1 Programmierstile

Jeder Mensch hat seine eigene Art, zu sprechen oder zu schreiben. Der eine verwendet gern kurze Sätze, der andere verschachtelt Sätze lieber und benutzt häufig Nebensätze. Manch einer substantiviert gern Verben oder schreibt bzw. spricht vorzugsweise im Passiv. Welchen Sprachstil man verwendet, hängt aber nicht nur vom persönlichen Geschmack, sondern auch wesentlich von der Sprache ab, in der man etwas formuliert, und von dem Zweck (z.B. Juristendeutsch), für den ein Sprachwerk gedacht ist. Die Struktur der Sprache begünstigt bestimmte Stilrichtungen. Man sagt z.B., daß man im Deutschen eher dazu neigt, Sachverhalte zu verklausulieren, als im Englischen.

Auch bei der *Programmierung* unterscheidet man Stile, die aber wegen der wenigen Sprachmittel viel stärker von der Programmiersprache beeinflußt sind, als dies bei natürlichen Sprachen der Fall ist. Die meisten Programmiersprachen sind nur für einen einzigen Programmierstil konzipiert. Jeder Programmierer muß daher seine Denkweise an diesen Stil anpassen, wenn er in einer solchen Programmiersprache programmieren will.

1.1 Klassifikation von Programmiersprachen

Unter den zahlreichen Ansätzen zur Klassifikation von Programmiersprachen, die zu einer Fülle von Kategorien geführt haben, greifen wir drei Klassen heraus, die sich als relativ stabil erwiesen haben und deren charakteristische Merkmale wohlunterscheidbar sind:

- die imperative Programmierung
- die funktionale Programmierung
- die prädikative Programmierung.

Zu jedem dieser Stile gibt es Programmiersprachen, die überwiegend nur den jeweiligen Stil unterstützen. Man nennt sie entsprechend *imperative*, *funktionale* bzw. *prädikative Programmiersprachen*, jedoch ist keine Programmiersprache rein imperativ/funktional/prädikativ; so enthalten imperative Sprachen meist auch funktionale Elemente, prädikative auch imperative usw. Unterschiede zwischen den Stilen bestehen in der „Eleganz“, in der „Klarheit“ und „Kürze“ und in der Mitteilbarkeit. Anhänger der einen Stilrichtung lassen sich aufgrund ihrer Gewöhnung und Erfahrung meist nicht zu einer anderen Richtung „bekehren“. Man sollte jedoch diese Stile kennen und sie je nach dem Aufgabenfeld einsetzen können.

Die drei Stile ordnet man zwei übergeordneten Kategorien zu, der Klasse der *prozeduralen* und der Klasse der *deklarativen* Sprachen (Abb. 1).

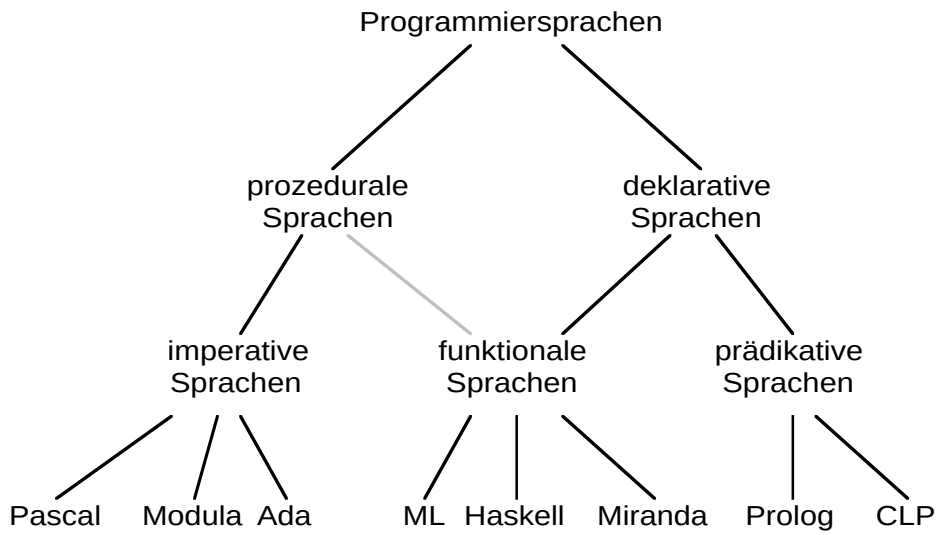

Abb. 1: Klassifikation von Programmiersprachen (die aufgeführten Programmiersprachen sind Beispiele)

Als vierter Stil wird häufig neben den drei obigen die *objektorientierte Programmierung* genannt. Objektorientierte Sprachen verfügen über leistungsfähige Typsysteme, die Datenabstraktion, -kapselung und *Vererbung* unterstützen. Unter einem *Objekt* versteht man dabei die Zusammenfassung von Daten und Operationen (*Methoden*) zu einer Einheit. Damit verknüpft sind Angaben über die Vererbungsbeziehungen zwischen Objekten in der Form „ist Spezialisierung von“ bzw. „ist Verallgemeinerung von“. Objekte können untereinander *Nachrichten* (Daten oder Anweisungen, bestimmte Methoden auszuführen) austauschen. Alle Objekte mit gemeinsamen Eigenschaften werden zu *Klassen* zusammengefaßt.

Zu den objektorientierten Programmiersprachen gehören Sprachen wie Smalltalk-80 (als Reinform), Oberon, Simula (als Urahn), C++ oder Eiffel. Die objektorientierten Sprachen sind jedoch, zumindest soweit es die zur Zeit gängigen betrifft, im Kern imperativ. Andererseits können die objektorientierten Konzepte mit allen drei Stilen kombiniert werden, ohne deren Charakter zu stören. Entsprechende Ansätze werden zur Zeit erforscht und erprobt. Logisch gesehen, läßt sich die objektorientierte Programmierung daher eher zusammen mit der strukturierten Programmierung, als deren Erweiterung sie aufgefaßt werden kann, in die Klasse der Softwareentwurfsmethoden, denn in die Klasse der Programmierstile einordnen.

Objektorientierte Sprachen unterstützen eine *evolutionäre* Art der Software-Entwicklung (*rapid prototyping* = schnelle Erstellung eines ersten lauffähigen Prototyps des geplanten Systems). Probleme werden hierbei nicht wie beim traditionellen Softwareentwurf dadurch gelöst, daß man die Lösung von Grund auf neu entwickelt, sondern man versucht, aufbauend auf einem Grobkonzept, ältere für andere Zwecke erstellte Softwarebausteine wiederzuverwenden und an die neue Aufgabenstellung anzupassen (bottom-up Vorge-

hen). Aus dem Programmieren im herkömmlichen Sinne, sprich dem Erschaffen neuer Software, wird bei der objektorientierten Programmierung also mehr und mehr ein *(Re-)Konfigurieren*, d.h. ein Wiederverwenden und Neuzusammenstellen bereits existierender Komponenten.

1.2 Prozedurale und deklarative Programmierung

Programme in prozeduralen Sprachen erfüllen einen bestimmten *Zweck:* Sie lösen ein Problem, d.h. sie beschreiben einen Lösungsweg, indem sie exakt festlegen, wie die Lösung zu einem Problem maschinell zu berechnen ist. Es ist unmittelbar ersichtlich, welche Größen gegeben und welche gesucht sind und auf welchem Weg man die gesuchten aus den gegebenen Größen gewinnen kann.
Programme in deklarativen Sprachen dienen nicht unmittelbar einem bestimmten Zweck. Sie beschreiben vielmehr exakt die allgemeinen Eigenschaften gewisser Objekte sowie ihre Beziehungen untereinander. Man spricht hier statt von Programmen auch von *Wissen*. Dieses Wissen induziert zunächst kein Problem, folglich läßt sich auch kein Lösungsweg daraus ableiten. Ferner ist nicht festgelegt, welches die bekannten und welches die gesuchten Größen sind. Einen Zweck bekommt das Programm erst, wenn man es um eine Problembeschreibung mit Angabe der bekannten und der gesuchten Größen ergänzt. Die Aufgabe des Computers besteht dann darin, das Wissen zur Lösung des Problems zu nutzen.

Bisher gibt es keine deklarativen Programmiersprachen im engeren Sinne. Auch prädikative Sprachen (s. unten) sind mit Ausnahme von kleinen „Spielbeispielen“ nicht deklarativ. Wir wollen daher den Unterschied zwischen deklarativem und prozeduralem Stil anhand der natürlichen Sprache erläutern.

Beispiel: Das folgende „Programm“ ist deklarativ:

> *Die Kaufkraft eines Geldbetrages sinkt nach Ablauf eines Jahres um die Preissteigerungsrate*.

Die Aussage formuliert das „zweckfreie“ (und zugegebenermaßen naive) Wissen über den Zusammenhang zwischen Geldbetrag, Kaufkraft und Preissteigerungsrate. Sie definiert kein Problem, daher sind gesuchte und gegebene Größen nicht festgelegt. Zu welchem Zweck dieses Wissen verwendet werden soll, muß der Programmierer bestimmen. Zum Beispiel kann er dieses Wissen zur Lösung folgender Probleme nutzen:

- um zu einem gegebenen Geldbetrag G zu Beginn eines Jahres und zu gegebener Preissteigerungsrate P die Kaufkraft K am Schluß eines Jahres zu berechnen, oder
- um zu G und K nach Ablauf eines Jahres P zu ermitteln, oder
- um zu K eine Tabelle von Paaren (G,P) zu liefern, so daß G vermindert um P der Kaufkraft K entspricht, oder
- um alle Tripel (G,P,K) mit der spezifizierten Eigenschaft auszugeben.

Weitere Beispiele für deklarative Programme:

1) Katzen trinken Milch.
2) Wenn A und B dieselbe Mutter haben, dann sind A und B Geschwister.
3) Das Quadrat einer geraden/ungeraden Zahl ist gerade/ungerade.

Eine mögliche prozedurale Darstellung des obigen deklarativen Programms ist:

> *Die Kaufkraft eines Geldbetrages nach Ablauf eines Jahres erhält man, indem man den Geldbetrag um die Preissteigerungsrate vermindert.*

Diese Darstellung erfüllt gegenüber der deklarativen einen wohldefinierten Zweck: Sie präzisiert das Problem, die Kaufkraft eines Geldbetrages nach Ablauf eines Jahres zu ermitteln, und den zugehörigen maschinell nachvollziehbaren Lösungsweg. Es wird ferner festgelegt, welche Größen als gegeben vorausgesetzt werden (Geldbetrag und Preissteigerungsrate) und welche Größe daraus ermittelt werden kann (Kaufkraft nach Ablauf eines Jahres).

Weitere Beispiele für prozedurale „Programme“:

1) Zum Öffnen Lasche anheben, zusammendrücken und farbige Ecke abreißen.
2) Falls Sie weitere Informationen wünschen, brauchen Sie nur den ausgefüllten Coupon zurückzusenden.
3) Zur Installation der Software müssen Sie mindestens die Dateien X und Y auf Ihre Festplatte kopieren. Alles weitere entnehmen Sie der Datei „Liesmich“.

1.3 Beschreibungsmächtigkeit der Programmierstile

Die natürlichen Sprachen, zumindest unseres Kulturkreises, sind gleichmächtig: Beliebige Sachverhalte lassen sich gleichermaßen in Deutsch, Englisch, Französisch, Italienisch usw. formulieren. Gelten ähnliche Aussagen auch im Bereich der Programmierstile? Zunächst muß man sich überlegen, welche „Sachverhalte“ man mit Programmiersprachen ausdrücken will. Offenbar ist dies die Menge aller berechenbaren, also durch Maschinen nachvollziehbaren Funktionen. Tatsächlich sind in diesem Sinne alle genannten Programmierstile gleich leistungsfähig und besitzen die gleiche Ausdruckskraft: Jede berechenbare Funktion läßt sich durch ein Programm jedes Programmierstils beschreiben. Dies bedeutet: Wenn man zu irgendeinem Problem ein Programm im Programmierstil A erstellt hat, dann kann man auch ein Programm in irgendeinem der anderen Stile schreiben, das die gleiche Aufgabe löst.

Diese Aussage beweist man z.B. durch folgenden Ringschluß: Es gibt ein imperatives Programm, das funktionale Programme simuliert, es gibt weiter ein funktionales Programm, das prädikative Programme nachvollzieht, und es gibt schließlich ein prädikatives Programm, mit dem man imperative Programme simulieren kann. Der exakte Beweis wird in anderen Veranstaltungen geführt.

1.4 Die Programmierstile im Detail

Wir wollen im folgenden keine neuen Programmiersprachen präsentieren, sondern nur den Charakter der einzelnen Stile erarbeiten. Die wichtigsten Unterscheidungsmerkmale der Stile sind hierbei

- der *Kalkül* oder das *mathematische Modell:* Jedem Programmierstil liegt ein bestimmtes mathematisches Modell oder ein Kalkül zugrunde. Alle Programmiersprachen eines Stils basieren auf diesem Modell und sind, vereinfacht gesprochen, durch Ergänzung des Modells um syntaktische Konstrukte (im Engl. *syntactic sugar* genannt) entstanden, welche zwar die Lesbarkeit von Programmen und die Benutzungsfreundlichkeit der Sprache verbessern, nicht aber ihre Ausdruckskraft erhöhen;
- der Begriff des *Programms*;
- der Begriff des *Befehls*, besser des elementaren Bausteins zur Beschreibung von Programmen und der Konstruktoren, um die elementaren Bausteine zu komplexeren zusammenzusetzen;
- das *Variablenkonzept:* Jedem Programmierstil liegt eine ganz bestimmte Vorstellung von dem Begriff der Variablen und den möglichen Operationen mit Variablen zugrunde.

Die Form, die ein Programm in den unterschiedlichen Stilen jeweils besitzt, werden wir anhand des bekannten Problems *Mischen/Verschmelzen von Zahlenfolgen* veranschaulichen:

> *Gegeben sind zwei aufsteigend sortierte Folgen von ganzen Zahlen. Gesucht ist ein Algorithmus, der die beiden Folgen zu einer aufsteigend sortierten Folge mischt*.

Zu jedem Stil geben wir jeweils ein Programm in einer fiktiven und in einer konkreten Programmiersprache an. Die Darstellung in der fiktiven Sprache läßt gegenüber der konkreten die Merkmale der Stile deutlicher hervortreten.

1.4.1 Imperative Programmierung

Zugrundeliegendes Modell: Die imperative Programmierung basiert auf dem mathematischen Modell der Registermaschine. Eine *Registermaschine* (Abb. 2) ist ein Automat, dessen Speicher aus einer festen Anzahl von Registern besteht. Jedes Register kann eine beliebig große natürliche Zahl aufnehmen. Auf ein Register können drei verschiedene Operationen angewendet werden: Erhöhen des Inhalts um 1, Vermindern des Inhalts um 1 (0 darf jedoch nicht unterschritten werden) und Abfrage des Inhalts auf 0. Programme der Maschine sind Folgen von sehr einfachen Assembler-ähnlichen Anweisungen der Form

i: do f; goto j

oder

i: if t then goto j else goto k.

Hierbei sind i und j Marken, f eine Operation der Form „Erhöhe Register x um 1“ oder „Erniedrige Register x um 1“ und t ein Test der Form „Register x=0?“. Eine Register-

maschine mit m Registern arbeitet folgendermaßen: Zunächst werden die r≤m Eingabewerte in den ersten r Registern abgelegt werden. Alle übrigen Register werden mit 0 initialisiert. Dann führt die Maschine das Programm beginnend bei der mit 0 markierten Anweisung aus. Die Maschine stoppt, sobald sie zu einer Marke verzweigen soll, die in dem Programm nicht existiert. Der Inhalt der ersten s Register bildet dann die Ausgabe der Maschine. Die Maschine berechnet also eine Funktion $f:IN_0^r \rightarrow IN_0^s$ berechnet. Dieses Maschinenmodell ist trotz seiner Einfachheit außerordentlich leistungsfähig. Man kann zeigen, daß es zu jeder berechenbaren Funktion f eine Registermaschine mit höchstens drei (!) Registern gibt, die f berechnet. Wir werden später genauer auf die Details eingehen.

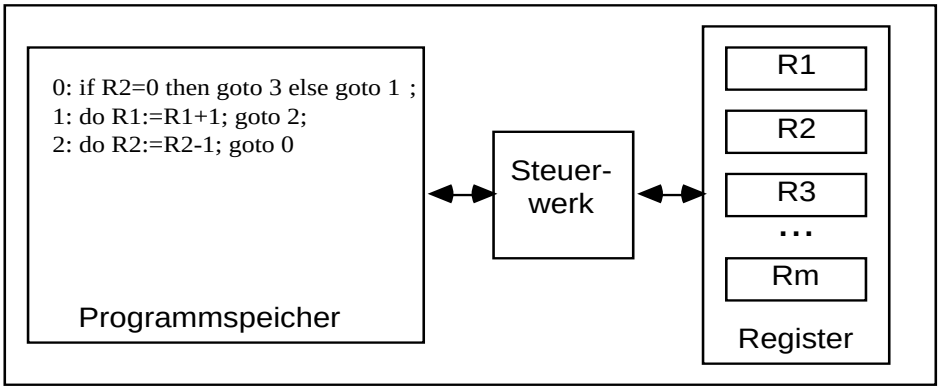

Abb. 2: Registermaschine

Programm: Ein imperatives Programm ist eine nach bestimmten Konstruktionsprinzipien aufgebaute Zusammenstellung von Befehlen (lat. imperare=befehlen) oder Anweisungen.

Start eines Programms: Der Ablauf eines Programms wird durch Ausführung des ersten Befehls gestartet.

Anweisung: Eine Anweisung ist die Aufforderung an den Computer, eine Handlung auszuführen. Man unterscheidet *elementare* Anweisungen und *strukturierte* Anweisungen, die mithilfe von gewissen Bildungsprinzipien (sog. *Konstruktoren*) aus den elementaren Anweisungen gewonnen werden können. Die wichtigste elementare Anweisung ist die *Zuweisung*. Zu den Konstruktoren gehören mindestens die *Konkatenation* (Sequenz, Hintereinanderausführung von Anweisungen), die *Alternative* (bedingte Anweisung: if, case) und die *Iteration* (Schleife: while, repeat).

Variablenkonzept: Eine Variable ist ein Behälter mit Bezeichner, in den Werte meist eines vorher festgelegten Typs abgelegt werden können. Die zentralen Operationen auf Variablen sind *Lesen* und *Schreiben*.

Das Variablenkonzept der imperativen Programmierung orientiert sich an der Funkti-

onsweise heutiger Rechner (*Von Neumann-Architektur*) mit Speichern, die aus einzelnen Speicherzellen bestehen. Eine Speicherzelle entspricht dabei einem Standardbehälter für Variablen.

Historische Entwicklung: Die imperativen Sprachen haben sich um 1950 aus den Maschinen- und Assemblersprachen der ersten industriellen Rechner entwickelt. Den Anfang bildeten die unstrukturierten Programmiersprachen FORTRAN (1954), COBOL (1959) und BASIC (1962). Es folgten auf der Basis von ALGOL60 (1958) die strukturierten Sprachen PASCAL (1970), C (1974) und MODULA-2 (1977). Mit ADA (1979) ist die Entwicklung imperativer Sprachen zu einem gewissen Abschluß gekommen. Die später vorgestellten Sprachen enthalten nahezu ausschließlich objektorientierte Erweiterungen.

Beispiele:

1) Mischen in der fiktiven imperativen Programmiersprache PRO:

```
def s1, s2 s3: Zahlenfolge;
solange s1 ≠ [ ] und s2 ≠ [ ] tue
    wenn erstes(s1) < erstes(s2) dann
        s3 ← s3 • [ erstes(s1) ];
        s1 ← rest(s1)
    sonst
        s3 ← s3 • [ erstes(s2) ];
        s2 ← rest(s2)
    ende
ende;
s3 ← s3 • s1 • s2.
```

Start des Programms: run misch.

2) Mischen in PASCAL:

```
program misch(f,g,h);
var f,g,h: file of integer;
begin
    reset(f); reset(g); rewrite(h);
    while not (eof(f) or eof(g)) do
    begin
        if f↑ < g↑ then
        begin
            h↑:=f↑; put(h); get(f)
        end else
        begin
            h↑:=g↑; put(h); get(g)
        end
    end;
    while not eof(f) do
    begin
        h↑:=f↑; put(h); get(f)
    end;
    while not eof(g) do
    begin
        h↑:=g↑; put(h); get(g)
    end
end.
```

1.4.2 Funktionale Programmierung

Zugrundeliegender Kalkül: Die funktionale Programmierung basiert auf dem *λ-Kalkül*, der Anfang der 30er Jahre von A. Church entwickelt wurde, um den Begriff der Berechenbarkeit zu präzisieren. Hierbei handelt es sich um einen mathematischen Formalismus zur Beschreibung von Definition und Anwendung von Funktionen, die durch Rechenvorschriften gegeben sind. Dabei verwendet er wie schon beim Registermaschinenmodell sehr elementare, aber überaus mächtige Operationen.

Formeln im λ-Kalkül (*λ-Ausdrücke*, *λ-Terme*) definiert man induktiv. Hierzu sei $X=\{x_1,x_2,x_3,...\}$ eine Menge von Variablen. Dann gilt:

(1) Jede Variable $x \in X$ ist ein λ-Term.

(2) Wenn M und N λ-Terme sind, dann ist auch (MN) ein λ-Term. (MN) bezeichnet die *Anwendung* des λ-Terms M auf den λ-Term N als Argument. In der Mathematik verwendet man hierfür die gebräuchlichere Notation M(N).

(3) Wenn $x \in X$ eine Variable und M ein λ-Term ist, dann ist auch (λx.M) ein λ-Term. Diese Vorschrift (sog. *Abstraktion*) beschreibt den Übergang von einem λ-Term M zu einer Funktion. Hierbei ist x der formale Parameter und M der Funktionsrumpf. Das λ entspricht dem Schlüsselwort function in Programmiersprachen. Der Punkt trennt Funktionskopf und Funktionsrumpf voneinander. Aus Sicht einer Programmiersprache beschreibt die Abstraktion also die Zusammenfassung von mehreren Konstrukten zu einer Funktion und ihre Parametrisierung. Allerdings kann man λ-Termen im Unterschied zu Funktionen in Programmiersprachen keine Namen geben, über die man sie später ansprechen kann. Daher muß man jeden λ-Term überall dort vollständig hinschreiben, wo man ihn verwenden will.

Ein funktionales Programm ist in diesem Sinne also nichts anderes, als eine Folge von benannten λ-Termen

$$f_1=T_1, ..., f_n=T_n.$$

Hierbei sind $f_1, ..., f_n$ die Funktionsbezeichner.

Bezeichnung: Ein Auftreten von x in M im λ-Term (λx.M) heißt *gebundenes* Auftreten von x. Kommt x im Term N außerhalb einer λ-Abstraktion (lx.M) vor, so ist dies jeweils ein *freies* Auftreten von x.

Die Anwendung von durch λ-Terme beschriebenen Funktionen auf ein Argument wird durch die sog. *β-Regel* definiert, welche die Ersetzung von formalen Parametern durch aktuelle beschreibt. Zugrundegelegt ist beim λ-Kalkül die Parameterübergabeart call-by-name (textuelle Ersetzung). Die β-Regel lautet:

$$((\lambda x.M)N)=M[x \leftarrow N].$$

Bedeutung: Die Funktion beschrieben durch den λ-Term (λx.M) wird angewendet auf den Argumentterm N, und der Gesamtterm kann ersetzt werden durch $M[x \leftarrow N]$, also durch den Term, den man erhält, wenn man in M die Variable x überall textuell durch den Term N ersetzt (*Substitution*), außer in Teiltermen von M, in denen x durch ein λ

gebunden ist, also anschaulich nicht in Unterfunktionen der Funktion (λx.M), die ebenfalls den formalen Parameter x besitzen.
Diese β-Regel ist also *zulässig*, wenn durch M[x←N] keine in N freie Variable gebunden wird. Anderenfalls muß man Umbenennungen vornehmen, sog. α-Konversionen. Jeder Term (λx.M) darf durch (λy.M[x←y]) ersetzt werden, falls M[x¨y] zulässig ist.
Bei*spiel:* Wir betrachten den λ-Term

((λx.((xd)((λy.(xy))a)))b).

Hierin sind zwei Funktionen verborgen: Die eine ist (λy.(xy)) mit formalem Parameter y und Rumpf (xy); diese Funktion ist geschachtelt im λ-Term (λx.((xd)((λy.(xy)))a)) mit formalem Parameter x und Rumpf ((xd)((λy.(xy)))a). Der zuletzt genannte λ-Term wird angewendet auf den aktuellen Parameter b. Die β-Regel bestimmt nun, wie der Gesamtausdruck ausgewertet wird: Im ersten Schritt wird die Funktion (λy.(xy)) mit dem aktuellen Parameter a versorgt, und man erhält durch textuelle Substitution des formalen durch den aktuellen Parameter den Ausdruck (xa). Im zweiten Schritt wird die so geänderte Funktion (λx.((xd)(xa))) auf den aktuellen Parameter b angewendet, was zu ((bd)(ba)) führt. Dies ist das Ergebnis des Gesamtausdrucks. Führt man die beiden Substitutionschritte in umgekehrter Reihenfolge durch, kommt man zum gleichen Resultat.
Dieser fortlaufende Ersetzungsprozeß symbolisiert die Berechnung eines Funktionsergebnisses. Daß dieses Funtkionsergebnis nahezu eindeutig sein sollte und tatsächlich auch ist, drückt sich in der sog. *Konfluenz* und *Church-Rosser-Eigenschaft* aus: Bricht die Auswertung von λ-Termen – wie immer man auch vorgeht – ab, so ist das Ergebnis bis auf Umbenennung eindeutig bestimmt. Auf den Ergebnisterm können dann nur noch α-Konversionen vorgenommen werden. Wie in obigem Beispiel ist es also für das Ergebnis gleichgültig, in welcher Reihenfolge man die Auswertung eines λ-Terms durchführt.
Man kann zeigen, daß man mit dem λ-Kalkül alle berechenbaren Funktionen beschreiben kann. Der λ-Kalkül ist also trotz seiner einfachen Struktur genauso mächtig wie Registermaschinen oder Programmiersprachen.
Um z.B. zu zeigen, daß Registermaschinen und l-Kalkül gleichmächtig sind, muß man alle Elemente von Registermaschinen durch λ-Terme beschreiben und umgekehrt. Das folgende Beispiel zeigt ansatzweise, wie man diese Aufgabe, die zuerst von A. Church (1941) bewältigt wurde, lösen kann.
Beispiel: Registermaschinen operieren mit natürlichen Zahlen; diese gibt es im λ-Kalkül nicht. Folglich muß man mit dem λ-Kalkül natürliche Zahlen und die Registeroperationen wie +1, -1 etc. simulieren. Wir definieren (zur besseren Lesbarkeit lassen wir Klammern weitgehend weg):

0 sei repräsentiert durch den λ-Term λf.λx.x,
1 sei repräsentiert durch den λ-Term λf.λx.fx,
2 sei repräsentiert durch den λ-Term λf.λx.f(fx),

3 sei repräsentiert durch den λ-Term λf.λx.f(f(fx)) usw.

Dann kann man auf diesen Darstellungen arithmetische Operationen durch λ-Terme definieren, z.B. die Nachfolgerfunktion N mit N(x)=x+1:

N ist repräsentiert durch den λ-Term λn.λf.λx.f(nfx).

Prüfen wir dies nach und berechnen den Nachfolger der Zahl 2:

N(2) ist repräsentiert durch den λ-Term (λn.λf.λx.f(nfx))(λf.λx.f(fx)).

Diesen Term können wir mit der β-Regel auswerten:

(λ**n**.λf.λx.f(nfx)) **(λf.λx.f(fx))** → λf.λx.f((λ**f**.λx.f(fx))**f**x)

N 2

durch Einsetzen von (λf.λx.f(fx)) für den formalen Parameter n in (λn.λf.λx.f(nfx)),

... → λf.λx.f((λ**x**.f(fx))**x**)

durch Einsetzen des letzten f für den formalen Parameter f in λf.λx.f(fx),

... → λf.λx.f(f(fx))

durch Einsetzen des letzten x für den formalen Parameter x in λx.f(fx). Damit ist der ursprüngliche Ausdruck vollständig ausgewertet, sein Ergebnis lautet λf.λx.f(f(fx)). Dies ist gerade die Darstellung der Zahl 3 in der oben gewählten Codierung. Folglich realisiert der λ-Term für N die Nachfolgerfunktion.

Programm: Ein funktionales Programm ist eine Menge von Funktionsdefinitionen (Funktionalgleichungen), die nach bestimmten Grundprinzipien aufgebaut sind.

Start eines Programms: Der Ablauf eines Programms wird durch Aufruf einer Funktion mit Parametern gestartet.

Funktion: Funktionen in funktionalen Programmiersprachen entsprechen Funktionen im mathematischen Sinne, also Abbildungen f: A→B von einer Menge A in eine Menge B. Die Funktionen sind jedoch nicht mengentheoretisch, sondern durch *Rechenvorschriften* definiert.

Funktionen werden nach gewissen Bildungsprinzipien (*Konstruktoren*) aus *elementaren* Funktionen aufgebaut. Zu den elementaren Funktionen gehören meist Addition, Subtraktion, Operationen auf linearen Listen, die Alternative if B then F(...) else G(...) usw. Die drei zentralen Konstruktoren spielten schon beim λ-Kalkül eine dominante Rolle:

= *Komposition/Einsetzung:* Funktionen können komponiert werden, d.h. als aktuelle Parameter von Funktionen können wiederum Funktionsanwendungen verwendet werden. Dieser Konstruktor ermöglicht auch die Definition *rekursiver* Funktionen.

= *Anwendung:* Eine Funktion kann auf aktuelle Parameter angewendet werden. Hierbei werden die aktuellen Parameter nach bestimmten Strategien ausgewertet und an die Stelle der formalen Parameter im Funktionsrumpf eingesetzt. Anschließend wird der modifizierte Funktionsrumpf ausgewertet.

= *Abstraktion:* Parametrisierung eines Ausdrucks, d.h. Übergang zu einer Funktionsdefinition und Angabe, welche Elemente des Ausdrucks als Parameter und welche als Konstanten aufzufassen sind.

Variablenkonzept: Eine Variable ist ein Bezeichner, der an ein konkretes Objekt meist eines bestimmten vorher festzulegenden Datentyps gebunden ist. Es gibt keine Zuordnung von Variablen zu Speicherzellen oder dynamische Veränderung von Werten, daher können Zwischenwerte nirgends abgelegt oder über einen längeren Zeitraum gespeichert werden. Funktionen können folglich keine Seiteneffekte besitzen, da die Ergebnisse ihrer internen Berechnungen nur über den Funktionswert nach außen gegeben und nicht an irgendeiner nach außen nicht sichtbaren Stelle festgehalten werden können. Diese Eigenschaft bezeichnet man als *referenzielle Transparenz:* Eine Programmiersprache heißt referenziell transparent, wenn alle in ihr möglichen Ausdrücke folgenden Prinzipien genügen:

- *Extensionalitätsprinzip:* Das einzig wichtige Merkmal eines Ausdrucks ist sein Wert. Es ist völlig unwichtig, wie dieser Wert berechnet wird (dies wäre die *intensionale* Sicht).
- *Leibnizsches Prinzip der Ersetzung von Identitäten:* Der Wert eines Ausdrucks ändert sich nicht, wenn man irgendeinen seiner Teilausdrücke durch irgendeinen anderen gleichen Werts ersetzt.
- *Prinzip der Definitheit:* Der Wert eines Ausdrucks ist innerhalb eines gewissen Kontexts (Gültigkeitsbereich der Variablen) immer gleich, unabhängig von der Stelle, an der der Ausdruck im Kontext auftritt.

Beispiel: Man betrachte den Ausdruck

$(2ax+b)\cdot(2ax+c)$ mit $a=4$, $b=5$, $c=6$, $x=7$.

Rechnet man den Ausdruck im Kopf aus, wird man i.a. den Teilausdruck $2ax$ nur einmal berechnen, den Wert 56 sodann in den Ausdruck einsetzen und anschließend

$(56+b)\cdot(56+c)$

weiter auswerten. Oder falls man bereits weiß, daß

$2ax=d+e$,

so kann man statt des ursprünglichen Ausdrucks auch den Ausdruck

$(d+e+b)\cdot(d+e+c)$

auswerten und wird zum gleichen Ergebnis gelangen. Die Tatsache, daß diese Ersetzungen erlaubt sind und nichts am Wert des Ausdrucks ändern, beruht im wesentlichen darauf, daß die Berechnung von $2ax$ keinen *Seiteneffekt* auf die Werte der übrigen Variablen besitzt.

Imperative Programmiersprachen sind nicht referenziell transparent. Hier kann es sich bei a um eine parameterlose Funktion handeln, die zwar immer den Wert 4 liefert, jedoch bei jedem Aufruf einen Seiteneffekt auf x ausübt, so daß $2ax$ beim ersten Mal zwar gleich 56, beim zweiten Mal jedoch einen anderen Wert besitzt.

Die wichtigste Operation auf Variablen ist die *Substitution*. Wird eine Funktion f mit formalem Parameter x und Rumpf R auf einen aktuellen Parameter E (z.B. einen Ausdruck) angewendet, so muß x innerhalb von R überall durch das Argument E substitutiert werden. Hierfür gibt es die bekannten Ersetzungsstrategien (*Substitutionsregeln*) Call-

by-value, Call-by-name, Call-by-need (lazy evaluation).

Historische Entwicklung: Wichtige Stationen der funktionalen Programmierung:

1930: A. Church: Entwicklung des λ-Kalküls.

1958: J. McCarthy: Erfindung der Programmiersprache LISP auf der Basis des λ-Kalküls. Weiterentwicklungen von LISP in der Folgezeit (z.B. SCHEME).

1965: P. Landin: Beschreibung der Sprache ISWIM (Vorläufer von ML). In der Folgezeit: Entwicklung vieler zentraler Ideen im Zusammenhang mit der Anwendung, Notation und Implementierung funktionaler Sprachen.

1978: J. Backus: Einführung der Sprache FP, die sich an dem Kombinatoren-Kalkül (ein dem λ-Kalkül vergleichbarer Kalkül) orientiert. In FP gibt es *keine* Variablen.

1978: R. Milner schlägt, beeinflußt durch ISWIM, die funktionale Sprache ML vor. ML (mittlerweile standardisiert) besitzt ein leistungsfähiges Typkonzept und Sprachelemente für die Definition von abstrakten Datentypen.

Beispiele:

1) Mischen in der fiktiven funktionalen Programmiersprache FUN:

```
funktion misch f:intlist g:intlist → intlist ≡
wenn f=leer dann g sonst
    wenn g=leer dann f sonst
        wenn (erstes f)<(erstes g) dann (erstes f,misch (rest f) g)
                                   sonst (erstes g,misch f (rest g))
        ende
    ende
ende .
```

Anwendung der Funktion durch Aufruf, z.B.: misch [3,7,19,54] [2,3,5,16,74].

2) Mischen in ML:

```
fun  misch [ ] g  = g: int list  |
     misch f [ ] = f   |
     misch (x::r)(x'::r')=if x<x' then [x]@(misch r (x'::r'))
                                  else [x]@(misch (x::r) r').
```

1.4.3 Prädikative Programmierung

Zugrundeliegender Kalkül: Die prädikative Programmierung basiert auf der *Prädikatenlogik*. Sie ist ein zentrales Hilfsmittel (nicht nur) der Mathematik und dient zur Formalisierung von Sachverhalten und logischen Argumenten und zur Präzisierung von mathematischen Begriffen wie Definition, Satz, Beweis, Folgerung, wahr und falsch, usw. Zur Präzisierung definiert man eine formale Sprache, den *Prädikatenkalkül*, der die korrekten Zeichenreihen (*Formeln*) zur Darstellung dieser Sachverhalte beschreibt. Eine solche Formel ist zunächst eine sinnleere Folge von Symbolen (Buchstaben und Zeichen), sie geht erst durch *Interpretation*, also durch Zuordnung der Symbole zu realen Objekten, in eine Aussage (*Prädikat*) über, deren Wahrheitsgehalt (*wahr* oder *falsch*) man bestimmen kann.

Prädikatives Programmieren ist in diesem Sinne nichts anderes als das Aufstellen von

Behauptungen und das Beweisen der Behauptungen in einem System von *gültigen* (d.h. immer wahren) Aussagen, wobei als Aussagen jedoch nur bestimmte Grundformen (sog. *Hornformeln*) zugelassen sind. (Im folgenden bedeutet ® stets „für alle“).

Beispiel: Eine typische Formel des Prädikatenkalküls ist z.B.:

$$(\exists f)\ (f(a)=b \wedge (©x)\ (p(x) \Rightarrow f(x)=g(x,f(h(x))))).$$

Eine Interpretation der Formel, die der sinnleeren Zeichenfolge eine Bedeutung zuordnet, ist bestimmt durch

a) eine nichtleere Menge M,
b) Zuordnung von a und b zu Elementen von M,
c) Zuordnung von p zu einem einstelligen Prädikat über M, also zu einer Abbildung p: $M \to$ {wahr, falsch},
d) Zuordnung von g zu einer zweistelligen Funktion g: $M \times M \to M$,
e) Zuordnung von h zu einer einstelligen Funktion h: $M \to M$.

Als Interpretation kann man z.B. wählen: M sei die Menge $\mathbb{N}_0$, a die Zahl 0, b die Zahl 1, p(x) die Bedingung $x>0$, $g(x,y)=x \cdot y$ und $h(x)=x-1$ für $x \geq 1$, und 0 für $x<1$. Die Formel geht bei dieser Interpretation in folgendes Prädikat über:

$$(\exists f)\ (f(0)=1 \wedge (©x)\ (x>0 \Rightarrow f(x)=x \cdot f(x-1))).$$

Diese Aussage ist wahr, denn die Fakultätsfunktion $f(x)=x!$ erfüllt dieses Prädikat. Sie ist aber nicht gültig, also nicht in jeder Interpretation wahr.

Programm: Ein prädikatives Programm ist eine Menge von wahren Aussagen über bestimmte Sachverhalte (*Fakten*) und von logischen *Schlußregeln*, mit denen aus gegebenen Fakten neue hergeleitet werden können. Fakten und Regeln lassen sich im deklarativen Sinne als *Wissen* über bestimmte Sachverhalte auffassen.

Start eines Programms: Der Ablauf eines Programms wird durch Eingabe einer Behauptung gestartet. Der Computer versucht dann die Behauptung mithilfe der Regeln und der Fakten zu beweisen.

Faktum: Ein Faktum ist eine wahre Aussage über die Eigenschaften von oder die Beziehungen zwischen Objekten. Besteht etwa zwischen den Objekten $a_1,...,a_n$ die Beziehung e, so schreibt man das Faktum

$$e(a_1,...,a_n).$$

Beispiel: Besteht zwischen den Objekten (hier: Folgen) X,Y,Z die Beziehung „Z ist die gemischte Version der Folgen X und Y“, so schreiben wir:

$$\text{misch}(X,Y,Z).$$

Dann ist offenbar

$$(®Y)\ (\text{misch}([\ ],Y,Y))$$

ein Faktum, denn mischt man die leere Folge mit einer beliebigen Folge Y, so ist Y das Ergebnis.

Schlußregel: Schlußregeln besitzen die allgemeine Form

$$\text{Prämisse} \Rightarrow \text{Konklusion}.$$

Bedeutung: Ist die Prämisse erfüllt, so gilt die Konklusion. Die Prämisse wird gebildet

aus Termen, die durch logisches UND verknüpft sind.

Beispiel: Offenbar ist

$$(\copyright A,B,C,D,E,F,G)\ ((A \leq C \wedge misch([A,B],[C,D],[E,F,G])) \Rightarrow misch([B],[C,D],[A,E,F,G]))$$

eine sinnvolle Regel für die Formulierung des Mischens zweier Folgen der Länge 2 zu einer Ergebnisfolge, die bereits drei Elemente enthält. Hierbei ist angenommen, daß das erste Element der ersten Folge kleiner oder gleich dem ersten Element der zweiten Folge ist und daher an die Ergebnisfolge angefügt werden darf.

Variablenkonzept: Variablen sind im prädikativen Sinne Unbestimmte.

Die wichtigste Operation auf Variablen ist die *Unifikation*. Hiermit bezeichnet man den Prozeß zwei Ausdrücke (*Terme*) „gleichzumachen“, indem man die in den Termen vorkommenden Variablen konsistent durch irgendwelche Terme ersetzt. Dieser Prozeß spielt eine zentrale Rolle beim Beweisen von Aussagen: Während Fakten und Regeln allgemeine Beziehungen zwischen einer Vielzahl von Objekten beschreiben, formuliert eine Behauptung eine Aussage über konkrete Objekte. Man muß daher, um die Fakten und Regeln anwenden zu können, die dort vorkommenden Objekte mit denen aus der Behauptung identifizieren. Dies leistet die Unifikation.

Beispiele:

1) Oben hatten wir das Faktum

$$(\copyright Y)\ (misch([\ ],Y,Y))$$

definiert. Möchte man nun die Behauptung $misch([\ ],[1,3],[1,3])$ beweisen, so kann man das Faktum verwenden, wenn man die Variable Y des Faktums durch die Folge $[1,3]$ ersetzt (unifiziert).

2) Gegeben sei das Faktum

$$(\copyright D,F,H)\ (z(a(b(D)),d(e(F)),g(H))).$$

Kann man dieses Faktum verwenden, um die Behauptung

$$(\exists H,K,F)\ (z(H,K,g(F)))$$

zu beweisen? Ja, denn man kann beide Terme unifizieren, wenn man

H und F durch $a(b(D))$,

K durch $d(e(a(b(D))))$

ersetzt. Dann erfüllen also alle H,K,F die Behauptung, die die beschriebene Gestalt haben, wobei D beliebig gewählt werden kann.

Historische Entwicklung: Die prädikative Programmierung begann ihren Aufstieg in den frühen 70er Jahren und geht auf Arbeiten über automatisches Beweisen und Künstliche Intelligenz zurück. Diese Arbeiten mündeten in eine Publikation von J.A. Robinson (1965) über das *Resolutionsprinzip*, ein besonders zur Automatisierung geeignetes Verfahren für die Ableitung von Aussagen aus Fakten und Regeln. R.A. Kowalski (1972) gilt als Urheber der Idee, Logikkalküle als Programmiersprachen zu verwenden. Seine Ideen wurde zuerst von A. Colmerauer (1973) in der Sprache PROLOG verwirklicht. Die aktuellen Forschungen befassen sich damit, die Sprache PROLOG um ein Daten-

typsystem und um funktionale Elemente zu ergänzen.

Beispiele:

1) Mischen in einer fiktiven prädikativen Programmiersprache:

(©Y) (misch([ ],Y,Y)).

(©Y) (misch(Y,[ ],Y)).

(©A,B,X,Y,Z) (A≤B ∧ misch(X,[B|Y],Z) ⇒ misch([A|X],[B|Y],[A|Z])).

(©A,B,X,Y,Z) (B<A ∧ misch([A|X],Y,Z) ⇒ misch([A|X],[B|Y],[B|Z])).

Erläuterung: Die ersten beiden Fakten beschreiben die Situation, in der eine der beiden Folgen leer ist. In diesem Fall ist das Ergebnis die jeweils andere Folge. Die beiden Regeln formulieren den Vorgang des Mischens, sowohl für den Fall, daß das erste Element A der einen Folge kleiner ist als das erste Element B der anderen Folge und in die Ergebnisfolge übernommen werden muß, als auch für den umgekehrten Fall. Der Strich-Operator „|“ spaltet eine Folge in ihr erstes Element und den Rest auf. Die erste Regel lautet also umgangssprachlich: Für alle Zahlen A,B und alle Folgen X,Y,Z gilt: Falls A≤B gilt und die gemischte Version der Folgen X und [B|Y] gleich Z ist, wobei B das erste Element der Folge [B|Y] und Y der Rest ist, dann ist die gemischte Version der Folgen [A|X] und [B|Y] die Folge [A|Z].

Dieses Wissen ist insofern zweckfrei im deklarativen Sinne, als es nur allgemeine Aussagen über das Mischen von Folgen formuliert. Welche der Folgen Ein- und welche Ausgabefolgen sind, ist nicht festgelegt.

Zum Start (d.h. zur Zweckbestimmung) des Programms benötigt man daher eine Behauptung, die das System zu beweisen versucht. Eine Behauptung ist z.B. die Aussage

misch([3,7,19,54],[2,3,5,16,74],[2,3,3,5,7,16,19,54,74]).

Das System gibt die Antwort „wahr“, da die Behauptung aus den Fakten und Regeln des Programms herleitbar ist. Denn die Folge [2,3,3,5,7,16,19,54,74] ist die gemischte Version der Folgen [3,7,19,54] und [2,3,5,16,74]. Auf die Behauptung

misch([2,5],[1,2,8],[2,5,1,2,8])

gibt das System die Antwort „falsch“, denn die dritte Folge ist nicht die gemischte Version der beiden ersten. Eine andere Behauptung ist

(∃Y) (misch([2,5],[1,2,8],Y)).

In diesem Fall beweist das System die Behauptung und gibt, falls möglich, nacheinander alle Werte Y aus, für die die Aussage misch([2,5],[1,2,8],Y) wahr ist, (hier nur die Folge Y=[1,2,2,5,8]).

Die Zweckfreiheit des prädikativen Programms im deklarativen Sinne wird sehr schön dadurch veranschaulicht, daß man Behauptungen aufstellen kann, die nicht der traditionellen Problemstellung „Gegeben zwei Folgen. Wie lautet die Mischung der beiden?“ entspricht, z.B.:

(∃X,Y) (misch(X,Y,[1,2,5])).

Die Behauptung ist wahr, und das System gibt alle Folgen X,Y aus, deren Mischung

die Folge [1,2,5] ist, also:

X=[ ], Y=[1,2,5],
X=[1], Y=[2,5],
X=[2], Y=[1,5],
X=[5], Y=[1,2],
X=[1,2], Y=[5],
...
X=[1,2,5], Y=[ ].

2) Mischen in PROLOG:

```
misch([ ],Y,Y).
misch(Y,[ ],Y).
misch([A|X],[B|Y],[A|Z]) :– A≤B, misch(X,[B|Y],Z).
misch([A|X],[B|Y],[B|Z]) :– B<A, misch([A|X],Y,Z).
```

Erläuterung: Man beachte die Vertauschung von Prämisse und Konklusion (das Zeichen :– symbolisiert ein stilisiertes <=) und die Elimination der Quantoren ©... . Die Variablen muß man sich hier also implizit allquantifiziert denken. Das Komma in Prämissen entspricht dem logischen UND ^.

Das Programm wird gestartet durch Aufstellen einer Behauptung in der syntaktischen Form

```
?- misch([3,7,19,54],[2,3,5,16,74],[2,3,3,5,7,16,19,54,74]).
```

Ausgabe von PROLOG: yes. Die Aussage ist also wahr. Weitere Behauptung:

```
?- misch([2,5],[1,2,8],Y).
```

Ausgabe von PROLOG:

```
Y=[1,2,2,5,8].
```

Die Ausgabe ist also wahr; eine Belegung von Y, die die Aussage wahr macht, ist Y=[1,2,2,5,8]. Auch bei Behauptungen benötigt PROLOG also keine Quantoren für die Variablen; man muß man sich die Variablen implizit Existenz-quantifiziert vorstellen.

Bemerkung: Wie bereits erwähnt, gibt es zur Zeit keine rein deklarativen Sprachen; auch PROLOG ist im strengen Sinne nicht deklarativ und besitzt zahlreiche nicht-deklarative Elemente. Tatsächlich orientiert sich die Verarbeitung eines PROLOG-Programms an der imperativen Arbeitsweise des Von Neumann-Rechners, d.h. es liegt genau fest, in welcher Reihenfolge Fakten und Regeln zum Beweis einer Behauptung herangezogen werden: Aussagen, die im Programmtext vorne stehen, werden zuerst zum Beweis verwendet. Hierbei kann es zu der unerwünschten Situation kommen, daß PROLOG eine Behauptung nicht beweisen kann, obwohl ein Beweis existiert. Man spricht hier von der *prozeduralen Semantik* von PROLOG-Programmen, die sich von der *deklarativen Semantik* des zugrundeliegenden Prädikatenkalküls unterscheidet.