Grundkonzepte verschiedener Programmiersprachen: ein Überblick

Torsten Anders
Studio für elektroakustische Musik (SeaM)
Hochschule für Musik Franz Liszt, Weimar
torsten.anders@hfm.uni-weimar.de

Zusammenfassung

Alle Aufgaben, die sich mit dem Computer lösen lassen, lassen sich auch in verschiedenen Programmiersprachen und mit verschiedenen Programmierkonzepten lösen. Aber für verschiedene Aufgaben sind manche Sprachen angemessener als andere.
Dieser Text versucht, verschiedene Denkansätze der Programmierung und damit auch Denkansätze verschiedener Programmiersprachen gut verständlichen vorzustellen. Der „Wortschatz“ und die Syntax verschiedener Sprachen wird dabei natürlich nur gestreift.

Inhaltsverzeichnis

---

1 Prozedurale Programmierung

typische Sprachvertreter

Pascal, C, Basic, Ada

Musik-Programmiersysteme

Max, Super Collider

Grundelemente

Die prozedurale Programmierung ist das am häufigsten verwendete Sprachkonzept. Die besonders oft eigesetzte Sprache C ist beispielsweise *die* Sprache für die Programmierung von Betriebssystemen[1].

Prozedurale Sprachen sind häufig kompilierende Sprachen, d.h. ein Programm muß erst fertig geschrieben werden, bevor es in eine Folge von maschinenverständliche Anweisungen übersetzt werden kann. Die Übersetzung erfolgt durch den *Compiler* (der Übersetzer) unter zuhilfenahme des *Linker*, der die Funktionalität von im Programm verwendeten Bibliotheken in das Programm einfügt. Anschließend kann mit Hilfe des *Debugger* (Entwanzer) eine eigentlich immer notwendige Fehlersuche erfolgen. Wenn auch diese etwas komplex anmutende Vorgehensweise von modernen *Entwicklungsumgebungen* weitgehend automatisch vollzogen wird, ist die Arbeitweise immer noch: Erst muß das Programm soweit fertig geschrieben werden, daß es kompiliert werden kann, dabei treten häufig Fehlermeldungen auf. Anschließend können die einzelnen Funktionalitäten des Programms getestet werden.

[1] Sowohl Windows als auch MacOS (was früher in Pascal programmiert war) sind in C programmiert. Ursprünglich entwickelt wurde C Anfang der 70-er Jahre für UNIX.

Beim prozeduralen Programmieren wird deutlich unterschieden zwischen *Prozeduren* (Programmen) und *Daten*, auf die die Prozeduren angewendet werden.

Verschiedene *Datentypen*[2] enthalten die Daten des Programms, diese können *Variablen* zugewiesen werden. Eine Variable ist ein im Programm mit Datentyp deklarierter Name, dem Daten seines Datentyps zugewiesen werden können. Die Zuweisung erfolg in C durch den `=`[3] Operator, der nicht der Vergleichsoperator ist:

```
int var = 10;
```
[4]

An Daten können verschiedene vordefinierte Prozeduren vollzogen werden wie read, print, mathematische Operationen...

```
var = var + 1;
```
[5]

Dabei ist der Ablauf des Programms ist grundsätzlich *sequenziell*, die *Anweisungen* werden in der Reihenfolge ihres Auftretens abgearbeitet.

Jedoch kann dieser Fluß bedingt sein:

```
if (var==1)
  var = var + 1;
```
[6]

Es sind *Verzweigungen*

```
if (var==1)
  var = var + 1;
else
  var = var - 1;
```
[7]

und *Schleifen* möglich

```
int counter = 0;
```

[2]Für eine effiziente Programmierung ist eine genaue Festlegung durch den Programmierer, welche Variable im Programm wieviel Speicher benötigen wird und wie der binär gespeicherte Wert zu interpretieren ist, erforderlich. Jeder Variablen ist deshalb bei ihrer *Deklaration* ein Datentyp zuzuordnen. Verschiedene Datentypen sind *Integer* (Ganzzahl) und *Float* (Fließkommazahl) jeweils verschiedener Größe, *Character* (Zeichen), *String* (Zeichenkette) und *Boolean*, d.h. *true* (logisch wahr) oder *false* (logisch falsch). Aus einfachen Datentypen lassen sich auch komplexere Datentypen zusammensetzen. Ein *Array* (Datenfeld) ist ein häufig verwendeter zusammengesetzter Datentyp.

[3]Die Gleichheit dagegen testet in C der `==` Operator. In Pascal hat man sich deshalb entschlossen `=` als Vergleichsoperator festzulegen, der Zuweisungsoperator ist dort `=:`.

[4]Weise der als Interger deklarierten Variablen `var` den Ganzzahlwert `10` zu. Das Semikolon am Ende der Zeile signalisiert dem Compiler: hier ist die Anweisung zuende. (In Pascal signalisiert es Trennung zwischen zwei Anweisungen, in C den Abschluß einer Anweisung). Bei einem Zuweisungsoperator steht vor dem Operator der Variablenname und danach der zugewiesene Wert, der auch ein Ausdruck sein kann.
Alle Beispiele werden in C gegeben.

[5]Weise der Variablen `var` als neuen Wert ihren `um 1` erhöhten Wert zu.

[6]Wenn die Variable `var` den Wert `1` enthält, dann und nur dann erhöhe sie um `1`.

[7]Wenn die Variable `var` den Wert `1` enthält, dann erhöhe sie um `1`, andernfalls ziehe `1` von ihr ab.

```
while (counter < 5)[8]
  {[9]
  println(counter);[10]
  counter++;[11]
  }
```

Es könne eigene Prozeduren definiert werden

```
int summe(int x, int y)
{
    return x + y;
}[12]
```

die im weiteren Programm aufgerufen werden können[13] . Ein Programm beginnt beim Programmstart immer mit der Einstiegsprozedur, in C ist das `main`.

In Sprachen wie C ist eine sehr maschinennahe Programmierung möglich aber auch nötig. Dadurch erreicht der Programmierer sehr effiziente Programme, muß sich aber mit vielen maschinennahen Details beschäftigen. Dazu gehört - wie schon gesehen - die genaue Unterscheidung von Ganz- und Fließkommazahlen und anderen Datentypen. Aber auch das Referenzieren von genauen Speicheradressen, die Variablen zugewiesen werden können, sogenannte *Pointer* (Zeiger) auf Bereiche im im dynamischen[14] Speichen (dazu gehören der *Heap*[15] und der *Stack*[16]) müssen vom Programmierer beherrscht werden. Vor allem müssen die auf diese Weise im Speicher belegten Adressen wieder explizit im Code freigegeben werden, wenn sie nicht mehr benötigt werden - eine sehr fehleranfällige Angelegenheit.

Für besonders rechenintensive Programme, z.B. zur Soundsynthese, ist deshalb ein Programmieren in C (oder C++) erforderlich, weil z.Z. in keiner Sprache effizienter programmiert werden kann. Für sehr komplexe Programme, z.B. zur Komposition, ist dagegen eine höhere Sprache wie z.B. Lisp zu empfehlen.

2 Objektorientierte Programmierung

typische Sprachvertreter

Smalltalk erwirkte den Durchbruch für objektorientierte Programmierung, eine rein objektorientierte Sprache, wird allerdings im Vergleich zu C++ und Java relativ wenig verwendet

[8] Solange der Wert der Variablen `counter` kleiner ist als `1` wiederhole die `while` Schleife.

[9] Die geschweiften Klammern umschließen einen *Block*, alle Anweisungen innehalb dieses Blocks werden sequenziell abgearbeitet. Der Block ist wichtig, denn er sieht dabei für die `while` Schleife wie eine einzige Anweisung aus. Die Schleife erwartet in ihrem Körper nur eine Anweisung.

[10] Gebe den Wert von `counter` aus.

[11] Erhöhe den Wert von `counter` um `1`, d.h. *inkrementiere* `counter`.

[12] Definiert die Prozedur `summe`, die zwei Ganzzahlen als Argument erwartet und die Summe der beiden (was ebenfalls eine Ganzzahl sein muß) zurückgibt.

[13] d.h. nach der Definition und nicht vorher, im Gegensatz zu Sprachen wie Lisp

[14] Der zur Laufzeit des Programms anzufordernde Speicher, im Gegensatz zum statischen Speicher, der schon vom Compiler mit z.B. den Namen von Variablen belegt wird - deren Wert wird eben erst zur Laufzeit zugewiesen und also im dynamischen Speicher verwaltet.

[15] Haufenspeicher: eine Speicherform bei der ich per Adresse genau auf bestimmte Speicherzellen zugreifen kann.

[16] Stapelspeicher: ein Speicher in den man Werte quasi aufstapelt um sie in umgekehrter Reihenfolge wieder auszulesen.

C++ eine objektorientierte Erweiterung von C

Java ähnlich C++, plattformunabhängig durch Konzept der Virtuellen Maschine

CLOS eine objektorientierte Erweiterung von Common Lisp

Musik-Programmiersysteme

Max, Super Collider, Patch Work, Common Music

Grundelemente

Objektorientierten Programmierung (OOP) ist ein schon seit den 70-er Jahren in der Wissensmodelierung verwendetes Verfahren, daß sich in den letzten Jahren für eigentlich alle Anwendungsbereiche der Programmierung durchgesetzt hat.

Gegenüber der rein prozeduralen Programmierung bestehen die Vorteile der Objektorientierung vor allem darin, daß größere Programmierprojekte leichter überschaubar sind, leichter erweitert werden können und das eine Wiederverwendung von Code in anderen Projekten einfacher ist.

Die leichtere Überschaubarkeit wird durch ein Modell erreicht, das unserer Welterfahrung näher ist, als die rein prozedurale Denkweise. Grundelemente sind nicht verschiedene Datentypen, an denen verschiedene Prozeduren vollzogen werden. Grundelement ist das Objekt, in dem sowohl Daten, als auch Programme *gekapselt*[17] sind.

Das Objekt `meinFahrrad` beispielsweise enthalte Angaben über seine Farbe und den aktuellen Gang, je nach Sprache *Attributvariablen*, *Slot* u.ä. genannt. Außerdem beschreiben *Methoden* (Programme) was der Schaltvorgang oder das Treten eines Fahrrads bewirken.

Das Objekt `meinFahrrad` ist aber nur eine *Instanz* der *Klasse* `Fahrrad`. Die Klasse enthält die Angaben, welche Eigenschaften in einem Objekt gespeichert werden können, und in der Klasse sind die Methoden der Klasse definiert.

In einem Programm, das die Definition der Klasse `Fahrrad` enthält, kann das Objekt `meinFahrrad` instanziiert und dieses Objekt quasi als ein zusammengesetzter Datentyp im Programm verwenden werden. Das Objekt „weiß“ aber auch gleich, wie diese intern gespeicherten Daten verwendet werden. Zum Schalten muß im Programm nur die entsprechende Methode der Klasse `Fahrrad` (z.B. eine Methode `Hochschalten`) mit der Instanz `meinFahrrad` aufgerufen werden - wie dieser Schaltvorgang im einzelnen erfolgt, wie sich dabei beispielsweise der Wert des aktuellen Ganges ändert, braucht der Programmierer beim Methodenaufruf nicht zu wissen.

Durch dieses Konzept können Objekte verschiedener Klassen auf denselben Methodennamen verschieden reagieren. Eine Instanz der Klasse `Fahrrad` reagiert beispielsweise anders auf die Methode `Treten`, als eine Instanz der Klasse `Ball` oder gar der Klasse `Hund`.

Manche objektorientierten Programmiersprachen (C++) erlauben ein *Überladen* von Methoden. Das bedeutet, daß mehrere Methoden mit dem gleiche Namen in der selben Klasse definiert werden. Durch die Zahl oder den Datentyp der Argumente wird dann diese oder jene Methode aufgerufen.

Objektorientierte Programmierung hilft also zu einer besseren Übersicht und Pflege von Programmen. Weil die Definition einzelner Objektklassen so gut vom übrigen Programm gekapselt ist, läßt sie sich

[17] Kapselung bzw. *Abstraktion* meint das Verbergen von Details - z.B. des Speicherungsformats einer Datenbank. Der Programmierer hat den Überblick zu behalten, die Verwaltung von Details kann er Programmen überlassen. Im Beispiel wären das Prozeduren zum Anlegen, Ändern und Auslesen von Feldern der Datenbank, die sich leichter merken lassen, als das genaue Format eben der Datenbank.

auch gut extrahieren und damit weiterverwenden. So sind auch fertige Programmkomponenten in Bibliotheken vertreibbar[18].

Durch das Konzept der *Vererbung* ist ein Programm aber auch leichter erweiterbar, als eine rein prozedurale Programmierung erlauben würde: Eine Klasse kann von anderen Klassen Eigenschaften ererben, die Kindklasse ist dabei in der Regel stärker spezialisiert, als die Elternklasse. Beispielsweise könnten die Klasse `Mountainbike` und `Tandem` Spezialisierungen mit zusätzlichen oder erweiterten Attributvariablen und Methoden der Klasse `Fahrrad` darstellen. Grundsätzlich verdeckt dabei eine stärker spezialisierte Methode die Methode der Elternklasse mit dem gleichen Namen.

Manche Sprachen (C++, CLOS) erlauben die *Mehrfachvererbung*: eine Klasse kann mehr als eine Elternklasse haben. Andere Sprachen (Java) verzichten aus Sicherheitsgründen auf diese Möglichkeit, denn in größeren Objekthierachien kann schwer überschaut werden, welche Methode aufgerufen wird, wenn diese nicht im entsprechenden Objekt, aber in verschiedenen Elternklassen gleichzeitig definiert wird.

3 Funktionale Programmierung

typische Sprachvertreter

Common Lisp eine sehr umfangreiche Sprache

Scheme ein überschaubarer Lispdialekt, häufig zur Lehre und auch als Skripsprache eingesetzt

Musik-Programmiersysteme

Patch Work und Common Music sind beide in Lisp implementiert und erweiterbar

Modalys verwendet die Syntax und den Sprachvorrat von Scheme

Super Collider beinhaltet viele Lispfunktionen

Grundkonzepte von Lisp

Lisp ist eine Sprache zur Symbolverarbeitung und wird deshalb besonders für Aufgaben der Künstlichen Intelligenz, z.B. für Wissensbasierte Systeme, eingesetzt. Lisp ist eine der ältesten heute noch verwendeten Programmiersprachen.

Lisp ist traditionell eine *interpretierende*[19] Sprache: der Lispinterpreter und der Benutzer kommunizieren miteinander durch die *read-eval-loop*[20] des Interpreters. Wenn auch moderne Implemetationen von Lisp aus Gründen der Performance selbtverständich kompilieren, hat sich diese Grundbedienung nicht geändert.

[18] Beispielsweise sind Bibliotheken zur Entwicklung grafischer Oberflächen häufig objektorientiert: eine Klasse `Fenster` hat Eigenschaften wie `Position`, `Größe`, `Hintergrundfarbe` des Fensters usw. Methoden, die durch Signale wie Mausklicks oder Dragging mit der Maus *Messages* erhalten, können diese Eigenschaften ändern.

[19] Im Gegensatz zum Compiler übersetzt ein Interpreter den Programmcode erst zur Laufzeit. Der Verlust an Effizienz bei der Ausführung (eine mehrfach verwendete Programmzeile muß immer wieder neu übersetzt werden) wurde wegen der schnellen Änderbarkeit des Programmes in Kauf genommen.

[20] Der Benutzer gibt Lispcode ein, den der Interpreter liest (read). Der Interpreter ermittelt den Wert der Eingabe (eval), gibt diesen aus und wartet auf die nächste Eingabe (es handelt sich also zum eine Schleife oder loop).
In der read-eval-loop kann grundsätzlich jeder Lispcode eingegeben werden, der auch in einem Lispprogramm verwendet werden kann. Hier ist also jegliches Ausprobieren von Code möglich, aber auch fertige Lispprogramme verwenden die read-eval-loop als Schnittstelle zum Benutzer.

Im Gegensatz zu Sprachen wie C oder Pascal vollzieht Lisp eine automatische Datentypenkonvertierung. Dadurch wird der Programmierer von der fehlerträchtigen Unterschiedung zwischen z.B. Ganzzahlen- und Fließkommazahlenrechnung verschont. Ebensowenig braucht sich der Programmierer um das ebenfalls fehlerträchtige Belegen oder Freigeben von Speicher zu kümmern. Nicht mehr referenzierten Speicher gibt die von Zeit zu Zeit einsetzende *Garbagecollection* frei. Beide Verfahren aber kosten etwas Performance - dafür hat der Programmierer den Kopf frei für die eigentliche Programmieraufgabe.

Lisp besitzt eine grundsätzlich sehr einfache und einheitliche Syntax - Grundelemente von Lisp sind *Atome* in *Listen*[21]:

```
> (das sind 7 atome in einer liste)
```
[22]

Es gibt keine deutliche Trennung zwischen *Prozeduren* (auch *Funktionen*[23] genannt) und Daten. Es wird nur durch die *Prefixnotation* unterschieden: die Prozedur am Kopf der Liste wird auf die übrigen Atome (Daten) der Liste angewendet:

```
> (+ 1 2 3 4 5)

15
```

Symbole sind mit Namen identifizierbare Variablen, die einen Wert (Zahl, Wort, Liste...) enthalten, aber auch Namen von Prozeduren sind Symbole:

```
> (define zahl 1)
```
[24]

Die Wertermittlung eines Symbols ebenso wie die Auswertung einer Liste mit einer Prozedur/Funktion an erster Stelle, also eines Programmes, gibt immer einen Wert zurück[25]. Diesen Vorgang der Wertermittlung nennt man *Evaluation.* *Side Effects*[26] können zusätzlich erfolgen (display, Wertzuweisung...):

```
> zahl
```
[27]

```
1
```

[21] Listen sind durch runde Klammern eingeschlossene Symbole. Listen können beliebig verschachtelt sein. Es gibt viele verschiedene Funktionen, um Listen auszuwerten oder zu verändern.
Es gibt deutlich mehr auf Lisp basierende Kompositionssysteme als für irgend eine andere Sprache, da eine verschachtelte Liste besser eine Partitur darstellen kann als ein Array oder gar einzelne Zahlen und weil die Bearbeitungsmöglichkeiten von Listen in Lisp so vielfältig sind.

[22] Alle Beispiele verwenden Scheme. Die Eingaben des Benutzer sind durch den vorangehenden Promt der read-eval-loop gekennzeichnet, anderes sind Rückgaben von Lisp. Der Prompt ist stark implementationsabhängig.

[23] Sie verhalten sich ähnlich mathematischer Funktionen denn sie geben einen Wert zurück der von den Eingabewerten abhängig ist.
Tatsächlich wurde Lisp anfangs für mathematische Anwendungen entwickelt. Sehr genaue Darstellungen von Zahlen z.B. als Verhältnis oder als Komplexe Zahlen unterscheiden Lisp noch heute von den meisten anderen Sprachen.

[24] Weise dem Symbol `zahl` den Wert `1` zu.

[25] Der zurückgegebene Wert kann u.a. ein Symbol, eine Zahl oder eine Liste sein. Durch die Prefixnotation ist aber auch das Zurückgeben von ganzen Programmen möglich, da auch ein Programm eine Liste ist.

[26] *Zusätzliche Effekte* sind alles, was sich permanent im Programm ändert. Funktionen sind genaugenommen Prozeduren ohne side effects.

[27] Evaluiere den Wert von `zahl`.

Der Programmfluß ist grundsätzlich nicht sequenziell, sondern erfolgt durch gegenseitigen oder geschachtelten Aufruf von Prozeduren. Die aufgerufenen Prozeduren übergeben dabei ihre Rückgabewerte der aufrufenden, die diese dann als ihre Argumente weiterverwendet. Die Autoren von [HW94, Seite 29 ff] vergleichen dies anschaulich mit einer Eimerkette, das jeweilige Ergebnis wird weitergereicht. Die `+` Prozedur übergibt ihr Ergebnis der `display` Prozedur, die dieses Ergebnis ausdruckt:

```
> (display (+ 1 2))[28]
3
()
```

So wie einem Symbol eine Zahl als Wert zugewiesen werden kann, kann auch eine neue Funktion als Wert zuwiesen werden. Eine Funktionsdefinition besteht aus einer Kombination von bereits definierten Funktionen, die häufiger benötigt wird. Diese Kombination wird also in einer neuen Funktion *gekapselt*.

```
> (define (quadrat x)
>    (* x x))[29]

quadrat
> (quadrat 5)

25
```

Definierte Funktionen können wiederum von anderen Funktionen verwendet werden. Einzelne Funktionen sollten möglichst klein sein, um ihre Definition leicht überblicken zu können. Ein klassisches Lispprogramm besteht vor allem aus einer Anzahl von Funktionsdefinitionen, die in der read-eval-loop sofort ausgetestet werden können.

Zwei Grundsätzliche Strategien zur Entwicklung größerer Lispprogramme werden unterschieden: bei der Programmierung *bottom to top* werden zunächst die grundlegenden Funktionen geschieben, die in komplizierteren zusammengesetzten Funktionen verwendet werden. Bei der Programmierung *top to bottom* wird mit der komplexen Hauptfunktion begonnen, Hilfsfunktionen werden verwendet, als wären sie schon geschrieben. Ist auf diese Weise genau bekannt geworden, was eine Hilfsfunktion leisten muß, kann auch sie programmiert werden, wobei wiederum später zu schreibende Hilfsfunktionen eingesetzt werden können.

Bestimmte Lispfunktionen erlauben sequenzielles Programmien, andere Konditionen, Verzweigungen und Schleifen - es stehen also die Grundkonstrukte der prozeduralen Programmierung zur Verfügung. Da diese Funktionen von der üblichen Regel (erstes Element ist Funktion die auf Rest der Liste angewendet wird) abweichen, werden sie *Special Forms* genannt.

Typische Kontrollstrukturen für Lisp arbeiten eine Datenliste ab: *iterativ* durch eine Schleife (ähnlich while u.a. in C oder Pascal) oder *rekursiv* durch Selbstaufruf der Prozedur für das nächste Datenelement.

[28] Zeige das Ergebnis des Ausdrucks $1 + 2$.
Zusätzlich gibt die Schemefunktion `display` auch einen Wert zurück, nämlich die leere Liste `()`, was in Lisp auch NIL (nichts) und *logisch falsch* entspricht.
(Die Common Lisp Funktion `print` dagegen gibt den ausgedruckten Wert zurück, darum würde in der Ausgabe `3` zweimal erscheinen.)

[29] Weise dem Symbol `quadrat` mit dem Argument `x` die Funktion $quadrat(x) = x * x$ zu.

```
> (define (fibonacci n)
    (if (or (= n 0) (= n 1))[30]
    1[31]
   (+ (fibonacci (- n 1))
      (fibonacci (- n 2)))))[32]
```

Wiederum durch die nur unvollständige Trennung zwischen Daten und Programmen in Lisp ist auch die Übergabe von Funktionen als Argumenten möglich. Dieses als *Higher Order Programming* bekannte Konzept erlaubt beispielsweise die Anwendung einer Funktion nacheinander auf alle Glieder einer Liste. Diese Kontrollstruktur wird *Mapping* genannt:

```
> (map quadrat '(1 2 3 4 5))[33]

(1 4 9 16 25)
```

Eine weitere Möglichkeit ist, eine anonyme Funktion direkt innehalb einer Funktion zu definieren. Die Idee dazu stammt von dem Mathematiker Alonzo Church und wird *Lambdakalkül* genannt. Das Lambdakalkül stellt die theoretische Grundlage von Lisp überhaupt dar:

```
> (map (lambda (x) (+ x x)) '(1 2 3 4 5))[34]

(2 4 6 8 10)
```

4 Deklarative Programmierung, Logikprogrammierung

typischer Sprachvertreter

Prolog

Musik-Programmiersysteme

PWConstraints und PWSituation zwei Bibliotheken in Patch Work

Grundkonzepte von Prolog

Auch Prolog ist eine Sprache zur Symbolverarbeitung, die andere wichtige Sprache der Künstlichen Intelligenz. Und wie Lisp ist auch Prolog theoretisch fundiert: Grundkonzepte bilden die *Prädikatenlogik* (erster Ordung) und *Hornklauseln*.

Prolog ist wie Lisp traditionell eine interpretierende Sprache. Dem Interpreter wird eine Datenbasis (das Prologprogramm) übergeben, über die Fragen an den Interpreter gestellt werden können.

[30] Wenn Argument `n` entweder `0` oder `1`

[31] dann gib `1` zurück

[32] sonst addiere die beiden Selbstaufrufe.

[33] Berechne von jedem Element der Liste das Quadrat: die Funktion `map` ruft ihr erstes Argument, eine Funktion, mit jedem Element des zweiten Argumentes, einer Liste, auf. Das Zeichen `'` ist eine Abkürzung für die Funktion `quote`, die die Evaluation der Liste - mit der nicht definierten Funktion `1` - unterdrückt.

[34] Die Funktion `lambda` gibt eine anonyme Funktion zurück.

Zum Verständis Prologs hilft es, alle anderen Sprachkonzepte zunächst zu vergessen: es gibt in Prolog weder Prozeduren noch Funktionen, auch keine Kontrollstrukturen, die denen anderen Sprachen vergleichbar wären. Dagegen verwendet der Programmierer in Prolog *Symbole*, *Variablen* und *Prädikate.* Diese werden an ihrer Stellung im Ausdruck unterschieden.

In Prolog wird *deklarativ* programmiert: innehalb des Prologprogrammes werden *Fakten* und *Regeln* (rules) formuliert. In einem Fakt wird mindestens ein Symbol mit einem Prädikat versehen:

```
schoen(wetter).
```
[35]

Solch ein Fakt in der Regelbasis ist immer gültig. Soll dieser Fakt aber nur unter bestimmten Umständen gelten, kann eine Regel formuliert werden:

```
schoen(Wetter) :-
    trocken(Wetter),
    sonnig(Wetter).
```

Damit wäre die Gültigkeit des Prädikates `schoen` für die Variable `Wetter`[36] abhängig von der Gültigkeit der Prädikate `trocken` und[37] `sonnig` für die selbe Variable. Wir können den obigen Ausdruck also verstehen als: wenn das Wetter sowohl trocken als auch sonnig ist, dann ist es schön. Wir können diese Beziehung auch anders herum verstehen: um schönes Wetter zu haben, sorge erst dafür, daß es trocken und daß es sonnig ist.

Diese Sichtweise ermöglicht in Prolog eine sehr direkte Umsetzung von *Zielorientierter Programmierung* (goal driven programming). Dabei wird die Gültigkeit eines Ausdruckes[38] abhängig gemacht von den zuvor zu erzielenden Gültigkeiten anderer Ausdrücke - die Abfrage dieser Gültigkeiten aber macht der Interpreter/die Interferenzmachine selbständig, indem sie ihre Datenbasis nach einen passenden Ausdruck durchsucht.

Welche Fakten oder Regeln jeweils erfüllt werden müssten, erkennt die Interferenzmachine durch das *Matching*[39] (to match etwa: passend, angemessen kombinieren). Ein Fakt, ein Regelkopf usw. „matcht“ einen anderen, wenn die Struktur beider Ausdrücke übereinstimmt. Die Struktur wird durch die Prädikate und die Verschachtelung bestimmt, Variablen können auch komplexe Strukturen matchen. Folgende Strukturen matchen alle untereinander:

```
familie( vater(adam),
         mutter(eva),
         kinder([kain, abel])).
familie( vater(X),
         mutter(eva),
         kinder([Y, Z])).
familie( X, Y, Z).
```
[40]

[35] Das Prädikat steht vor, die damit genauer bezeichneten oder voneinander abhängigen Symbole innehalb der Klammer. Der Punkt schließt die Aussage ab.

[36] Variablen sind am Großbuchstaben oder einem Unterstrich (_) am Anfang erkennbar .

[37] Die Relation *und* ist ausgedrückt durch das Komma, ein Semikolon drückt *oder* aus.

[38] Ein gültiger Ausdruck kann in Prolog auch die Ausführung von Aktionen bewirken, denn Funktionalitäten, die in anderen Sprachen als Prozedur definiert sind (z.B. `print`), sind in Prolog Prädikate.

[39] In der Logik wird dasselbe Unifikation genannt.

[40] Die eckigen Klammern schließen eine Liste ein.

Hat die Interferenzmachine durch das Matching einen Ausdruch gefunden, der zwar der Struktur nach paßt, aber nicht gültig ist, führt sie automatisch ein *Backtracking*[41] durch. Sie geht im Suchvorgang bis vor den „Aufruf“ des matchenden Ausdrucks zurück, und versucht einen anderen Ausdruck zu finden, der ebenfalls matcht. Findet sie keinen Ausdruck, ist das in der Anfrage gestellte Prädikat nicht gültig und Prolog gibt `no` zurück.

In Prolog gilt also die *Closed World Assumption,* die Annahme einer geschlossenen Welt: nur explizit gültige Aussagen sind gültig.

Es sind aber prinzipiell auch mehrere Lösungen einer Suchanfrage möglich, da verschiedene Ausdrücke matchen können. Diese „Unbestimmtheit“ wird *nicht determiniert* genannt, es gibt nicht genau eine determinierte Lösung.

Auch der Kopf einer Regel kann Ausdrücke im Körper der Regel matchen. Dadurch sind *rekursive* Regeldefinitionen möglich, die insbesondere bei der Verwendung von Listensehr ausdrucksstarkes deklaratives Definieren von Regeln ermöglichen:

```
mitglied(X, [X | _]).[42][43]
mitglied(X, [_ | Schwanz]) :-
  mitglied(X, Schwanz).
```

Solch ein mehrstelliges Prädikat ist auch immer seine eigene „Umkehrfunktion“, in einer Anfrage kann sowohl die erste, als auch die zweite (und auch beide oder keine) Stelle des Prädikates eine Variable sein.

Innehalb einer Prologimplementierung sind verschiedene Prädikate, einschließlich solcher zur Ein- und Ausgabe, vordefiniert.

Bestimmte Prädikate erlauben das Hinzufügen oder Löschen von Regeln aus der aktuellen Regelbasis. Dadurch kann sich ein Prologprogramm abhängig von seinen Eingaben dynamisch während der Abfragen verändern.

5 Skriptsprachen

typische Sprachvertreter

Perl, Tcl, (AppleScript, VisualBasic for Applications)

Musik-Programmiersysteme

Cecilia verwendet Tcl/TK

Patch Work kann durch AppleScript automatisiert werden

[41] Dieser Mechanismus ist neben dem Matching die wichtigste Kontrollstruktur in Prolog. Diese beiden Kontrollstrukturen ermöglichen die große Abstraktionsfähigkeit bzw. das hohe Sprachniveau von Prolog. Weil Backtracking aber auch zur Lösung von Problemen eingesetzt wird, die prozedural leicht realisierbar sind, ist die Programmierung in Prolog häufig wenig effizient. Deshalb hat sich diese Sprache wenig durchgesetzt und gilt ungeachtet ihrer hohen Ausdruckstärke immer noch als experimentell.
Das Backtracking kann durch bestimmte Ausdrücke auch unterdrückt werden.

[42] Das erste Element einer Liste (der *Kopf*) kann deklarativ durch einen Strich (|) vom Rest (*Schwanz*) getrennt werden, der Schwanz bezeichnet - wie in Lisp - eine Unterliste:
[kopf | schwanz].

[43] Der Unterstrich allein bezeichnet (irgendeine) *anonyme Variable*, jeder Unterstrich kann dabei eine andere Variable bezeichen.

Anwendungen

Skriptsprachen sind als Kitt zwischen verschiedenen Programmen einschließlich dem Betriebssytem gedacht. Diese Sprachen sollen insbesondere praktikabel (leicht zu verwenden, effizient und komplett) sein, Schönheit (Kürze, Eleganz aber auch theoretische Fundierung) ist weniger wichtig.[44] Sie sind in der Regel prozedural, allerdings wird auch Scheme als Skriptsprache verwendet und Perl ist (seit Version 5) objektorientiert.

Besonders ausdrucksstark sind die (in verschiedenen Skriptsprachen leider je mit etwas anderer Syntax enthaltenen) regulären Ausdrücke um Texte nach ganz bestimmten Mustern zu durchsuchen und zu bearbeiten.

Ein zusätzliches Aufgabenfeld für verschiedene Skriptsprachen entstand in letzter Zeit für Internetdokumente, Perl ist inzwischen ein Quasistandart für CGI-Skripte[45]. Tcl ist insbesondere in Zusammenhang mit TK - einem freien Tool zur schnellen Entwicklung von GUI[46] für verschiedene Plattformen (X, Apple, Windows) - wichtig, wird aber auch zunehmend im Internet eingesetzt.

6 Literaturempfehlungen

Eine sehr gut lesbare Einführung in Grundlagen des Programmierens bietet [HW94]. Dieser Text ist für nichttechnische Studiengänge gedacht als Vorbereitung auf [AG85], was die eigentliche Einführung in die Computer Science (Informatik) für alle technischen Studiengänge am MIT[47] darstellt. Beide Bücher verwenden Scheme[48]. Der zweite Text ist auch in Deutsch erhältlich, der erste verwendet ein wirklich freundliches Englisch. Leider muß man sagen, daß amerikanische Lehrbücher in aller Regel deutlich besser lesbar sind, als in Deutschland geschriebene.

Ein gutes Lehrbuch für Java und damit eine junge objektorientierte Sprache ist [**?**]. Eine regelmäßig aktualisierte Version des Buches ist ladbar von *http://java.sun.com/Series/Tutorial.* Eine gute Einführung in Common Lisp bietet [WH93], ab der dritten Auflage wird CLOS, die objektorientierte Erweiterung von Lisp eingeführt. Mit [Bra86] läßt sich gut Prolog lernen. Beide Bücher sind auch in deutscher Übersetzung erhältlich.

Literatur

[AG85] Harold Abelson and Gerald Jay Sussman mit Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

[44] vergl. Perl Manual

[45] Common Gateway Interface, ermöglicht Kommunikation zwischen Browser und Web-Server, notwendig für Interaktivität wie z.B. bei Formularen

[46] Graphic User Interface - grafische Benutzerschnittstelle, -oberfläche

[47] Massachusetts Institute of Technology, eine amerikanische Eliteuniversität für technischen Studiengänge

[48] Die Autoren von [HW94, Seite xvii ff] rechtfertigen dies, indem sie schreiben:
„There are two schools of thought about teaching computer science. ...

- **The conservative view:** Computer programs have become too large and complex to encompass in a humans mind. Therefor, the job of computer science education is to teach people how to discipline their work in such a way that 500 mediocre programmers can join together and produce a program that correctly meets its specification.
- **The radical view:** Computer programs have become too large and complex to encompass in a humans mind. Therefor, the job of computer science education is to teach people how to expand their minds so that the programs *can* fit, by learning to think in a vocabulary of larger, more powerful, more flexible ideas than the obvious ones. Each unit of programming thought must have a big payoff in the capabilities of the program.

... Symbolic programming is one aspect of the reason why we like to teach computer science using Scheme instead of a more traditional language. More generally, Lisp (and therefor Scheme) was designed to support what we've called the radical view of computer science. In this view, computer science is about tools for expressing ideas.“

[Bra86] Ivan Bratko. *PROLOG Programming for Artificial Intelligenz*. Addison-Wesley, Massachusetts, 1986.

[HW94] Brian Harvey and Mathew Wright. *Simply Scheme, Introducing Computer Science*. MIT Press, 1994.

[WH93] Patrik Henry Winston and Bertold Klaus Paul Horn. *LISP*. Addison-Wesley, 3rd edition, 1989, 1993.