiter plagen müsste. Generell sind Prozesse in Erlang vom Betriebssystem unabhängig, was sie nicht so schwerfällig macht, wie die Protzesstrukturen eines Betriebssystems. So kann man problemlos mit sehr großen Anzahlen von Prozessen parallel arbeiten. In Erland ist jeder Prozess für sich und hat einen einzigartigen Namen der ihn identifiziert. Um den Speicher eines anderen Prozesses zu verändern muss man ihm eine Nachricht schicken und hoffen, dass sie empfangen und verstanden wird. Dieses Vorgehen nennt man 'Send and Pray'. Um sicher zu stellen, dass die Nachricht angekommen ist und etwas bewirkt hat muss man nachfragen. Das ist wie im echten Leben, hier ein Beispiel: *Rosi: Hey Klaus, Abendessen gibt es heute schon um 18 Uhr.* Die Prozesse sind in diesem Fall Menschen und ihr gemeinsames Interesse ist das Abendessen. Die Frau sagt ihrem Mann, wann es Essen gibt. Nun heißt es für Rosi abwarten und hoffen, dass Klaus auch zur richtigen Zeit zum Essen kommt (Send an Pray). Will sie sicher gehen, dass er ihr auch zugehört hat, muss sie noch mal nachfragen ob ihre Nachricht bei ihm angekommen ist: *Rosi: Hast du gehört? Klaus: Ja, 18 Uhr Abendessen.* Die Kommunikation zwischen den parallel laufenden Prozessen ist der Kern der Nebenläufigkeit in Erlang Prozessen. Für diesen Zweck gibt es den Bang Operator: das Ausrufezeichen. Die Nachricht von Rosi würde in Erlang etwa so aussehen: `Klaus ! {abendessen, 18}` Der Prozess Klaus wird *gebangt* (benachrichtigt) mit dem Wert {abendessen, 18}.

10.4.4 Beispiel für Prozesskummunikation

Das folgende Beispiel soll zeigen, wie Prozesse erstellt werden und Nachrichten austauschen. Das Beispiel erstellt einen neuen Prozess, sendet diesem Prozess eine Nachricht und bekommt eine Nachricht zurück, welche dann ausgegeben wird.

```
1  -module(echo).
2  -export([go/0, loop/0]).
3
4  go() ->
5          Pid2 = spawn(echo, loop, []),
6          Pid2 ! {self(), hello},
7          receive
8                  {Pid2, Msg} ->
9                          io:format("P1 ~w~n",[Msg])
10         end,
11         Pid2 ! stop.
12
```

```
13 loop() ->
14         receive
15                 {From, Msg} ->
16                         From ! {self(), Msg},
17                         loop();
18                 stop ->
19                         true
20         end.
```

Das Programm wird gestartet mit dem Aufruf der Funktion `go()` (`echo:go()`). In `go()` wird nun ein neuer Prozess erstellt, er erhält den Namen `Pid2` und ist ein Sohnprozess.

Der Befehl zur Prozesserzeugung `spawn` verlangt nach drei Dingen: einem Modul, einer Funktion in diesem Modul und Parameter für diese Funktion. Pid2 soll in diesem Fall die Funktion `loop()` ausführen, wofür er keine Parameter braucht, daher eine leere Liste (`[]`). `Pid2 = spawn(echo, loop, [])`

In Zeile 6 wird `Pid2` gebangt, das heisst ihm wird eine Nachricht geschickt. Inhalt der Nachricht ist die PID des Vaterprozesses, also die des laufenden Programms, und die Konstante hallo in Form eines Tuples. `Pid2 ! {self(), hello}`

Der Prozess `Pid2` wurde mit der Funktion `loop()` gestartet die nur aus einem `receive`-Block besteht und auf Nachrichten wartet. Ihre Aufgabe ist es Nachrichten zu empfangen und weiterzuschicken, bis eine Nachricht mit der Botschaft `stop` eintrifft.

Durch den Bang in Zeile 6 erhält nun `Pid2` eine Nachricht, die in das vorgegebene Schema passt. `From` und `Msg` sind Variablen, die die Werte der enpfangenen Nachricht aufnehmen. `From` ist die PID des Vaterprozess und `Msg` der Wert `hello`. Nun sendet der Sohnprozess seine eigene PID und die Nachricht an den Vaterprozess zurück. `From ! {self(), Msg}`

Der Vaterprozess ist ebenfalls mit einem `receive`-Block ausgestattet. Er wartet darauf eine Nachricht von `Pid2` zu erhalten und um diese Botschaft auszugeben. `{Pid2, Msg} -> io:format(P1 ~w ~n",[Msg])`

Hat er diese Nachricht erhalten, sendet er eine weitere Nachricht an `Pid2` mit der Botschaft `stop`. `Pid2` empfängt sie und merkt dank Pattern Maching, dass er genau das Richtige für `stop` hat. `Pid2` ist nun nicht mehr in seiner wartenden Funktion gefangen und beendet sich.

10.4.5 Errorhandling

Man kann Prozesse miteinander verlinken um davon informiert zu werden, ob ein Prozess abgestürzt ist. Diese Verknüpfungen zwischen Prozessen werden dazu genutzt Routinen zu implementieren, die auf Prozessabstürze reagieren können. Mit der Nachricht, dass ein Prozess verendete kommt die Information, wieso das geschehen ist. Somit ist der Prozess, der den Absturz registriert hat in der Lage den Prozess durch einen neuen zu ersetzen oder anderweitig das Programm zu retten. Diese Verknüpfung von Prozessen lassen sich in beide Richtungen anwenden, zu dem hat man die Möglichkeit komplexe Verknüpfungsketten und Bäume zu entwerfen.

10.5 Erlang in der Welt

10.5.1 Yaws - Yet another webserver

Yaws ist ein in Erlang geschriebener Webserver (http high performance 1.1), der jeden Client mit einem eigenen leichtgewichtigen Prozess bedient. Es gibt Yars als standalone Webserver und als embedded Webserver in Erlang Anwendungen. Der große Vorteil von Yars ist die Nebenläufigkeit der Sprache in der es implementiert ist. Vergleicht man es mit dem Apache Webserver in Bezug auf die Leistung bei parallelen Verbindungen, wird Yars Stärke deutlich. Eine bekannter Angriff auf ein solches System ist der so genannte "Distributed denial of service attack", bei dem versucht wird durch eine extrem hohe Anzahl gleichzeitiger Zugriffe den Webserver zu überlasten und somit außer Dienst zu stellen. Während der Apache Webserver in einem Vergleichstest[apa] schon bei 4000 gleichzeitigen Sessions in die Knie geht, bewältigt Yars über 80000 gleichzeitige Verbindungen. Der große Abstand der Ergebnisse beruht nicht darauf, dass Apache schlechteren Code verwendet als Yars, sondern das Apache auf das unterliegende Betriebssystem angewiesen ist. Jeder Webserver, der mit den Prozessen und Threads des Betriebssystems arbeitet würde wahrscheinlich zu einem ähnlichen Ergebnis kommen. Da Erlang nicht auf das Betriebssystem angewiesen ist, sondern seine eigene Prozessstruktur verwendet ist es von dieser Einschränkung nicht betroffen.

10.5.2 Der Erfolg

Der Telekommunikationsswitch AXD301 von Ericsson ist nicht nur das erste in Erlang realisierte Projekt, sonder auch der profitabelste bisher. Die hohe Robustheit und Zuverlässigkeit veranlasste zum Beispiel die Brittisch Telecom dazu ihr Netz auf dieses Gerät umzustellen und weiter auszubauen. Ericsson wurde dadurch zum Softswitch Weltmarktführer. Auch die Konkurrenz von Ericsson setzt immer mehr auf Erlang. Nortel Networks entwickelte mit Erlang den Alteon SSL Acclerator, was auch den Marktanteil dieses Unternehmens stark vergrößerte.

10.5.3 Fazit

Erlang ist eine sehr kompakte Sprache, mit der man schnell warm werden kann. Der Code fällt verhältnismäßig klein aus, was ihn übersichtlich und damit leicht wartbar macht. Das macht sich nicht zuletzt in den Kosten der Softwareentwicklung bemerkbar. Man kann schnell lauffähige Programme schreiben, mit denen im Sinne eines Prototyps Systemanforderungen überprüft werden können. Denkt man an die Entwicklung der multicore Prozessoren, könnte Erlang dank seiner hohen Nebenläufigkeit und Skalierbarkeit bald noch viel größeres Interesse wecken. Denn ein Erlang Programm sollte soviel mal schneller laufen, wie viele Prozessorkerne es verwenden kann[erlb].

Obwohl diese Sprache schon mehr als 20 Jahre alt ist hat sie das Potential vor allem in der Telekommunikation einer großen Zukunft entgegen zu gehen.

Kapitel 11

Prolog (Lars Thielmann)

11.1 Einleitung

Dieses Dokument soll die Spache Prolog etwas näher beleuchten und stellt die Grundkonzepte und den Aufbau der Sprache vor. Außerdem stellt es keinen Anspruch auf Vollständigkeit dar. Prolog ist eine logische Programmiersprache, die nach dem dem Paradigma der deklarativen Programmierung arbeitet. Der Unterschied zur imperativen Programmierung ist, bei der festgelegt wird wie und in welcher Reihenfolge ein Prbolem gelöst wird, dass Logik-Programme aus einer Menge von Axiomen bestehen. Diese Axiomen sind eine Ansammlung von Fakten und Regeln, in denen nicht beschrieben wird, wie das Problem gelöst wird. Das *Wie* ist bereits fest vorgegeben und zwar in einem sogenannter Beweiser oder auch Inferenzmaschine gennant. Dazu stellt man dem Interpreter eine Wissensbasis zur Verfügung und stellt Anfragen, die dieser mit Hilfe der Wissensbasis zu beantworten versucht, indem er die Lösung von den vorhandenen Regeln herleitet. Dies geschiet in Prolog mit einer *Depth-First*-Suche mit *Unifikation* und *Backtracking*. Erhält man ein negatives Ergebnis bedeutet dies, dass aufgrund der Wissensbasis keine Antwort gefunden werden kann. Das hängt eng mit der *closed world assumption*[1] zusammen.

Im weiteren Verlauf dieses Textes soll dieses Konzept und wie es funktioniert näher beschrieben, sowie einige Beispiele und besondere Fähigkeiten der Sprache gezeigt werden.

11.2 Geschichte

Im July 1970 kamen Robert Pasero und Philippe Roussel in Montreal an. Sie folgten einer Einladung von Alain Colmerauer, der zur Zeit Assistenzprofessor für Informatik an der Universität Montreal und Leiter des automatischen Übersetzungs Projektes TAUM (Traduction Automatique de l'Université de Montréal) war. Während ihrem zwei-Monatigen Aufenthalt in Montreal schrieben Robert und Philippe einige nicht-deterministische kontextfreie Analysato-

[1] Alles, was nicht eindeutig beweisbar ist, wird als falsch angesehen.

ren in Algol 60 und einen französichen Paraphrasen Generator in Q-Systems, die Programmiersprache, die Alain für sein Übersetzungsprojekt eintwickelte. Gleichzeitig began Jean Trudel, ein Doktorand von Alain, mit der Arbeit an einer automatischen Theroembeweisführung.

Nach einigen Jahren der Forschung, in denen ein erstes primitives System zur Kommunikation mit natürlicher Sprache entstand, entstand 1972 die erste Prolog Implementierung in Algol-W durch Philippe Roussel. Er war es auch, der zu diesem Zeitpunkt der Sprache ihren Namen gab: ***Pro**grammation en **Log**ique*.

Heute erfreut sich Prolog im Bereich der künstlichen Intelligenz großer Beliebtheit und wird zur Entwicklung sogenannter Expertensysteme eingesetzt.

11.3 Prinzip von Prolog

11.3.1 Aufbau der Sprache

Das Konzept von Prolog ein komplett anderes das von herkömmlichen Programmiersprachen. Es gibt keine Kontrollstrukturen, keine Zuweisungen, keine Datentypen (mit Außnahme der Liste). Alles was man kennen und lieben gelernt hat gibt es nicht mehr. Stattdessen muss der Entwickler eine Datenbank, oder auch Wissensbasis (engl. knowledge base), mit Fakten und Regeln erstellen, die das Problem vollständig beschreiben. Diese Wissensbasen basieren auf der Prädikatenlogik der ersten Stufe und bestehen aus einer Menge von Hornklauseln. Hornklauseln bestehen aus nur einem positiven Literal und alle Variablen sind gebunden. Die Klauseln entsprechen dabei Fakten und Regeln. Ein Fakt ist eine atomare Formel mit der Syntax

$p(t_1, ..., t_n).$

Dabei ist p das Prädikatszeichen und $t_1, ..., t_n$ die Terme. Neben den Fakten gibt es die Regeln. Regeln sind bedingte Aussagen mit der Syntax

$A : -B_1, ..., B_n.$

Dabei sind A und B atomare Formeln. A nennt man hierbei den Kopf und B den Rumpf der Regeln. Man kann Fakten auch leicht als Regeln aufschreiben, da sie bedingungslos wahr sind:

$p(t_1, ..., t_n) : -true.$

Nehmen wir nun ein kleines Beispiel um uns daraus eine kleine Wissensbasis aufzubauen. Da wir (hoffentlich) alle Asterix und Obelix kennen, brauche ich darauf wohl nicht näher einzugehen. Ich stelle vier Fakten und vier Regeln auf:

Fakten:

1. Asterix ist ein Gallier.
2. Obelix ist ein Gallier.
3. Caesar ist ein Römer.
4. Caesar ist Kaiser.

Regeln:

1. Alle Gallier sind stark.

2. Wer stark ist, ist mächtig.
3. Wer Kaiser und dazu noch Römer ist, ist mächtig.
4. Römer spinnen.

Wenn wir das nun in die oben vorgestellte Syntax bringen, sieht das dann so aus:

```
1  %Fakten
2  gallier(asterix).
3  gallier(obelix).
4  roemer(caesar).
5  kaiser(caesar).
6
7  %Regeln
8  stark(X) :- gallier(X).
9  maechtig(X) :- stark(X).
10 maechtig(X) :- roemer(X), kaiser(X).
11 spinnt(X) :- roemer(X).
```

Listing 11.1: gallier.pl

Im Interpreter kann ich nun Anfragen stellen, die mir dann beantwortet werden. Zum Beispiel möchte ich wissen, wer mächtig ist, stelle ich diese Frage dem Interpreter.

```
1 | ?- maechtig(X).
2
3 X = asterix ? ;
4
5 X = obelix ? ;
6
7 X = caesar
```

Das Semikolon ist eine Usereingabe, damit das nächste Ergebnis angezeigt wird. X enthält dabei die Werte, die aufgrund der Datenbasis gefunden wurden. Variablen werden immer groß geschrieben.

11.3.2 Vom Programm zur Berechnung

Aber wie findet der Prolog Interpreter die Antwort auf eine Frage? Um ein Anfrage zu beantworten, durchläuft der Interpreter das gegebene Programm mit einer Tiefensuche und durchläuft die Wissensbasis von oben nach unten. Dabei sucht er nach einer Klausel, deren Kopf sich mit der Anfrage deckt. Ist der gefundene Kopf ein Fakt, so ist der Beweis gelungen und die Anfrage beantwortet. Ist der gefundene Kopf eine Regel, so versucht der Interpreter diese Regeln von links nach rechts zu beweisen. Es kommt also drauf an, in welcher Reihenfolge man die Fakten und Regeln in die Wissensbasis führt. Das Umordnen von Klauseln und das Umvormulieren von Regeln kann für einen anderen Suchbaum sorgen. Dieser ist zwar logisch äquivalent, aber eine zuvor lösbar Anfrage kann

diesmal vielleicht nicht gelöst werden. Die Anfrage kann nicht mehr beantwortet werden, da sich die Lösung in einem nicht zu erreichenden Teilbaum steckt. Ist ein Ast bis zum Ende durchlaufen worden und es wurde keine Antwort gefunden, springt die Inferenzmaschine mittels Backtracking zurück zur letzten Verzweigung und durchsucht den nächsten Teilbaum.

11.4 Charakterisierende Eigenschaften

11.4.1 Der Cut-Operator '!'

Bei manchen Lösungen werden unnötige Berechnungen durchgeführt und sind dadurch ineffizient. Die Idee des Cut ist es, Suchbäume zu stutzen und unnötige Berechnungen zu vermeiden. Der Cut ist immer wahr und er verhindert ein Backtracking, das heißt für alle Ziele vor dem Cut werden keine Lösungen mehr gesucht und es können keine alternativen Klauseln ausgewertet werden. Die Variablenbelegung bleibt bestehen. Dazu ein kleines Beipiel:

```
1 if_then_else(P, Q, R) :- P, Q.
2 if_then_else(P, Q, R) :- not P, R.
```

Nehmen wir mal an P ist wahr, wenn die Klausel ausgewertet wird. Danach folgt die Auswertung von Q. Wenn Q nun nicht hergeleitet werden kann, kommt es zu Backtracking und es wird versucht P in einer alternativen Rechnung erneut herzuleiten. Das obwohl wir wissen, das P hergeleitet werden kann. Wenn P fehlschlägt, wird in einer anderen Klausel versucht not(P) herzuleiten, obwohl das gerade vorher bewiesen wurde. Nach Anänderung des Programms, mit Hilfe des Cut-Operators, sieht das Beispiel dann so aus:

```
1 if_then_else(P, Q, R) :- P, !,Q.
2 if_then_else(P, Q, R) :- R.
```

Wenn P nun wahr ist, springt die Kontrolle weiter zum Cut und dann weiter zum Q. Wenn Q nun nicht hergeleitet werden kann, findet kein Backtracking statt, sondern die Anfrage wird mit *No* beantwortet. Und wenn P fehl schlägt, wird die zweite Klausel ausgewählt und mit R die Berechung fortgesetzt.

Man unterscheidet:

- grüne Cuts
  - schneiden Suchbäume weg, die keine Lösungen enthalten.
  - verändern die deklarative Bedeutung des Programms nicht.
  - können weggelassen werden.
- rote Cuts
  - schneiden Suchbäume weg, die Lösungen enthalten.
  - verändern die deklarative Bedeutungs des Programms.
  - können nicht weggelassen werden.

11.5 Typische Probleme und Beispielapplikationen

Da Prolog eine logische Sprache ist, sind viele typische Probleme im Bereich der künstlichen Intelligenz. Aber auch in der Computerlingualistik. Mit Prolog lassen sich einfacher Programme entwerfen, die für die Verarbeitung natürlicher Sprache geeignet sind. Zum Beispiel Natürlich-sprachliche Datenbankschnittstellen. Bei der Netzplantechnik ist Prolog ebenfalls eine gute Wahl. Im folgenden sind zwei einfache Beispielprogramme gezeigt, die demonstrieren sollen wie man bestimmte Probleme mit Prolog löst.

11.5.1 Quicksort

Zu den großen Stärken von Prolog zählt die Entwicklung von Algorithmen. Der Entwickler kann sich ganz dem Problem widmen und muss sich keine Gedanken über den Ablauf oder Typen machen. Als Beispiel soll hier der Quicksort Algorithmus gezeigt werden. Dabei wird die zu sortierende Liste an einem willkürlichen Punkt (Pivot Element) geteilt. Alle größeren Elemente werden in eine, alle kleineren Elemente in eine andere Liste gesteckt. Auf diese neuen Listen wird wieder der Quick-Sort-Algorithmus angewandt (rekursiv), bis es nichts mehr zu sortieren gibt. Dann werden die Teillisten zu einer sortierten Liste verbunden. Im folgenden Beispiel wird als Pivot Element immer das erste gewählt.

```
1  quicksort([], []).
2  quicksort([Kopf|Rest], SortierteListe):-
3          teilen(Kopf, Rest, Kleiner, Groesser),
4          quicksort(Kleiner, Kleiner_Sortiert),
5          quicksort(Groesser, Groesser_Sortiert),
6          append(Kleiner_Sortiert, [Kopf|
              Groesser_Sortiert], SortierteListe).
7
8        teilen(_, [], [], []).
9        teilen(Element, [Kopf|Rest], [Kopf|Kleiner],
            Groesser):-
10                               Kopf < Element, !,
11                               teilen(Element, Rest,
                                    Kleiner, Groesser).
12       teilen(Element, [Kopf|Rset], Kleiner, [Kopf|
            Groesser]):-
13                               teilen(Element, Rest,
                                    Kleiner, Groesser).
```

Listing 11.2: quicksort.pl

11.5.2 Einsteins Rätsel

Ein schönes Beispiel zu einem Problem, welches man mit Prolog lösen kann, stellt Einsteins Rätsel dar. Es soll angeblich von ihm im 19. Jahrhundert erstellt worden sein mit dem Vermerk, dass nur 2% der Weltbevölkerung es lösen

können. Es geht darum, dass man herausfinden soll, wer der Personen einen Fisch als Haustier hat.

- Es gibt 5 Häuser mit je einer anderen Farbe.
- In jedem Haus wohnt eine Person anderer Nationalität.
- Jeder Hausbewohner bevorzugt ein bestimmtes Getränk, raucht eine bestimmte Zigarettenmarke und hält ein bestimmtes Haustier.
- Keine der 5 Personen trinkt das gleiche Getränk, raucht die gleichen Zigaretten oder hält das gleiche Tier wie seine Nachbarn.

Frage: Wem gehört der Fisch?

Dazu gibt es einige Hinweise:

- Der Brite lebt im roten Haus.
- Der Schwede hält einen Hund.
- Der Däne trinkt gern Tee.
- Das grüne Haus steht direkt links neben dem weißen Haus.
- Der Besitzer des grünen Hauses trinkt Kaffee.
- Die Person, die Pall Mall raucht, hält einen Vogel.
- Der Mann, der im mittleren Haus wohnt, trinkt Milch.
- Der Besitzer des gelben Hauses raucht Dunhill.
- Der Norweger wohnt im 1. Haus.
- Der Marlboro-Raucher wohnt neben dem, der eine Katze hält.
- Der Mann, der ein Pferd hält, wohnt neben dem, der Dunhill raucht.
- Der Winfield-Raucher trinkt gern Bier.
- Der Norweger wohnt neben dem blauen Haus.
- Der Deutsche raucht Rothmans.
- Der Marlboro-Raucher hat einen Nachbarn, der Wasser trinkt.

Mit Hilfe dieser Hinweise können wir nun eine Wissensbasis erstellen und im Interpreter eine entsprechende Anfrage stellen.

```
1 erstes(E, [E|_]).
2 mittleres(M, [_,_,M,_,_]).
3 links(A, B, [A|[B|_]]).
4 links(A, B, [_|R]) :- links(A, B, R).
5 neben(A, B, L) :- links(A, B, L); links(B, A, L).
6
```

```
run :-
  X = [_,_,_,_,_],
  member([rot,brite,_,_,_], X),
  member([_,schwede,_,_,hund], X),
  member([_,daene,tee,_,_], X),
  links([gruen,_,_,_,_], [weiss,_,_,_,_], X),
  member([gruen,_,kaffee,_,_], X),
  member([_,_,_,pallmall,vogel], X),
  mittleres([_,_,milch,_,_], X),
  member([gelb,_,_,dunhill,_], X),
  erstes([_,norweger,_,_,_], X),
  neben([_,_,_,marlboro,_], [_,_,_,_,katze], X),
  neben([_,_,_,_,pferd], [_,_,_,dunhill,_], X),
  member([_,_,bier,winfield,_], X),
  neben([_,norweger,_,_,_], [blau,_,_,_,_], X),
  member([_,deutsche,_,rothmans,_], X),
  neben([_,_,_,marlboro,_], [_,_,wasser,_,_], X),
  member([_,N,_,_,fisch], X),
  write('Der '), write(N), write(' hat einen Fisch als
      Haustier.'), nl.
```

Listing 11.3: einstein.pl

Wer die Antwort darauf wissen möchte, kann es gerne selber ausprobieren ;)

11.6 Editoren und Interpreter

Um Prolog Programme zu entwickeln reicht prinzipiell ein einfacher Texeditor. Aber es gibt auch spezielle Editoren und Entwicklungsumgebungen. Um den produzierten Code auch auszuführen, bedarf es noch einem Interpreter und/oder Compiler. Hier sollen ein paar vorgestellt werden.

SWI-Prolog Ein speziell für Prolog gedachter Editor mit integriertem Interpreter. Neben Syntax Highlighting bietet er noch einen Debugger mit grafischer Oberfläche.. Verfügbar für Windows, Linux und MacOS X

GNU-Prolog Der GNU Compiler. Besitzt auch einen Interpreter sowie eine umfangreiche Dokumentation, in der auch beschrieben wird, wie man Prolog Funktionen aus C-Programmen heraus aufruft. Er ist für alle gängigen Betriebssysteme und Architekturen verfügbar.

Visual Prolog Eine kommerzielle IDE mit allem was man von einer Entwicklungsumgebung erwarten kann. Compiler, Debugger, Syntaxhighlighting, Projekten und vielem mehr. Als Testversion für jedermann zu haben. Die Vollversion von Version 7.1 ist für 299Euro zu haben. Verfügbar allerdings nur für Windows 2000/XP.

SICStus Prolog Prolog System des SICS (Swedish Institute of Computer Science), erlaubt z. B. constraints basierte Programmierung (englisch)

QuProlog Ein freier Prolog Compiler für Linux, Unix und auch MacOS X.

Eclipse Es existiert auch ein Plugin für Eclipse um damit bequem mit Prolog zu arbeiten. Mehr Inforamtionen dazu gibt es auf www.amzi.com

11.7 Zusammenfassung

Prolog ist eine Programmiersprache die sich von herkömmlichen Sprachen wie C++ und Java vollkommen unterscheidet, da sie nach dem Konzept der deklarativen Programmierung arbeitet, die auf Prädikatenlogik beruht. Es wird beschrieben, was das Problem ist und nicht wie es gelöst werden soll. Dadurch lässt sich sehr schnell implementieren und ist gut für *Rapid-Prototyping*. Die Sprache ist plattformunabhängig und es gibt Werkzeuge für jede verbreitete Architektur und Betriebssystem.

Allerdings fehlen Mittel zur Strukturierung. Es gibt keine Datenkapselung und keine Module, was bei größeren Projekten problematisch werden kann.

Aber für alle Probleme, die auf logischen Zusammenhängen basieren, ist Prolog eine sehr gute Wahl.

Kapitel 12

XQuery
(Fabian Meyer)

12.1 Einleitung

Im Laufe der letzten Jahr hat sich ein neuer Standard für Datenhaltung in der Informationstechnik breit gemacht: Die Extensible Markup Language(kurz XML). Mit dem Aufkommen dieser neuen Technologie kamen jedoch nicht nur Vorteile. Eines der größten Probleme sind Queryanfragen auf die Datenbestände, die auf Datenbanken mit der Structured Query Language(kurz SQL) realisiert werden. Dies gestaltet sich in XML schwieriger, denn bestehende Technoloigien wie das Document Object Model(kurz DOM) und die Simple API for XML(kurz SAX) benötigen meist eine komplexe Implementierung in einer Hochsprache, bevor auf dem XML-Dokumenten gearbeitet werden kann. Daher entschloss sich das World Wide Web Consortium(kurz W3C) einen neuen Standard zur einfachen Verarbeitung von XML zu definieren und veröffentlichte nach einigen Betaversionen am 23. Januar 2007 die Version 1.0 von XQuery. Um die neue Sprache zu verstehen, werden jedoch Grundkenntnisse in XML sowie in XPath benötigt. Diese hoffe ich in den nächsten zwei Kapiteln vermitteln zu können.

12.2 XML

12.2.1 Einführung

Die Extensible Markup Language ist eine Auszeichnungssprache zur Darstellung und Speicherung von baumartigen Datenstrukturen. Sie ist ein vom World Wide Web Consortium definierter Standard, der am 10. Februar 1998 in seiner ersten Auflage spezifiziert wurde. Der aktuelle, vierte Version wurde am 29. September 2006 veröffentlicht und beinhaltet einige Erweiterungen wie zum Beispiel XML-Schema, auf das ich jedoch nicht weiter eingehen werde. XML ist stark an die Standard Generalized Markup Language angelehnt, jedoch nicht so komplex. Oft werden die Dokumente zum Datenaustausch zwischen Applikationen verwendet, da sie völlig unabhängig sind und in fast jeder Hochsprache eine Implementierung zur Verarbeitung vorliegt.

12.2.2 Aufbau

Im Kopf eines XML-Dokuments werden die Version, die Kodierung sowie einige andere Eigenschaften definiert. Das Dokument besteht aus so genannten Elementen, Attributen und Textknoten. Elemente werden durch ein öffnendes Tag definiert und mit einem schließenden Tag beendet. Ein jedes Element kann Unterelemente, so genannte Kinder, haben, wobei Text eine Sonderform eines Kindes ist, die als Textknoten bezeichnet wird. Zudem gibt es kinderlose Elemente, diese werden im öffnenden Tag direkt wieder geschlossen. Eine Überschneidung von Elementen ist nicht Möglicht. Attribute sind Eigenschaften von Elementen und werden im öffnenden Tag des Elements durch ein Key-Value-Paar definiert. Hält ein Dokument diese Regeln ein, gilt es als wohl geformt.

12.2.3 Beispiel

```
1  <?xml version="1.0" encoding="ISO-8859-1" standalone="
      yes"?>
2  <buchladen>
3     <buch name="Buch1" preis="19.90">
4        <kapitel titel="Kapitel 1" />
5        <kapitel titel="Kapitel 2" />
6        <kapitel titel="Kapitel 3" />
7        <kapitel titel="Kapitel 4" />
8        <kapitel titel="Kapitel 5" />
9     </buch>
10    <buch name="Buch2" preis="29.90">
11       <kapitel titel="Kapitel 1" />
12       <kapitel titel="Kapitel 2" />
13       <kapitel titel="Kapitel 3" />
14       <kapitel titel="Kapitel 4" />
15    </buch>
16    <buch name="Buch3" preis="39.90">
17       <kapitel titel="Kapitel 1" />
18       <kapitel titel="Kapitel 2" />
```

```
        <kapitel titel="Kapitel 3" />
        <kapitel titel="Kapitel 4" />
        <kapitel titel="Kapitel 5" />
        <kapitel titel="Kapitel 6" />
    </buch>
</buchladen>
```

12.3 XPath

12.3.1 Einführung

Die Abkürzung XPath steht für XML Path Language und steht, wie der Name schon erahnen lässt, für eine Abfragesprache um einen Pfad durch XML-Dokumente zu bilden. Sie ermöglicht es, bestimmte Elemente aus der Baumstruktur auszuwählen, auf Kinder, Eltern etc. dieser Elemente zuzugreifen, sowie einfach Zähl- und Vergleichsoperationen auszuführen. Die Sprache wurde, wie auch schon XML, von dem World Wide Web Consortium standardisiert und im Jahr 1999 in der Version 1.0 veröffentlicht. Seitdem hat sie in einer Vielzahl von Implementierungen zur Verarbeitung von XML-Dokumenten Einzug gehalten.

12.3.2 Aufbau

Ein Ausdruck in XPath setzt sich aus einem oder mehreren Lokalisierungsschritten, gefolgt von beliebig vielen Prädikaten zusammen. Ein Lokalisierungsschritt besteht aus einer Achse sowie einem optionalen Knotentest, welcher die Auswahl einschränkt und durch zwei Doppelpunkte separiert ist. Folgende Achsen sind in XPath definiert:

Achse	adressierte Knoten	Abkürzung
child	direkt untergeordnete Knoten	oder *
parent	der direkt übergeordnete Elternknoten	..
self	der Kontextknoten selbst (nützlich für zusätzliche Bedingungen)	.
ancestor	übergeordnete Knoten	
ancestor-or-self	übergeordnete Knoten inklusive des Kontextknotens	
descendant	untergeordnete Knoten	
descendant-or-self	untergeordnete Knoten inklusive des Kontextknotens	//
following	unabhängig von der Hierarchie der Knoten, nachfolgend im XML-Dokument	
following-sibling	wie following, und vom gleichen parent stammend	
preceding	gleiche Ebene der Baumstruktur, vorhergehend im XML-Dokument	
preceding-sibling	wie preceding, und vom gleichen parent stammend	
attribute	Attributknoten	@
namespace	Namensraumknoten die aus dem Attribut xmlns stammen	

[?]

Prädikate können eine Auswahl, zusätzlich zu einem Knotentest, weiter einschränken. Sie werden in eckigen Klammern definiert, können auf Vergleichsoperatoren wie = != and or ¡ ¿ ¡= ¿=, mathematische Operatoren wie + - * div mod zugreifen und verfügen zusätzlich über folgende Funktionen:

Funktion	Beschreibung
normalize-space()	Entfernen von Leerzeichen am Anfang und Ende des Strings und Reduktion aufeinanderfolgender Leerzeichen auf eines
substring()	Einen Teilstring selektieren
substring-before(source, splitter)	Einen Teilstring vor dem ersten Vorkommen des Trennzeichens selektieren
substring-after(source, splitter)	Einen Teilstring nach dem ersten Vorkommen des Trennzeichens selektieren
string-length()	Länge des Strings
count()	Anzahl der Knoten in einer Knotenmenge
id()	Selektiert Elemente über die DTD-ID
name()	Name des Knotens

[?]

12.3.3 Beispiele

Die folgenden Beispiele beziehen sich auf das XML-Dokument des Beispiels aus dem vorangehenden Kapitel.

```
//child::Buch[@preis > 20]
```

Selektiert alle Bücher die mehr als 20 kosten.

```
//child::Buch[count(kapitel) mod 2 == 0]
```

Selektiert alle Bücher die eine gerade Kapitelanzahl haben.

12.4 XQuery

12.4.1 Einführung

Auf der Suche nach einer einfachen, intuitiven Abfragesprache für XML wurde im Jahr 2007 die Version 1.0 von XQuery verabschiedet. Sie verwendet einen an SQL und C angelehnten Syntax und XPath zur Pfad Darstellung. Die Entwickler erhofften sich durch den Release dieser einfach gehaltenen Sprache XML weiter zu verbreiten und auch Laien die Verarbeitung zu ermöglichen. XQuery ist stark typisiert und Turing-vollständig. Ein sehr großes Anwendungsgebiet von XQuery ist es, XML in einer andere Markup-Sprache, zum Beispiel HTML, zu transformieren. Ein einfaches XML-Dokument ist für einen außen stehenden nur sehr schwer zu lesen und zu verstehen, dort setzt XQuery an. Mit einer einfachen Abfrage ist es möglich HTML und XML zu mischen und so eine benutzerfreundliche Darstellung zu ermöglichen.

12.4.2 Aufbau

12.4.3 Variablen

Variablen sind in XQuery stark Typ gebunden und werden mit einem $ anführend definiert. Die Typisierung wurde von XML-Schema übernommen und bietet folgende einfache Datentypen:

Typ	Beschreibung
xsd:string	Einfacher Datentyp zum Speichern von Zeichenketten, zum Beispiel 'Das ist ein String'
xsd:decimal	Eine Dezimalzahl, zum Beispiel '1246.67'
xsd:integer	Eine ganze Zahl, zum Beispiel '-1234'
xsd:float	Eine Flieskommazahl, zum Beispiel '1.234E10'
xsd:boolean	Ein boolscher Ausdruck, zum Beispiel 'true'
xsd:date	Eine Datumsangabe, zum Beispiel '2007.11.12'
xsd:time	Eine Zeitangabe, zum Beispiel '11:13:56'

Zusätzlich zu diesen einfachen Datentypen gibt es die Möglichkeit komplexe Datentypen zu definieren. Somit können Sequenzen, eine Auswahl etc. definiert werden. Folgend zwei kleine Beispiele:

```
1  <xsd:complexType name="pc-Typ">
2     <xsd:sequence>
3        <xsd:element name="name"        type="xsd:string
            "/>
4        <xsd:element name="hersteller" type="xsd:string
            "/>
5        <xsd:element name="prozessor"  type="xsd:string
            "/>
6        <xsd:element name="mhz"        type="xsd:integer
            " minOccurs="0"/>
7        <xsd:element name="kommentar"  type="xsd:string
            "  minOccurs="0" maxOccurs="unbounded"/>
8     </xsd:sequence>
9     <xsd:attribute name="id" type="xsd:integer"/>
10 </xsd:complexType>
11
12 <xsd:complexType name="computer">
13    <xsd:choice>
14       <xsd:element name="desktop" type="pc-Typ"/>
15       <xsd:element name="laptop" type="laptop-Typ"/>
16    </xsd:choice>
17 </xsd:complexType>
```

[?]

12.4.4 Abfragen

Alle Abfragen in XQuery bauen auf dem dem FLWOR(gesprochen flower) Prinzip auf. Dies ist eine Kombination auf verschiedenen Anweisung die nun im einzelnen erklärt werden.

Funktion	Parameter	Beschreibung
for	$forvar1 at $posvar1 in ¡Expr¿	$forvar ist das aktuell ausgewählte Element der in ¡Expr¿ definierten Menge, über die iteriert werden soll. Zusätzlich kann in $posvar1 eine Variable deklariert werden, in der die aktuelle Position des Elements gespeichert wird.
let	$letvar1 := ¡Expr¿	Als $letvar1 kann somit eine Variable mit einem beliebigen Ausdruck initialisiert und im Verarbeitungsschritt verwendet werden.
where	¡BoolExpr¿	Hier kann gefilter werden, welche Elemente in den Verarbeitungsschritt überführt und welche direkt verworfen werden. Dies wird durch einen boolschen Ausdruck definiert.
order by	¡Expr¿ ascending/descending	Gibt an in welcher Reihenfolge die Elemente sortiert werden sollen. Zudem kann mit ascending/descending angegeben werden, ob auf- oder absteigend sortiert wird.
return		In return geschieht nun die Verarbeitung der Elemente. Hier können Operationen durchgeführt und Mitschungen mit anderen Markupsprachen vorgenommen.

12.4.5 Funktionen

Vordefinierte Funktionen

In XQuery gibt es einen kleinen Pool von vordefinierten Funktionen die zur Verarbeitung von XML-Dokumenten verwendet werden können. Die wichtigen Knotenfunktionen werde ich nun vorstellen, alle weiten Funktionen können hier nachgelesen werden.

Funktion	Parameter	Beschreibung
fn:node-name(node)	Knoten	Gibt den Knotennamen zurück.
fn:nilled(node)	Knoten	Gibt an, ob der Knoten leer ist.
fn:base-uri(node)	Knoten	Gibt die URI des Knotens zurück.
fn:document-uri(node)	Knoten	Gibt die URI des Dokuments zurück.

Eigene Funktionen

Zusätzlich zu den vordefinierten Funktionen gibt es in XQuery die Möglichkeit eigene Funktionen zu definieren. Dies geschieht wie in den meisten anderen Programmiersprachen auch:

```
declare function prefix:funktionsname($parameter AS datatype)
  AS Rückgabedatentyp
{
   //Funktionscode
};
```

Nun ein kleines Beispiel wie eine solche Funktionen aussehen kann:

```
declare function mathe:quadrat($zahl AS xs:decimal?)
  AS xs:decimal?
{
   let $ergebnis := $zahl * $zahl
   return $ergebnis
};
```

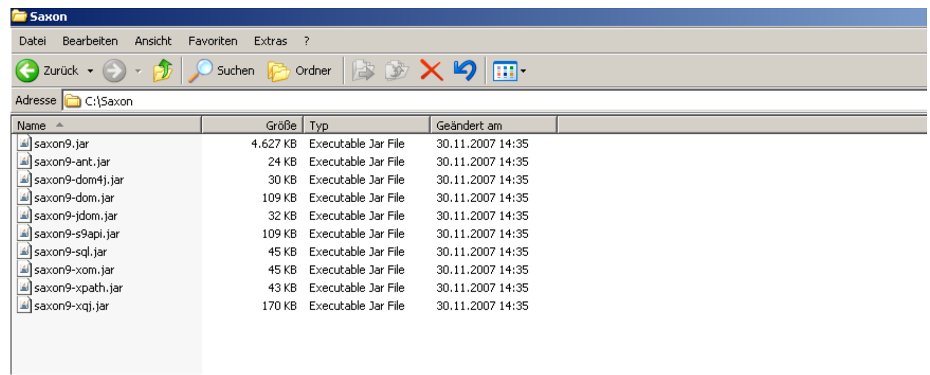
12.4.6 Intallation

Um unsere XQuery Script zu testen, benötigen wir eine einen Interpreter. Ich habe mich für SAXON entschieden, da er relativ einfach zu installieren und benutzen ist. Die folgenden Schritte werden Sie durch die Installation führen. Unter der URL
http://sourceforge.net/projects/saxon ist das Projekt hinterlegt und kann herunter geladen werden.

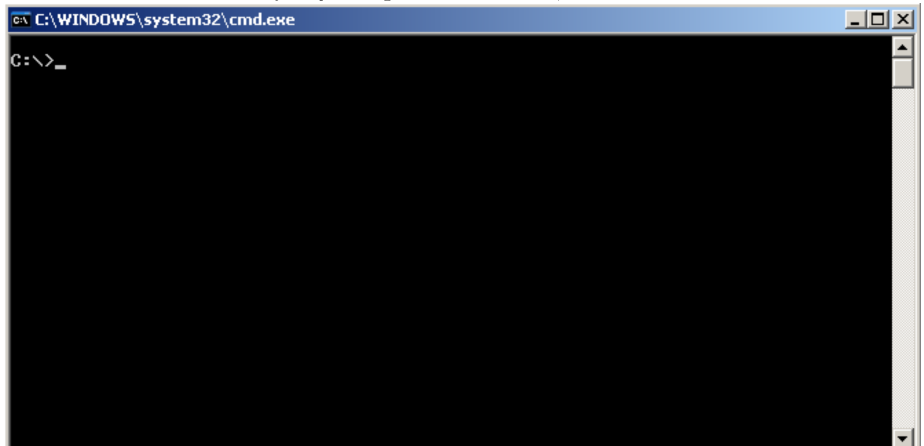
Latest File Releases

Package	Release	Date	Notes / Monitor	Downloads
aelfred	7.0	November 27, 2002	-	Download
DTDGenerator	7.0.1	September 26, 2007	-	Download
instant saxon	6.5.3	August 10, 2003	-	Download
saxon	9.0.0.2	November 30, 2007	-	Download
saxon6	6.5.5	November 24, 2005	-	Download

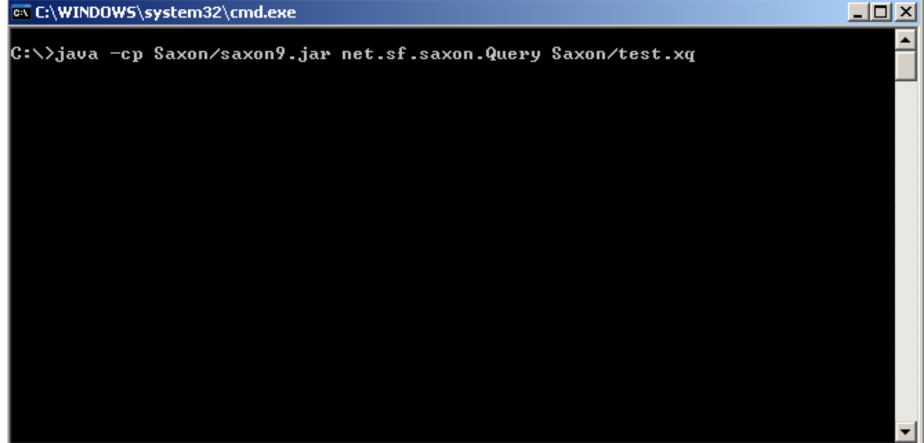
Das heruntergeladene Archiv wird nun in einen beliebigen Ordner entpackt.



Liegt eine Installation der Java Virtual Machine vor, ist die Installation an diesem Punkt bereit abgeschlossen. Andernfalls kann diese unter http://www.java.com/de/download/manual.jsp heruntergeladen und installiert werden. Um nun ein XQuery Script auszuführen, öffnen Sie die Kommandozeile.



Als letzten Schritt müssen Sie Java mit den Parametern -cp [pfad zu Saxon] net.sf.saxon.Query [pfad zum XQuery Script] aufrufen



Mit dieser kurzen Anleitung sind Sie nun in der Lage die folgenden XQuery Beispiele umzusetzen und nachzuvollziehen.

12.5 Beispiele

Alle Beispiele beziehen sich auf das im XML-Kapitel definierten Beispiel, unserem kleinen Buchladen. Zuerst möchte ich Ihnen ein Gefühl dafür geben, wie XQuery überhaupt arbeitet. Daher werden wird unser Dokument nur ausgeben und keine Änderungen daran vornehmen. Dies geschieht wie folgt:

```
1 for $buch in doc("test.xml")/buchladen/buch
2 return $buch
```

Wir machen nicht anderes als unser File zu öffnen und über einen XPath Ausdruck festzulegen, dass wir über all unsere Bücher in der Variablen $buch iterieren möchten. Dies erzeugt folgendes Ergebnis:

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <buch name="Buch1" preis="19.90">
3     <kapitel titel="Kapitel 1"/>
4     <kapitel titel="Kapitel 2"/>
5     <kapitel titel="Kapitel 3"/>
6     <kapitel titel="Kapitel 4"/>
7     <kapitel titel="Kapitel 5"/>
8  </buch>
9  <buch name="Buch2" preis="29.90">
10    <kapitel titel="Kapitel 1"/>
11    <kapitel titel="Kapitel 2"/>
12    <kapitel titel="Kapitel 3"/>
13    <kapitel titel="Kapitel 4"/>
14 </buch>
15 <buch name="Buch3" preis="39.90">
16    <kapitel titel="Kapitel 1"/>
17    <kapitel titel="Kapitel 2"/>
18    <kapitel titel="Kapitel 3"/>
19    <kapitel titel="Kapitel 4"/>
20    <kapitel titel="Kapitel 5"/>
21    <kapitel titel="Kapitel 6"/>
22 </buch>
```

Damit sind wir leider nicht sehr viel weiter als zuvor. In unserem Dokument wurden lediglich ein Knoten entfernt, was zudem dazu führt, dass es nicht mehr wohl geformt ist. Als nächsten bringen wir etwas HTML mit ins Spiel und versuchen unser schwer leserliches XML-Dokument in ein verständliches HTML-Dokument um zu forment.

```
1 <html>
2    <head>
3       <title>Mein Buchladen</title>
4    </head>
5    <body>
6       <ul>
7       {
```

```
8           for $buch in doc("XMLBeispiel.xml")/buchladen
               /buch
9           return
10             <li preis="{data($buch/@preis)}">{data(
                  $buch/@name)}</li>
11             <ul>
12             {
13                for $kapitel in $buch/kapitel
14                return
15                   <li>{data($kapitel/@titel)}</li>
16             }
17             </ul>
18       }
19       </ul>
20    </body>
21 </html>
```

In diesem Fall betten wir unser XML-Dokument in HTML Listen ein. Eine geschweifte Klammer leitet das Skript ein und beendet dies auch wieder. Wir iterieren über jedes Buch und seine Unterkapitel und geben dies aus.

```
1  <html>
2     <head>
3        <title>Mein Buchladen</title>
4     </head>
5     <body>
6        <ul>
7           <li preis="19.90">Buch1</li>
8           <ul>
9              <li>Kapitel 1</li>
10             <li>Kapitel 2</li>
11             <li>Kapitel 3</li>
12             <li>Kapitel 4</li>
13             <li>Kapitel 5</li>
14          </ul>
15          <li preis="29.90">Buch2</li>
16          <ul>
17             <li>Kapitel 1</li>
18             <li>Kapitel 2</li>
19             <li>Kapitel 3</li>
20             <li>Kapitel 4</li>
21          </ul>
22          <li preis="39.90">Buch3</li>
23          <ul>
24             <li>Kapitel 1</li>
25             <li>Kapitel 2</li>
26             <li>Kapitel 3</li>
27             <li>Kapitel 4</li>
28             <li>Kapitel 5</li>
29             <li>Kapitel 6</li>
```

```
          </ul>
       </ul>
    </body>
</html>
```

Möchte man nun nach ganz bestimmten Büchern suchen, kann man zusätzliche Bedingungen einfließen lassen.

```
<html>
    <head>
       <title>Mein Buchladen</title>
    </head>
    <body>
       <ul>
       {
          for $buch in doc("XMLBeispiel.xml")/buchladen
             /buch[@preis < 30]
          where fn:contains(fn:string($buch/@name), "
             Buch1")
          return
             <li preis="{data($buch/@preis)}">{data(
                $buch/@name)}</li>
             <ul>
             {
                for $kapitel in $buch/kapitel
                return
                   <li>{data($kapitel/@titel)}</li>
             }
             </ul>
       }
       </ul>
    </body>
</html>
```

Wir haben nun das Beispiel so modifiziert, dass wir über den Pfad nur Knoten selektieren, deren Preis ¡ 30 ist. Zudem haben wir in ‘where‘ definiert, dass wir nur Bücher möchten, deren Titel ‘Buch1‘ enthält. Dies führt zu folgendem Ergebnis:

```
<html>
    <head>
       <title>Mein Buchladen</title>
    </head>
    <body>
       <ul>
          <li preis="19.90">Buch1</li>
          <ul>
             <li>Kapitel 1</li>
             <li>Kapitel 2</li>
             <li>Kapitel 3</li>
```

```
            <li>Kapitel 4</li>
            <li>Kapitel 5</li>
         </ul>
      </ul>
   </body>
</html>
```

12.6 Fazit

Alles in allem kommt XQuery mit sehr guten Ideen und Ansätzen. Leider fehlt im Moment noch die Möglichkeit XML-Dokumente zu verändern. Zudem stellt sich die Frage, ob XQuery überhaupt noch als eine Query-Sprache zu bezeichnen ist. Sie bietet eine Vielzahl an vordefinierten Funktionen und lässt zudem die eigene Definition von solchen zu. Dadurch geht sie eher in die Richtung von Stored Procedues. Dennoch lassen sich grundlegende Anfragen schnell und einfach realisieren und verbergen die Komplexität, die diese Sprache eigentlich bietet.

Literaturverzeichnis

[apa] Apache vs. yaws. Website. `http://www.sics.se/~joe/apachevsyaws.html`.

[Arm03] Joe Armstrong. *Making reliable distributed systems in the presence of sodware errors*. PhD thesis, 2003.

[Arm07] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Programmers, 2007.

[AVWW] Joe Armstrong, Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG*. Pragmatic Programmers.

[Bri] Walter Bright. Digitalmars. Website. `http://www.digitalmars.com/d/`.

[DI] Hal Daumé III. Yet Another Haskell Tutorial. `http://darcs.haskell.org/yaht/yaht.pdf`.

[doc] docwiki.et. Website. `http://www.docwiki.net/`.

[erla] Erlang. Website. `http://www.erlang.org/`.

[erlb] What's all this fuss about erlang? Website. `http://www.pragmaticprogrammer.com/articles/erlang.html`.

[Eth] Eric Etheridge. Haskell Tutorial for C Programmers. `http://www.haskell.org/~pairwise/intro/intro.html`.

[ffc] Flying frog consultancy. Website. `http://www.ffconsultancy.com`.

[fsh] F sharp .net. Website. `http://fsharp.name/`.

[HPF] Paul Huda, John Peterson, and Joseph Fasel. A Gentle Introduction to Haskell, Version 98. `http://www.haskell.org/tutorial/index.html`.

[hub] Hubfs: The place for f sharp. Website. `http://cs.hubfs.net/`.

[Hug90] John Hughes. Why Functional Programming Matters. In D. Turner, editor, *Research Topics in Functional Programming*. Addison-Wesley, 1990.

[Knu69] Donald Knuth. *The Art of Computer Programming volume 2: Seminumerical algorithms*. 1969.

[Man07] Programmiersprache d. 2007.

[MS07] Philipp Haller Michel Schinz. A Scala Tutorial for Java programmers.
http://www.scala-lang.org/docu/files/ScalaTutorial.pdf, 2007.

[MSR] F sharp homepage at microsoft research. Website. http://research.microsoft.com/fsharp/fsharp.aspx.

[NET] .net framework developer center. Website. [?].

[New] Jeff Newbern. All about Monads.
http://www.nomaware.com/monads/html/.

[Oa] Martin Odersky and al. Scala Homepage.
http://www.scala-lang.org/.

[Oa06a] Martin Odersky and al. An overview of the scala programming language - second edition. Technical Report LAMP-REPORT-2006-001, EPFL Lausanne, Switzerland, 2006.

[Oa06b] Martin Odersky and al. An Overview of the Scala Programming Language - Second Edition.
http://www.scala-lang.org/docu/files/ScalaOverview.pdf, 2006.

[Ode07a] Martin Odersky. Scala By Example.
http://www.scala-lang.org/docu/files/ScalaByExample.pdf, 2007.

[Ode07b] Martin Odersky. The Scala Language Specification - Draft 2.6.
http://www.scala-lang.org/docu/files/ScalaReference.pdf, 2007.

[Ode07c] Martin Odersky. Scala Rationale.
http://www.scala-lang.org/docu/files/ScalaRationale.pdf, 2007.

[Pap01] Lothar Papula. *Mathematik für Ingenieure und Naturwissenschaftler, Band 1, 10. Auflage*. Vieweg, 2001.

[PJ03] Simon L. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[PJ07] Simon L. Peyton Jones. A Taste of Haskell. O'Reilly Open Source Convention, Portland, Oregon, USA
http://blip.tv/file/324976, http://blip.tv/file/325646/, July 2007.

[str] Homepage von robert pickering. Website. http://www.strangelights.com.

[VB] Carsten Böckmann Volkert Barr. Moderne programmierung nebenläufiger und verteilter anwendungen mit erlang.