BERUFSAKADEMIE MANNHEIM
STAATLICHE STUDIENAKADEMIE
Fachrichtung Informationstechnik

Seminar Informationstechnik

Funktionale Programmiersprachen

von

Martin Kolleck
Matrikelnr.: **181984**

Thijs Metsch
Matrikelnr.: **197534**

Mannheim, den 6. Mai 2004

Deutsches Zentrum
für Luft- und Raumfahrt
e.V.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Reden mit dem Computer

Um überhaupt eine Art von Kommunikation zwischen zwei Menschen oder Mensch und Computer aufbauen zu können, muss erst die Schnittstelle definiert werden. Diese Schnittstelle sorgt für die Grundlagen, die eine weitere Kommunikation ermöglichen. Die Schnittstelle zur Kommunikation zwischen Mensch und Computer ist heutzutage durch Tastatur (Maus) und Bildschirm gegeben. Lochkarten oder sogar einfache Kippschalter waren früher von Nöten um Großrechner zu betreiben. Ist die Schnittstelle gegeben bzw. definiert worden, kann der Mensch mit dem Computer kommunizieren. Damit es nicht zu Verständnisproblemen kommt muss eine Sprache definiert werden.

2 Sprachen

Eine Sprache wird definiert durch:

- Die Menge der Symbole, die in Worten verwendet werden können. (Alphabet)
- Die Menge der Worte, die aus diesem Alphabet gebildet werden. (lexikalischer Aufbau)
- Die Menge der gültigen Sätze, die aus Worten gebildet werden können. (syntaktischer Aufbau)
- Die Bedeutung der Sätze. (semantischer Aufbau)

Sind diese Regeln festgelegt, verfügen beide Partner über dieselbe Basis und können diese zur Kommunikation nutzen. Programmiersprachen sind eher an der menschlichen Sprache orientiert und nicht an den sehr einfachen Maschinensprachen, die der Computer ausführen kann. So kann man Programmiersprachen in höhere und niedere Programmiersprachen unterteilen. Zu den ersten zählen unter anderem die objektorientierten Programmierspachen und zu den zweiten Assembler.

Um die Programmiersprachen ausführen zu können, muss man sie in die Maschinensprache übersetzen, die von der verwendeten Computerplattform abhängt. Erst danach ist der Prozessor in der Lage, die Programme auszuführen. Hierfür gibt es wiederum eine Unterteilung in:

- Compiler - Das Programm wird übersetzt. Das Ergebnis der Kompilierung kann auf der jeweiligen Plattform ausgeführt werden.

- Interpreter - Das Programm wird zur Laufzeit verarbeitet.

Die meisten Programmiersprachen können mit Hilfe von Automaten verarbeitet werden. Zum Beispiel mit der Turingmaschine. Ebenso gibt es die Möglichkeit mit Hilfe von Kalkülen zu arbeiten. Ein Kalkül ist ein Regelsystem, das mittels Axiomen und Regeln einen Handlungsraum definiert. So ist das Lambda-Kalkül eine formale Sprache zur Untersuchung von Funktionen.

3 Geschichte der Programmiersprachen

Folgende Graphik zeigt die verschiedene Beziehungen bzw. Einflüsse zwischen den Programmiersprachen.

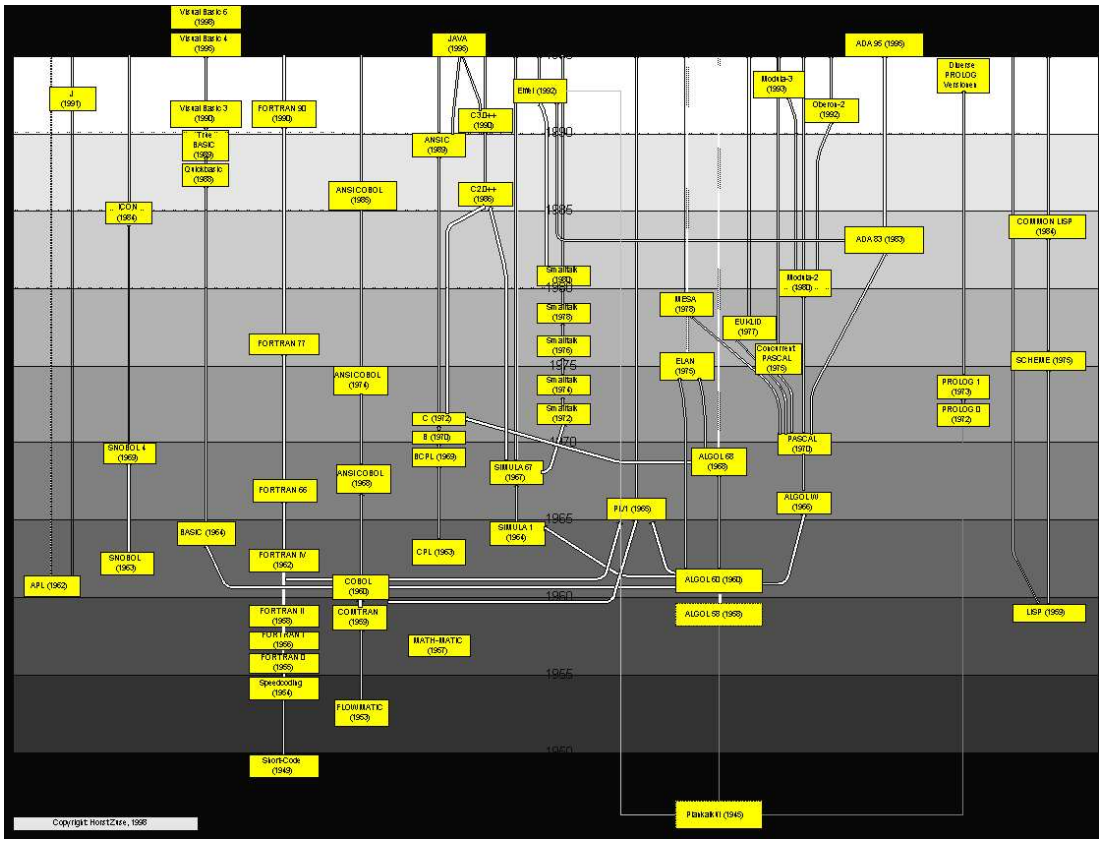

Abbildung 1: Entwicklung der Programmiersprachen [7]

Einen großen Einfluss auf viele Programmiersprachen hat das Plankalkül eine Programmiersprache, die in den 40er Jahren von Konrad Zuse entwickelt wurde. Sie wurde erst wesentlich später an der Universität Berlin implementiert. Das Plankalkül setzt sich aus:

- Zuordnungsanweisungen (assignment statements),
- Unterprogrammen (sub routines),
- bedingte Anweisungen (conditional statements),
- Schleifen (loops),
- Gleitkommaarithmetik (floating point arithmetic),
- Feldvariablen (arrays),
- zusammengesetzten Datentypen (structs) und
- Zusicherungen (assertions)

zusammen.

Mittels dieser Elemente war es möglich Algorithmen zu entwickeln.

Ein weiteres Beispiel, welches die Beziehung zwischen den Sprachen aufzeigt ist Java. Aus C++ und C geht diese Programmiersprache hervor. So wurde das Konzept der Objektorieniertheit von C++ in die Sprache bernohmen. Die Pointerarithmetik von C/C++ wurde jedoch fallen gelassen.

4 Typen von Programmiersprachen

Programmiersprachen kann man in verschiedene Typen unterordnen. Die vier Basistypen sind die iterativen, objektorientierten, funktionalen und die logischen/ deklarativen Sprachen.

4.1 Imperative Programmiersprachen

Imperative Programmiersprachen sind die am weitesten verbreiteten Programmiersprachen. Sie basieren auf einer festen Folge von Befehlen, die vom Prozessor ausgeführt werden. Dazu gehören Sprachen wie C, Pascal und Basic. Die wesentlichen Konstrukte bei diesen Sprachen sind die Sequenz, die Selektion und die Iteration. Durch diese drei Konstrukte wird die gesamte Ablaufsteuerung gewährleistet. Die Sequenz ist eine Zusammenfassung von mehreren kleinen Befehlen zu einer einzigen großen Befehlseinheit, die Selektion bietet die Auswertung von Bedingungen und die Iteration ermöglicht es, einen Befehl zu wiederholen. Die Vorteile der imperativen Programmiersprachen liegen in der Nähe zum Prozessor, der ebenfalls nur eine Folge von Befehlen verarbeiten kann und in dem

damit verbundenen Einfluss auf die Abarbeitungsreihenfolge des Programms. Ein weiterer Vorteil ist darin zu sehen, dass alle imperativen Programme stark algorithmisch aufgebaut sind. Zu diesem Thema gibt es bereits viele Theorien und Ansätze, so dass der Programmierer bei der Problemlösung auf ein breites, gut erforschtes und erprobtes Wissen aufbauen kann. Doch es gibt nicht nur Vorteile. Die größten Schwächen zeigen die imperativen Programmiersprachen, wenn es darum geht, ein solches Programm zu verstehen. Die Quelltexte sind meist schwer lesbar und wenig intuitiv geschrieben. Meist bleibt einem nichts weiter übrig mit einem Debugger Zeile für Zeile nachzuverfolgen und die Änderung von Variablen nachzuvollziehen, um hinter die Funktionsweise eines Programms zu kommen. Zur Unübersichtlichkeit tragen auch die sogenannten Seiteneffekte[1] bei.

4.2 Objektorientierte Programmiersprachen

Die nächste Klasse von Programmiersprachen sind die objektorientierten Sprachen. Sie lehnen sich noch an die imperativen Sprachen an und einige sind aus solchen hervorgegangen. Als Beispiele seien hier C++, Java und Oberon genannt. Wie der Name suggeriert beschäftigen sich diese Sprachen mit Objekten, die Eigenschaften (Attribute) und Fähigkeiten (Methoden) haben. Das primäre Konstrukt in der objektorientierten Programmierung ist die Klasse. Des weiteren sind die Konzepte der Vererbung und der Polymorphie wesentlich. Jedes Objekt wird durch eine Klasse repräsentiert. Die Attribute stellen einen bestimmten Zustand des Objekts dar, während die Methoden ein Verhalten an die Außenwelt weitergeben. Innerhalb der Methoden werden die Konstrukte der imperativen Programmiersprachen verwendet. Beziehungen zwischen den Objekten werden unter anderem durch Vererbung hergestellt. Die Polymorphie bietet dem Programmierer die Möglichkeit, Objekte mit gemeinsamen Eigenschaften in einer Vererbungshierarchie zusammenzufassen und erst zur Laufzeit festzulegen, von welchem Typ ein Objekt ist und dann eine entsprechende Methode aufzurufen. Dies wird auch als late-binding bezeichnet. Die Vorteile objektorientierter Programmiersprachen sind vor allem in deren Modularität zu finden. Die Aufteilung in Objekte macht es möglich, bestimmte Objekte in anderen Programmen wiederzuverwenden. Des Weiteren ist es einfacher, das Programm zu warten, da sich Änderungen oft nur auf ein einzelnes Modul auswirken. Die Kehrseite der Medaille ist ein hoher Aufwand in der Designphase. Projekte, die mit objektorientierter Programmierung umgesetzt werden, müssen sehr gut geplant und durchdacht werden, damit zwischendurch keine Probleme auftreten. Einen weiteren Nachteil findet man im nötigen Mehraufwand bei der Programmierung.

[1] Änderung einer globalen Variablen in einer Funktion

4.3 Funktionale Programmiersprachen

Diese Klasse von Programmiersprachen basiert, wie der Name schon sagt, auf Funktionen. Zu ihr gehören Sprachen wie Haskell und Common LISP. Jeder Vorgang wird als Funktionsaufruf angesehen. Es gibt keine Variablen, weder global noch lokal. Alle benötigten Werte müssen als Parameter an die Funktion übergeben werden. Da es keine Variablen gibt, gibt es auch keine Zuweisungen, wie sie in imperativen und objektorientierten Programmiersprachen möglich sind. Die Ablaufsteuerung erfolgt in funktionalen Programmiersprachen über Rekursion. Funktionale Programmiersprachen haben ihre Vorteile in einer einfach lesbaren und zu erlernenden Sprache. Die Programme sind im Allgemeinen auch weniger anfällig für Fehler, weil sie zum einen weniger logische Fehler zulassen (Seiteneffekte sind nicht möglich) und zum anderen werden durch den Compiler bzw. Interpreter bereits viele Fehlerquellen (z.B. bei Typkonvertierungen und Speicherverwaltung) beseitigt. Weitere Vorteile liegen auch hier in der hohen Wartbarkeit, die durch Modularität erreicht wird. Als letzter Punkt ist hier auch noch die Parallelität anzuführen. Der Interpreter kann selbst entscheiden, in welcher Reihenfolge die Funktionen ausgeführt werden und kann so auch parallele Funktionen selbständig auf mehrere Prozessoren verteilen, soweit das System dies zulässt. Nachteile funktionaler Programmiersprachen sind darin zu sehen, dass das Programm meist von einem Interpreter abgearbeitet wird. Dieser sucht sich seinen Weg zu Lösung des Problems selbst. Der Interpreter entscheidet sebständig, in welcher Reihenfolge Funktionen ausgewertet werden. Der Programmierer hat somit wenig Einfluss auf die einzelnen Abarbeitungsschritte und somit auch nicht die Möglichkeit, wie bei imperativen Programmiersprachen zu optimieren. Der Laufzeitnachteil lässt sich durch den Einsatz eines Compilers vermindern, da hier das Programm auf die jeweilige Plattform zugeschnitten wird. Es bleibt dennoch ein Geschwindigkeitnachteil gegenüber den imperativen Sprachen. [1]

4.4 Logische Programmiersprachen

Die letzte Klasse von Programmersprachen, die hier noch Erwähnung finden soll, ist die der logischen Programmiersprachen. Bekanntester Vertreter in dieser Kategorie ist die Sprache Prolog. Logische Sprachen orientieren sich stark an der mathematischen Aussagenlogik. Daher sind auch die grundlegenden Elemente der logischen Programmiersprachen Objekte, Eigenschaften und Beziehungen. Die Objekte stellen die elementare Datenstruktur dar. Das könnte zum Beispiel eine Person sein. Die Objekte können Eigenschaften besitzen, so zum Beispiel die Eigenschaft X ist Vater von Y. Eine Beziehnung könnte folgendermaßen definiert sein: X ist Großvater von Z, genau dann, wenn es ein Y gibt, so dass gilt: Y ist Vater von X und Z ist Vater von Y. Logische Programmiersprachen arbeiten

meistens mit einem interaktiven Interpreter. Dem Interpreter können Wahrheitswertfragen gestellt werden, die dann mit Hilfe der gegebenen Eigenschaften und Beziehnugen gelöst werden.

An einem konkreten Beispiel sähe das folgendermaßen aus:

- Max ist eine Person.
- Karl ist eine Person.
- Tom ist eine Person.
- Karl ist Vater von Max.
- Tom ist Vater von Karl.
- Ist Tom Großvater von Max?

Die Frage letzte Frage würde vom Interpreter dann mit ja beantwortet werden.

Vorteile der logischen Programmiersprachen sind vor allem in der kurzen, prägnanten Ausdrucksform zu sehen. Durch die Anlehnung an mathematische Aussagenlogik bietet sie eine einfache Verständlichkeit und Nachvollziebarkeit. Das Wissen, das in einem Programm enthalten ist, wird plattformunabhängig gehalten. Alles was auf einer anderen Plattform nötig ist, ist der entsprechende Interpreter. Die Nachteile sind ähnlich denen der funktionalen Programmiersprachen. Da sich der Computer versucht an Hand der ihm gegebenen Informationen versucht, die Fragen zu beantworten, hat der Programmierer keinen Einfluss auf die Abarbeitung der einzelnen Schritte und somit auch wenig Einfluss auf die Geschwindigkeit des Programms.

5 Einführung in Haskell

Haskell [2] gilt als reine funktionale Programmiersprache. Sie enthält keine Elemente von imperativen Programmiersprachen. Haskell baut auf dem Lambda-Kalkül auf, eine formale Sprachdefinition die den Umgang mit Funktionen erleichtert. Haskell verdankt seinem Namen dem Amerikanischen Logiker und Mathematiker Haskell Brooks Curry (1900-1982).

Man kann zwischen Funktionalen Programmiersprachen und SQL (Standard Quering Language) Parallelen ziehen. So sagt man nur *was* man erreichen möchte, aber nicht *wie* man zum Ergebnis gelangt. So auch bei Haskell.

6 Geschichte

In den 80er Jahren wurden viele funktionale Programmiersprachen entwickelt, darunter zum Beispiel Erlang und Miranda. Aufgrund der Vielzahl von funktionalen Programmiersprachen wurde der Ruf nach einem Standard immer lauter.

So wurde 1987 ein Haskell Komitee gegründet, das 1990 die erste Sprachdefinition herausbrachte (Haskell 1.0). 1992 folgt dann Haskell 1.2 und 1995 wurde der Vorschlag für Haskell 1.3 eingereicht.

Haskell 98 wurde im Jahre 1998 verabschiedet und gilt als fertig. Es werden keine weiteren Funktionalitäten hinzugefügt.

Haskell 2 wird jedoch wohl bald folgen. Neben dem Standard Haskell gibt es auch Gruppen im Internet die zum Beispiel Haskell++ entwickeln, eine Haskell Version mit objektorientierten Eigenschaften.

7 Eigenschaften von Haskell

Wie schon aus der Beschreibung von funktionalen Programmiersprachen hervorgegangen ist besitzt Haskell keinen linearen Ablauf im Programm. So kann man zum Beispiel die Fakultät auf zwei Weisen definieren [3]:

```
fac :: Int -> Int
Fac 0 = 1
Fac n = n * Fac(n-1)
```

oder:

```
Fac n = n* Fac(n-1)
Fac 0 = 1
```

Hieraus resultiert auch direkt, dass die Hauptaufgabe beim Compiler [5] bzw. beim Interpreter liegt. Diese müssen dann entscheiden wie die mathematische Definition der Fakultät programmtechnisch umgesetzt werden soll.

Ebenso wird die komplette Speicherverwaltung von Haskell vom Compiler übernommen. Dies kann teilweise dazu führen das mehr Speicher alockiert wird als wenn man ein C Programm nutzt.

Unter LazyEvaluation versteht man die Möglichkeit zur Laufzeit zu entscheiden welche Parameter einer Funktion genutzt bzw. gebraucht werden. Nur benötigte

Parameter oder Funktionen werden dann auch tatsäechlich berechnet. Weiterhin ist Haskell case-sensitive, das heißt auf Groß- und Kleinschreibung ist zu achten.

Haskell Programme lassen sich in Module aufteilen um eine gewisse Modularität zu gewährleisten.

8 Sprachkonstrukte in Haskell

Der Haskellstandard bietet mehrere Datenstrukturen. Als Benutzer hat man selber die Möglichkeit weitere Datenstrukturen zu definieren (z.B. rekursive Datenstrukturen). In Haskell gibt es unter anderem folgende Datenstrukturen: Integer, Char, Funktionen, Listen und Tupel. Einem Ausdruck muss nicht zwingend ein Datentypen zugewiesen werden, es wird jedoch empfohlen, da sonst die Mechanismen der Typüberprüfung des Compilers nicht funktionieren. Die Zuweisung eines Datentyps sieht wie folgt aus:

```
funktion :: Integer -> Integer
liste :: [Integer]
tupel :: (Integer, [Integer]) -> Char
```

Hier wurde zunächst eine Funktion definiert, die einen Parameter vom Typ Integer hat und als Ergebnis einen Integer liefert. Die zweite Deklaration beschreibt eine Liste aus Integern und die dritte eine Funktion, dessen Parameter ein Tupel aus einem Integer und einer Liste von Integern besteht und die ein Char zum Ergebnis hat.

Haskell unterstützt logischerweise Funktionen. Im allgemeinen versteht man unter einer Funktion eine Folge von Gleichungen. Eine Funktion wird festgelegt durch:

```
<Funktionsname> <Parameter> = <Funktionsrumpf>
```

Bei den Funktionen wird das sogenannte pattern-matching unterstützt. Man kann durch die Festlegung bestimmter Muster beeinflussen, welche der Funktionsdefinitionen verwendet werden. Das bietet die Möglichkeit, Bedingungen zu prüfen.

Natürlich gibt es auch mehrere Operatoren in Haskell. Neben den mathematischen Operatoren wie +, -, /, * sind unter anderem folgende noch definiert worden (ohne, dass Anspruch auf Vollständigkeit erhoben wird):

- [1] ++ [2] (concatenate) zum Zusammenfügen von Listen
- x <− xs (Element von) beschreibt die Zugehörigkeit zu einer Liste (als Menge gesehen), ganz wie das mathematische Symbol ∈

- | (hat Eigenschaft)
- x:xs (infix operator) fügt ebenfalls Elemente zu einer Liste zusammen. Der Operator wird beim pattern-matching verwendet.
- −− leitet Kommentarzeilen ein

9 Beispiele

9.1 Quicksort

Quicksort ist ein Sortieralgorithmus, der das Konzept von divide et impera verwendet. Einfach formuliert besagt der Algorithmus folgendes: Nimm ein Element der Liste, so dass das Element an der richtigen Position in der Liste steht. Sorge dafür, dass links von dem Element nur die kleineren und rechts vom Element nur die größeren Elemente stehen. Sortiere die linke und die rechte Seite.

In Haskell kann diese Formulierung fast eins zu eins übernommen werden. Der entsprechende Quelltext sieht folgendermaßen aus [4]:

```
qsort :: [Integer] -> [Integer]
qsort [] = []
qsort (x:xs) = qsort linke_Liste ++ x ++ qsort rechte_Liste
               where
                   linke_Liste  [y | y <- xy, y < x]
                   rechte_Liste [y | y <- xy, y >= x]
```

Man stellt fest, dass die Übersetzung des Quelltextes ins Deutsche fast der Beschreibung des Quicksort-Algorithmus entspricht. Hier lässt sich die einfache Verständlichkeit der Quelltexte sehr gut beobachten.

qsort ist eine Funktion, die als einziges Argument eine Liste von Integern erhält und als Ergebnis eine Liste von Integern zurückgibt. Wird die Funktion mit der leeren Liste aufgerufen ist das Ergebnis die leere Liste. Wird die Funktion mit einer nicht leeren Liste aufgerufen, so besteht die Liste aus dem ersten Element x und dem Rest xs (pattern-matching). Der Rest kann auch die leere Liste sein und somit wird die Rekursion irgendwann abgebrochen. Die Funktion gibt bei diesem Aufruf eine zusammengesetzte Liste zurück. Diese besteht aus der sortierten linken Liste, dem x und der sortierten rechten Liste. Nachfolgend werden noch die Eigenschaften der linken und rechten Liste genannt. Die linke Liste ist die Liste, die alle Elemente y enthält, die die Eigenschaft haben in xs enthalten zu sein und kleiner als x sind. Für die rechte Liste gilt analog, dass die Elemente in xs enthalten sind und größer oder gleich x sind.

9.2 Die Türme von Hanoi

Als weiterer einfacher Algorithmus, der auch die Ausgabe von Informationen enthält sei noch das Problem der Türme von Hanoi beschrieben. Dieses Spiel funktioniert so, dass man drei Stapel hat, auf die man Scheiben unterschiedlicher Größe plazieren kann. Dabei muss beachtet werden, dass keine größere Scheibe auf einer kleineren liegt. Anfangs befinden sich n Scheiben auf dem ersten Stapel. Ziel ist es, die Scheiben auf einen der anderen beiden Stapel zu verschieben. Der Grundgedanke der Lösungsstrategie ist, zunächst den Stapel von n-1 Scheiben auf den übriggebliebenen Platz zu verschieben. Das Verschieben von der letzten verbleibenden Scheibe ist trivial und benötigt keine weiteren Schritte. Anschließend wird der Stapel mit n-1 Scheiben auf den Zielstapel verschoben. Das Programmlisting sieht genau diese Vorgehensweise vor und sollte keiner weiteren Erklärung bedürfen. Die Funktion besteht wieder aus einer Abbruchbedingung für den Fall der einzelnen Scheibe und der Abarbeitung der anderen Scheiben. Als Parameter ist ein Tripel vorgesehen, dass die Anzahl der Scheiben auf dem Quellstapel, der Nummer des Quellstapels und der Nummer des Zielstapels besteht. Die Berechnung des freien Stapels sieht vielleicht im ersten Moment verwirrend aus, doch wird man bei genauerem Überlegen feststellen, dass die Summe der drei Stapelnummern genau sechs ist und bei der Subtraktion der beiden verwendeten Stapel von dieser Summe bleibt genau der noch freie Stapel übrig.

```
hanoi (1,source,dest) = print("von ",source," nach ",dest)
hanoi (n,source,dest) = do hanoi(n-1,source,6-(source+dest))
                           print("von ",source," nach ",dest)
                           hanoi(n-1,6-(source+dest),dest)
```

10 Vergleich zu Common LISP

LISP ist eine weitere funktionale Progrmmiersprache. Der Name ist Abkürzung für **Lis**t **P**rocessing. Alle Datenstrukturen in LISP sind aus Atomen (Skalarwerte) und Listen zusammengesetzt. Dabei ist es möglich, auch die Listen beliebig oft zu schachteln. Die wesentlichsten Operatoren, die in LISP verwendet werden sind:

- car (gibt das erste Element einer Liste zurück),
- cdr (gibt die Restliste (ohne das erste Element) zurück),
- cons (verknüpft zwei Listen),
- quote (verhindert Auswertung),

- eq (Test auf Gleichheit),
- cond (bedingte Ausführung) und ein
- Mechanismus zur Funktionsdefinition.

Mit diesem Funktionsumfang lassen sich alle weiteren LISP-Funktionen implementieren.

Anhand der Aufstellung lassen sich bereits einige Gemeinsamkeiten zwischen den beiden Sprachen erkennen. Beide bieten einfache Möglichkeiten zur Verarbeitung von Listen: das Zusammenfügen und Trennen in erstes Element und Rest.

Der größte Unterschied zwischen Haskell und Common LISP besteht darin, dass LISP imperative Programmteile unterstützt. LISP ist als keine reine funktionale Programmiersprache. Das hat zum einen den Vorteil, dass das erlernen einfacher wird, wenn man den imperativen Stil gewöhnt ist. Andererseits macht es die Vorteile der funktionalen Programmierung zunichte, wenn sie nicht genutzt werden.

Literatur

[1] Eisenbach, Susam, *Functional Programming* (Chichester: John Wiley & Sons, 1987).

[2] Haskell Homepage: `www.haskell.org`

[3] Fakultätsfunktionen: `http://www.willamette.edu/~fruehr/haskell/evolution.html`

[4] Quicksort: `http://haskell.org/aboutHaskell.html`

[5] Glasgow Haskell Compiler Homepage: `www.haskell.org/ghc`

[6] `www.wikipedia.de`

[7] `http://irb.cs.tu-berlin.de/~zuse/history/Programmiersprachen_2_large.gif`