Stand: Februar 1998 (1)

Siegfried Wendt
Universität Kaiserslautern

# Programmzweck versus Programmform

## Zielsetzung

In diesem Aufsatz geht es um eine Klassifikation von Programmen nach zwei orthogonalen Kriterien. Programm und Software werden dabei nicht als Synonyme angesehen; *Programm sein* wird hier gleichgesetzt mit *ausführbar sein*, d.h. etwas ist dann und nur dann ein Programm, wenn man die Frage beantworten kann, was es denn heißen solle, dieses Etwas werde ausgeführt. Es gibt durchaus Softwaregebilde, bezüglich derer diese Frage keinen Sinn hat und die demzufolge auch keine Programme sind – beispielsweise eine Funktions– oder eine Klassenbibliothek.

Klassifikation ist von Nutzen, wenn sie Vielfalt überschaubarer macht – die Vielfalt der Schüler einer großen Schule wird überschaubarer, wenn die Schüler "klassifiziert" sind, d.h. wenn sie in ihren Klassenzimmern sitzen. Die im folgenden vorgestellte Klassifikation soll die Vielfalt von Programmen überschaubarer machen.

## Ergebnisorientierung versus Prozeßorientierung

Der Zweck eines Programms ist entweder ergebnisorientiert oder prozeßorientiert. Der Nutzen eines Programms ergibt sich erst bei seiner Ausführung; dann erlebt der Nutzer das Verhalten des programmierten Systems.

*Ergebnisorientierung* liegt vor, wenn es demjenigen, der einen Nutzen von dem System haben will, genügt, eine Anfangssituation und eine Endsituation zu betrachten. In der Anfangssituation liefert der Nutzer das nötige Material und erteilt dem System den Auftrag zur Schaffung des Ergebnisses, und in der Endsituation übergibt ihm das System das Ergebnis. Das Ergebnis könnte beispielsweise ein Schrank sein, den ein Schreiner für einen Auftraggeber baut. In der Anfangssituation liefert der Auftraggeber dem Schreiner das erforderliche Holz, und in der Endsituation nimmt er den Schrank entgegen. Im Falle der Ergebnisorientierung hat der Nutzer keinerlei Interesse, den Prozeß der Ergebnisherstellung zu beobachten.

Bei der Informationsverarbeitung liegt Ergebnisorientierung immer dann vor, wenn aus den in der Anfangssituation bereitgestellten Informationen durch Verknüpfung ein Ergebnis gewonnen wird. Ergebnisorientierung äußert sich in diesem Falle immer als Anwendung einer Funktion auf ein gegebenes Argument – also beispielsweise als Berechnung des Wurzelwertes 5 zum Argument 25.

*Prozeßorientierung* liegt vor, wenn der Nutzen des Systems im Prozeß selbst liegt. Dabei ist es völlig unerheblich, ob in diesem Prozeß ein Ergebnis geschaffen wird oder nicht. Wenn kein Ergebnis geschaffen werden muß, braucht der Prozeß auch gar kein definiertes Ende zu haben, d.h. er darf potentiell immer weiterlaufen. Aus einem solchen Prozeß kann man natürlich nur einen Nutzen ziehen, indem man den Prozeß wahrnimmt.

Ein einfaches Beispiel einer Prozeßorientierung liegt vor, wenn ein Orchester ein Konzert gibt. Aus diesem Konzert kann man keinen Nutzen ziehen, wenn man nicht im Saal anwesend ist und zuhört. Im Gegensatz dazu braucht derjenige, der einen Schrank in Auftrag gegeben hat, nicht in der Schreinerwerkstatt anwesend zu sein, um einen Nutzen aus dem Schrank zu ziehen.

Dem Paar Ergebnisorientierung/Prozeßorientierung entspricht in einer Analogie das Paar Warenproduktion/Dienstleistung. Eine Ware kann man herstellen lassen und später abholen, eine Dienstleistung aber nicht. Der Dienst wird an dem Nutzer des Dienstes geleistet: Der Kranke wird gepflegt, der Theaterbesucher wird unterhalten, der Schüler wird unterrichtet.

Da es in der üblichen Betrachtungsweise der Systemtechnik neben dem System nur noch seine Umgebung gibt, muß es also im Falle der Prozeßorientierung die Umgebung sein, die sich vom System eine Dienstleistung erbringen läßt. Als typisches Beispiel bietet sich eine Fahrstuhlanlage an: Zur Umgebung zählen alle materiell–energetisch relevanten Teile, also der Motor, die Fahrkabine, die Stockwerke und die Menschen, die den Fahrstuhl benützen. Das System erbringt eine Dienstleistung, die darin besteht, daß der Motor zur rechten Zeit in richtiger Weise angesteuert wird, daß die Anzeigefelder jeweils die richtigen Informationen anzeigen, daß die Türen zur rechten Zeit auf– bzw. zugemacht werden, u.s.w. In diesem Fall ist also das prozeßorientiert arbeitende System die Steuerung der Fahrstuhlanlage. Sie steht hier zum Rest der Fahrstuhlanlage in der gleichen Beziehung wie ein Dirigent zu seinem Orchester.

Wenn man also nach dem *Zweck eines Programms* fragt, muß als erstes immer festgestellt werden, ob es sich um ein ergebnisorientiertes oder um ein prozeßorientiertes Programm handelt. Innerhalb eines prozeßorientierten Programmes wird es im Normalfall immer Abschnitte geben, die ergebnisorientiert sind.

## Alternative Möglichkeiten der Programmformulierung

Wenn man nach der *Art der Programmformulierung* fragt, betrachtet man das Programm als Sprachgebilde, dessen Kategorie festgestellt werden soll. Obwohl Programme in künstlichen Sprachen formuliert werden, können für ihre Interpretation dennoch keine anderen Kategorien gelten als diejenigen, die man aus dem Bereich der natürlichen Sprachen kennt.

Im Bereich der natürlichen Sprachen unterscheidet man zwischen Ausdrücken und ganzen Sätzen, und bei den Sätzen unterscheidet man zwischen Aussagen, Fragen und Anweisungen. Dementsprechend kann ein Programm

- *imperativ*, d.h. als Anweisung
- *deklarativ*, d.h. als Paar aus Aussage und Frage
- *funktional*, d.h. als Ausdruck

formuliert werden. Für jede dieser Programmformen ist festgelegt, worin die Programmausführung bestehen soll:

- Ein imperatives Programm ist als Anweisung formuliert und wird ausgeführt, indem die Anweisung ausgeführt wird – diese Programmkategorie stand offensichtlich Pate bei der Wortwahl ”Programmausführung”.

- Ein deklaratives Programm ist als Paar aus Aussage und Frage formuliert und wird ausgeführt, indem die Frage beantwortet wird. Dabei können nur solche Fragen beantwortet werden, deren Antwort aus der zuvor gemachten Aussage folgt.

– Ein funktionales Programm ist als Ausdruck formuliert, der in diesem Fall nur eine funktionale Umschreibung [1] sein kann. Ein solches Programm wird ausgeführt, indem das Ergebnis der in der Umschreibung angegebenen Funktion für das ebenfalls in der Umschreibung angegebene Argumenttupel berechnet wird.

Der Autor hat bisher keine ausführbaren Gebilde kennengelernt, die nicht einer dieser drei Sprachkategorien zugeordnet werden konnten. Er kann aber nicht beweisen, daß es solche nicht geben kann.

Es ist durchaus zulässig, daß ein Programm aus Teilen besteht, die zu unterschiedlichen Kategorien gehören. Man betrachte hierzu das Beispiel in Bild 1.

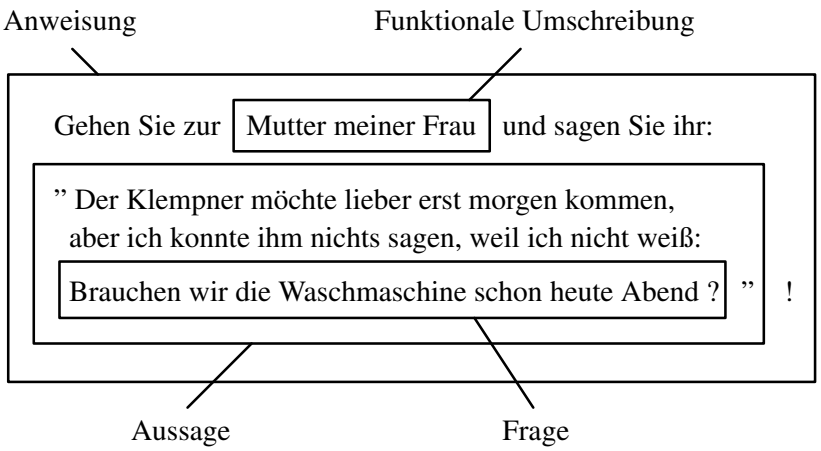


**Bild 1** Beispiel zum Vorkommen unterschiedlicher Kategorien in einem sprachlichen Gebilde

Im folgenden werden die drei Sprachkategorien für Programme näher betrachtet.

## Imperative Programme

Anweisungen verlangen Operationen. Eine *Operation* ist eine Zustandsveränderung. Deshalb kann man mit jeder Operation ein Zustandspaar verbinden, welches aus dem Zustand vor der Operation – z.B. der Blinddarm ist noch drin – und dem Zustand nach der Operation – der Blinddarm ist raus – besteht. Es ist üblich, diese beiden Zustände durch die Bezeichnungen PRE und POST zu unterscheiden.

Man kann zwischen elementaren und zusammengesetzten Anweisungen unterscheiden. In einer zusammengesetzten Anweisung wird die verlangte Operation dadurch umschrieben, daß angegeben wird, wie sich die große Operation aus kleineren Operationen zusammensetzt.

Die Operation, die sich erst bei der Ausführung der Anweisung ergibt, ist i.a. in der Anweisung nicht vollständig bestimmt. Vielmehr schreibt eine Anweisung meist nur den Typ der durchzuführenden Operation vor. Die konkrete Operation wird erst durch den Zustand PRE vollständig festgelegt, der zum Zeitpunkt des Ausführungsbeginns der Anweisung vorliegt.

---

1) Der Begriff *funktionale Umschreibung* wird im Mosaikstein "Zeigen, Nennen, Umschreiben – die drei Alternativen der Identifikation" erklärt.

Beispiele:

(1) Die Anweisung i := i+1 beschreibt als Operationstyp den Vorgang, daß der Inhalt der Speicher–zelle i, von dem angenommen wird, daß er eine ganze Zahl sei, um eins erhöht wird.
Der Inhalt von i zu Beginn der Anweisungsausführung ist der Zustand PRE, und der Inhalt von i am Ende der Operation ist der Zustand POST.
Weil in der Anweisung keine vollständige Operationsbeschreibung steht, können sich aufgrund der Ausführung der gleichen Anweisung zu unterschiedlichen Zeitpunkten unterschiedliche Operationen ergeben, z.B. $(i_{PRE}, i_{POST})_1 = (5, 6)$ und $(i_{PRE}, i_{POST})_2 = (12, 13)$.

(2) Die Anweisung "Lasse eine Minute lang die Sirene laufen!" ist eine prozeßorientierte Anweisung, denn der Nutzen der Anweisungsausführung liegt nicht im Herbeiführen des Zustands POST, sondern in den Erscheinungen im Intervall zwischen PRE und POST. Die Zustände PRE und POST unterscheiden sich nur in der Uhrzeit: Im PRE–Zustand ist es ruhig und im POST–Zustand auch.
Die in der Anweisung verlangte Operation ist eine Sequenz von drei einfacheren Operationen. In der folgenden Darstellung sind nicht nur die Anweisungen, sondern auch die jeweiligen PRE– und POST–Zustände angegeben:

(Uhrzeit t, Ruhe)
Schalte die Sirene ein!
(Uhrzeit t, Sirene läuft)
Warte eine Minute!
(Uhrzeit t + 1 min, Sirene läuft)
Schalte die Sirene aus!
(Uhrzeit t + 1 min, Ruhe)

Durch diese Beispiele wird deutlich, daß die Kategorien *Ergebnisorientierung* und *Prozeßorientierung* nicht nur für Programme, sondern auch für Anweisungen gelten.

Durch Verknüpfung von elementaren Anweisungen kann man zusammengesetzte Anweisungen bilden. Es gibt verschiedene Arten von Verknüpfungen von Anweisungen.

Sind die Anweisungen *folgenverknüpft*, so bedeutet dies, daß die Anweisungen nacheinander in der festgelegten Reihenfolge auszuführen sind.

Ein Sonderfall der Folgenverknüpfung ist die *Wiederholung*. In diesem Fall wird verlangt, daß eine Anweisung mehrfach hintereinander ausgeführt wird. Man unterscheidet in diesem Fall zwischen *anzahlbegrenzten* und *ergebnisbegrenzten* Wiederholungen. Bei einer anzahlbegrenzten Wiederholung ist zu Beginn der Ausführung der Wiederholung schon bekannt, wie oft die einzelne Anweisung zu wiederholen ist. Bei der ergebnisbegrenzten Wiederholung dagegen wird verlangt, daß die Anweisung so oft wiederholt werden soll, bis ein bestimmtes Ergebnis erreicht ist. Während eine anzahlbegrenzte Wiederholung garantiert in endlicher Zeit ausführbar ist, kann es bei einer ergebnisbegrenzten Wiederholung vorkommen, daß trotz beliebig häufiger Wiederholung der Anweisung das Ergebnis nie erreicht wird.

Eine weitere Verknüpfungsart ist die Alternativenverknüpfung. Sind die elementaren Anweisungen *alternativenverknüpft*, so besteht die Ausführung der zusammengesetzten Anweisung darin, aus der Menge der alternativenverknüpften Anweisungen eine auszuwählen und diese auszuführen. Bei einer Alternativenverknüpfung muß also angegeben werden, wie der aktuelle Zustand PRE eindeutig die Auswahl der auszuführenden Anweisung festlegt.

Sind die Anweisungen *mengenverknüpft*, so bedeutet dies, daß alle Anweisungen ausgeführt werden sollen, wobei aber keine Ausführungsreihenfolge vorgeschrieben wird; in diesem Fall ist eine *nebenläufige Ausführung* der Anweisungen möglich.

In Bild 2 sind die vier unterschiedlichen Arten von Anweisungsverknüpfungen durch Petrinetze veranschaulicht. Dabei entspricht das Schalten einer großen Transition, die jeweils durch das umfassende Rechteck symbolisiert ist, der Ausführung der zusammengesetzten Anweisung. Das Schalten einer Transition $a_i$ entspricht der Ausführung einer elementaren Anweisung.

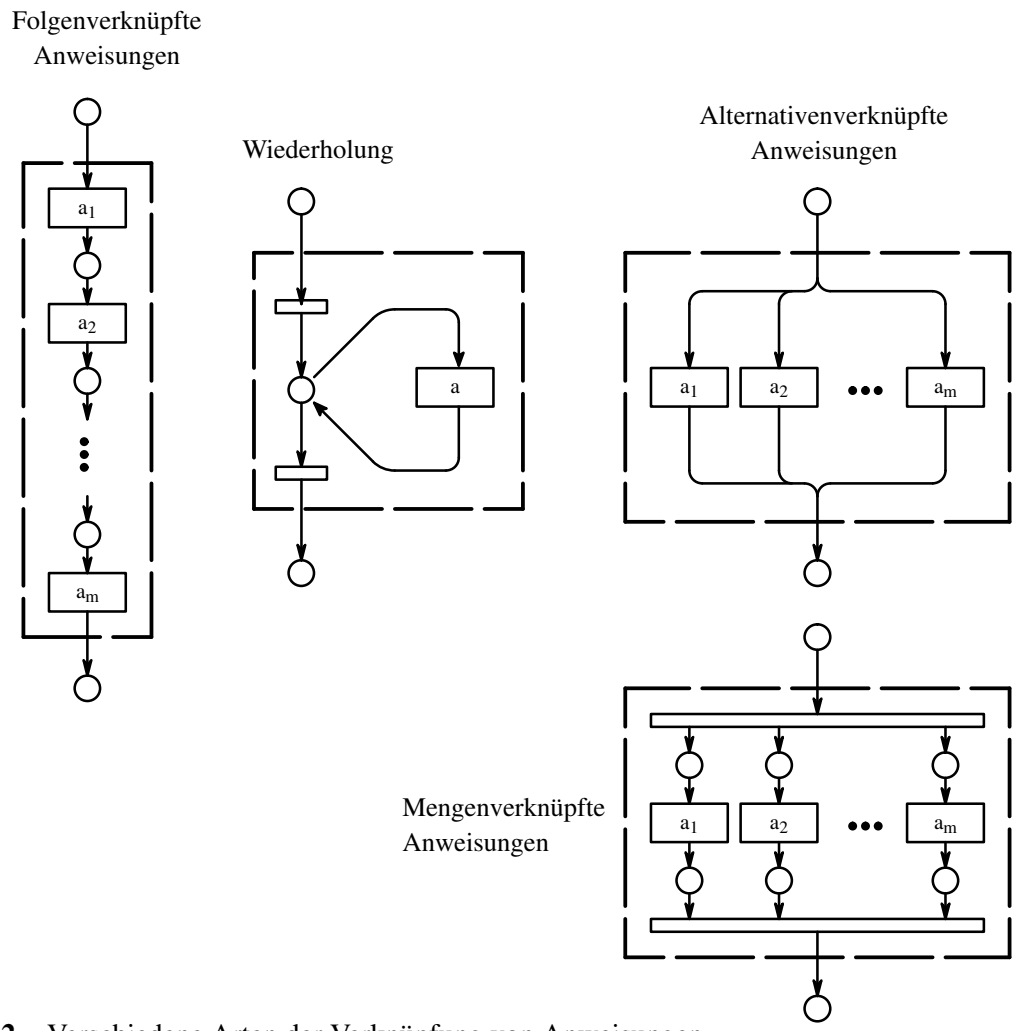


**Bild 2** Verschiedene Arten der Verknüpfung von Anweisungen

In den Petrinetzen für die Wiederholung und für die Mengenverknüpfung kommen neben den elementaren Transitionen $a_i$ noch jeweils zwei unbeschriftete Transitionen vor. Diese Transitionen werden gebraucht, damit jeweils das umfassende Rechteck als eine Transition in einem Petrinetz gesehen werden kann, die in einem Verfeinerungsschritt in die gezeigte Struktur überführt wird.

Die dargestellten vier Verknüpfungsarten sind selbstverständlich nicht nur auf elementare Anweisungen anwendbar, d.h. man kann auf diese Weise auch zusammengesetzte Anweisungen verknüpfen. Wenn ein als Anweisung formuliertes Programm derart aus Elementaranweisungen zusammengesetzt wird, daß ausschließlich die vier Verknüpfungsarten verwendet werden, dann läßt sich für dieses Programm ein Strukturbaum angeben. Die Blätter dieses Baumes sind die elementaren Anweisungen $a_i$, und die höheren Knoten sind zusammengesetzte Anweisungen $A_j$. Die Wurzel, d.h. die Anweisung $A_0$ ist das gesamte Programm. Bild 3 zeigt hierzu ein Beispiel.

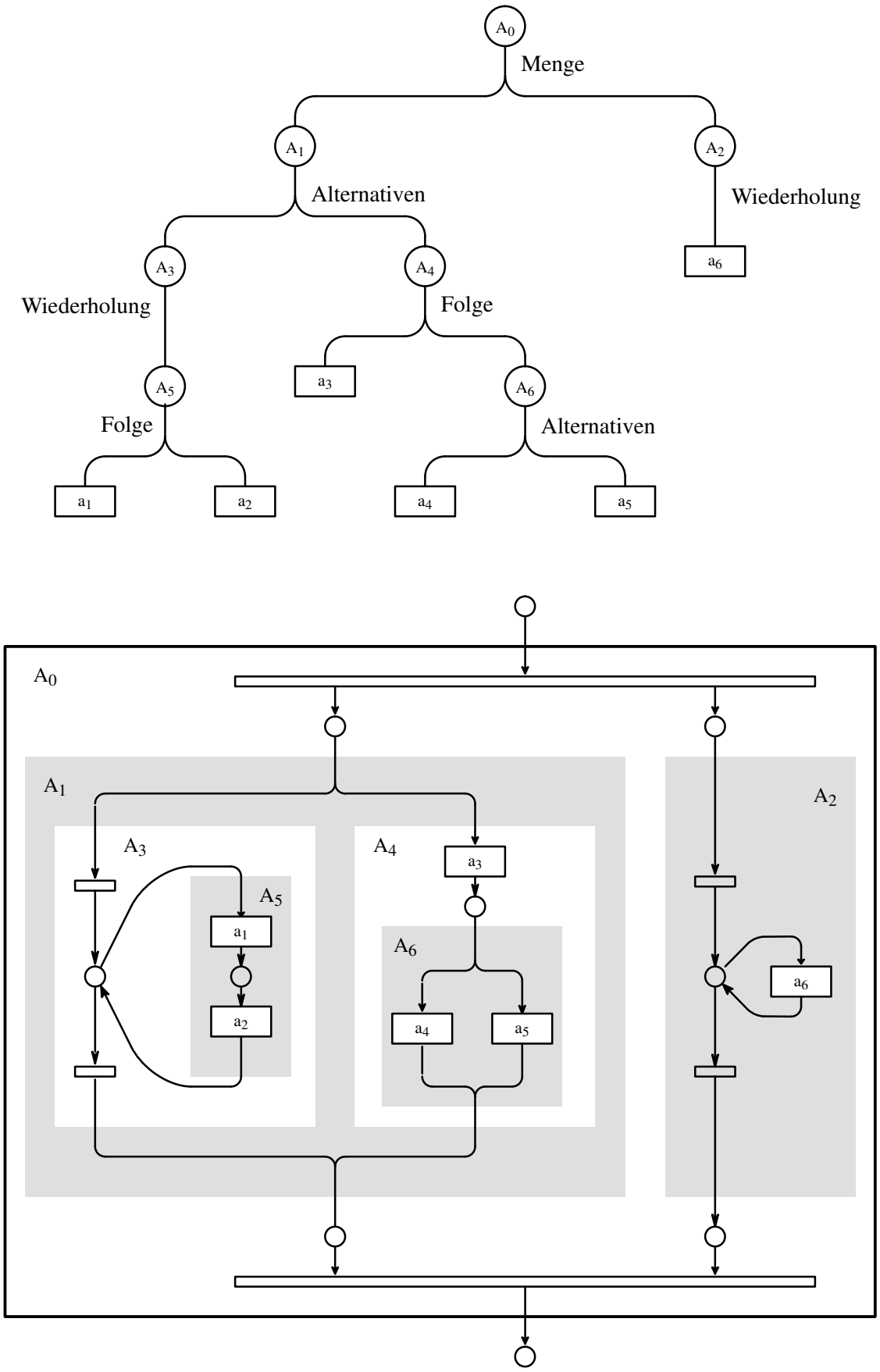


**Bild 3** Beispiel eines Programms, das durch sukzessive Verknüpfung von Anweisungen aufgebaut wurde; oben der Strukturbaum, unten das Petrinetz

Es ist möglich, Anweisungssprachen so zu konstruieren, daß man mit diesen Sprachen neben den Elementaranweisungen nur die vier eingeführten Verknüpfungsarten formulieren kann. Dann kann man mit diesen Sprachen nur Anweisungen zusammensetzen, für die sich ein Strukturbaum angeben läßt.

Man kann aber auch Sprachen konstruieren, mit denen man zusammengesetzte Anweisungen formulieren kann, für die es keinen Strukturbaum gibt. Bei der Verwendung solcher Sprachen kann man nicht mehr sinnvollerweise von einer *Verknüpfung*, sondern nur noch von einer *Vernetzung* von Anweisungen sprechen. Das Wesen einer Verknüpfung äußert sich in der Existenz eines Strukturbaumes: Aus Blättern werden durch Verknüpfung höhere Gebilde, und aus diesen werden durch weitere Verknüpfung noch höhere Gebilde, bis man schließlich bei der Wurzel des Baumes angelangt ist. Bei einer Vernetzung dagegen entsprechen die elementaren Bausteine den Knoten eines Netzes, und die Netzstruktur entsteht dadurch, daß entschieden wird, von welchen Knoten es Wege zu welchen anderen Knoten geben soll. Bild 4 zeigt hierzu ein Beispiel.

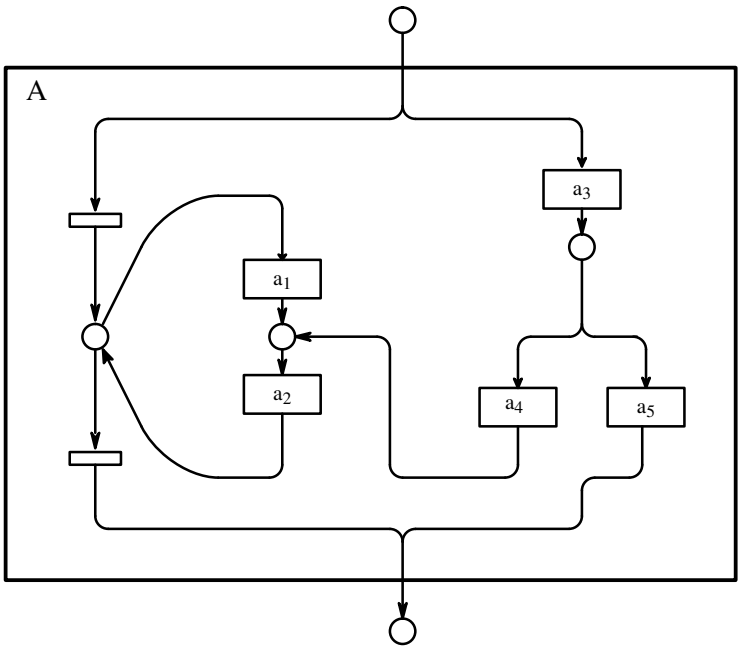


**Bild 4** Beispiel einer Anweisungsvernetzung, die keine Anweisungsverknüpfung ist

In diesem Beispiel ist die Netzstruktur einer Anweisung A gezeigt, die aus der Struktur der Anweisung $A_1$ aus Bild 3 dadurch entstanden ist, daß eine einzige Netzkante umgebogen wurde. Für die Anweisung A in Bild 4 läßt sich kein Strukturbaum angeben.

Es bringt viele Vorteile, wenn man ein Programm, das man als Anweisung formuliert, so aufbaut, daß sich ein Strukturbaum angeben läßt. Im Rahmen des vorliegenden Aufsatzes soll jedoch nicht dargestellt werden, worin diese Vorteile bestehen.

Die Elementaranweisungen, die bisher betrachtet wurden und die als Transitionen $a_i$ in den Petrinetzen und als Blätter im Strukturbaum vorkommen, dürfen nicht verwechselt werden mit den sogenannten *Metaanweisungen* [2)], die der Formulierung von Anweisungsvernetzungen dienen. Die Elementaranweisungen kann man auch als elementare Operationsanweisungen bezeichnen, denn sie verlangen die Ausführung von Operationen, die als unmittelbare Beiträge zur Realisierung des Programmzwecks angesehen werden können. Demgegenüber verlangen die Metaanweisungen von dem Ausführenden des Programms nur, daß er seine Aufmerksamkeit einem anderen Teil der Programmaufscheibung zuwendet, d.h. daß er an einen anderen Ort schaut.

Man denke zum Beispiel an ein Konzert. Das Programm ist in diesem Fall in Notenschrift formuliert, und jede Note stellt eine elementare Operationsanweisung dar, z.B. "Spielen Sie für die Dauer eines Vierteltaktes den Ton d !" Eine Metaanweisung könnte im Falle der Noten lauten: "Setzen Sie jetzt Ihr Spiel fort mit dem, was ab der Stelle P steht !"

Das GOTO–Statement in höheren Programmiersprachen und Sprunganweisungen in Assemblersprachen sind derartige Metaanweisungen.

---

2) Im Begriffspaar *Physik/Metaphysik* bezeichnet Metaphysik etwas, was jenseits der Physik liegt und somit gerade keine Physik mehr ist. Im Gegensatz dazu versteht man in Begriffspaaren wie *Sprache/Metasprache*, *Regel/Metaregel*, *Wunsch/Metawunsch* oder *Anweisung/Metaanweisung* unter dem jeweiligen MetaXXX ein XXX, dessen Gegenstand oder Thema selbst auch ein XXX ist. Beispiele:
   (1) Wenn man in deutscher Sprache über die Programmiersprache SMALLTALK spricht, ist SMALLTALK das Thema oder die Gegenstandssprache, und Deutsch ist die Metasprache, in der man über die Gegenstandssprache spricht.
   (2) Die abschließende Vorschrift einer Prüfungsordnung, die besagt, daß diese Prüfungsordnung am Tag nach ihrer Veröffentlichung im Staatsanzeiger in Kraft tritt, ist eine Metavorschrift, denn sie besagt etwas über die Gültigkeit von Vorschriften und ist somit eine Vorschrift, die Vorschriften zum Thema hat.
   (3) Wenn derjenige, dem eine Fee im Märchen drei Wünsche gewährt, als erstes wünscht, daß die Begrenzung der Anzahl der Wünsche wegfallen solle, dann äußert er einen Metawunsch, denn sein Wunsch hat Wünsche zum Thema.

## Deklarative Programme

Das Programm besteht aus einem Paar aus Aussage und Frage. Die Aussage – es wird sich im allgemeinen um eine zusammengesetzte Aussage handeln, d.h. um eine Aussage, die als Verknüpfung elementarer Aussagen formuliert ist – wird üblicherweise als *Wissensbasis* bezeichnet, und anstelle des Wortes 'Frage' wird üblicherweise das Wort *Anfrage* benutzt. Wenn man sagt, daß eine Wissensbasis aus *Wissenselementen* bestehe, meint man, daß die Wissensbasis eine Aussage ist, die durch konjunktive Verknüpfung aus einfacheren Aussagen gewonnen wird; diese konjunktiv verknüpften Aussagen werden als Wissenselemente bezeichnet.

Zur Formulierung von Aussagen bzw. Aussageformen stehen grundsätzlich die Sprachmittel der Aussagenlogik und der Prädikatenlogik zur Verfügung. Wünschenswert wäre es, alle Möglichkeiten der Prädikatenlogik erster Stufe verwenden zu können. Dann müßte man aber in Kauf nehmen, daß die Ausführbarkeit eines mit diesen Sprachmitteln formulierten Programms nicht mehr garantiert werden könnte, und daß eine gegebene Nichtausführbarkeit nicht in jedem Falle zweifelsfrei festgestellt werden könnte. Dies ist eine Konsequenz der Nichtentscheidbarkeit des Prädikatenkalküls erster Stufe. Die Nichtentscheidbarkeit besagt in diesem Falle, daß man keine Maschine bauen kann, die zu jeder mit den Mitteln des Prädikatenkalküls erster Stufe formulierbaren Aussage nach endlich vielen Schritten angibt, ob die Aussage logisch wahr[3)] ist oder nicht.

Zweckmäßigerweise wird man also zur Formulierung von Programmen, die aus einer Wissensbasis und einer Anfrage bestehen, nicht alle Möglichkeiten des Prädikatenkalküls erster Stufe zulassen, sondern man wird sich auf geeignete Weise einschränken. Die Frage, wie man sich zweckmäßigerweise einschränken sollte, wird hier nicht behandelt. Es werden aber weiter unten einige Beispiele für die Formulierung von Wissenselementen und Anfragen angegeben, wie sie in der Programmiersprache PROLOG möglich sind.

Eine Aussage, die ohne Quantifikation und damit variablenfrei formuliert ist, bezeichnet man als *elementares Faktum*. Durch elementare Fakten werden entweder faktisch wahre Sachverhalte ausgedrückt oder vorgegebene Axiome. Auch Axiome können als Sachverhalte angesehen werden, die aber nicht die beobachtbare Welt betreffen, sondern formal konstruierte abstrakte Welten. Mit einem Faktum verbindet man keine Erkenntnis, sondern nur Wissen. So wird man beispielsweise nicht von einer Erkenntnis sprechen, wenn gesagt wird, daß Johann Wolfgang v. Goethe im Jahre 1832 gestorben sei. Auch das Axiom, welches besagt, daß 1 eine natürliche Zahl sei, drückt keine Erkenntnis aus, sondern eine Festlegung.

Eine Aussage, die mit Quantifikatoren und damit mit Individuenvariablen formuliert ist, bezeichnet man als *Regel*. Mit dem Begriff Regel verbindet man intuitiv die Vorstellung einer allgemeingültigen Erkenntnis, und in der Formulierung erkennt man dies am Vorkommen eines Quantifikators.

Eine Regel kann eine logische Regel oder eine faktische Regel sein. Eine *logische Regel* ist aus sprachlichen Gründen wahr, wogegen sich die Wahrheit einer *faktischen Regel* auf Sachverhalte der äußeren

---

3) Man unterscheidet zwischen faktischer und logischer Wahrheit.
Faktische Wahrheit kann nur durch Beobachtungen begründet werden, z.B. daß das spezifische Gewicht von Eisen größer ist als das von Aluminium, oder daß Goethe im Jahre 1832 starb.
Logische Wahrheit dagegen ist die Konsequenz sprachlicher Vereinbarungen und kann deshalb ausschließlich formal aus der sprachlichen Form der Aussage geschlossen werden. Wer die zu den Elementen der Symbolmenge { 2, 3, 5, +, = } vereinbarten Bedeutungen kennt, muß auch die Wahrheit der Aussage 2+3=5 anerkennen.

Welt gründet. Daß sich die Erde alle 24 Stunden einmal um ihre Achse dreht oder daß alle Kinder der Frau Anna männlich sind, sind "regelhafte Fakten", d.h. Fakten, die unter Verwendung von Quantifikatoren formuliert werden.

In logischen Regeln wird nicht auf irgend etwas Beobachtbares Bezug genommen. Wenn beispielsweise gesagt wird, daß jede Elternteil–Kind–Beziehung, in der beide Beteiligten männlich sind, eine Vater–Sohn–Beziehung sei, so ist dies keine Erkenntnis aus der Beobachtung, sondern lediglich eine Konsequenz der Bedeutung der Wörter 'Elternteil', 'Kind', 'männlich', 'Vater' und 'Sohn'.

In Fakten und Regeln muß man vier unterschiedliche Arten von Dingen benennen, d.h. in den Formeln kommen Namen für vier unterschiedliche Arten von Dingen vor:

Namen für Individuen, z.B. ANNA; PETER

Namen für Individuenvariable, z.B. $k$, $m$

Namen für Funktionen, z.B. MutterVon, SohnVon

Namen für Prädikate, z.B. Gleich, Männlich, Elternteil–Kind, Vater–Sohn

Beispiele:

Elementare Fakten

(1) Männlich (PETER)

(2) Gleich ( ANNA, MutterVon (PETER) )

Faktische Regel:

(3) $\forall k$: Elternteil–Kind (ANNA, $k$) $\rightarrow$ Männlich ($k$)

Logische Regel:

(4) $\forall e, k$: Elternteil–Kind ($e$, $k$) $\cdot$ Männlich ($e$) $\cdot$ Männlich ($k$) $\rightarrow$ Vater–Sohn ($e$, $k$)

Wenn man nach den Möglichkeiten zur Formulierung von Fragen sucht, erkennt man bald, daß jede Frage als eine mit einem Fragezeichen abgeschlossene Aussage oder Aussageform formuliert werden kann. Eine Frage ist nämlich entweder eine *Behauptungsfrage* oder eine *Selektionsfrage*.

Eine Behauptungsfrage hat die umgangssprachliche Form: "Ist die Behauptung *xxx* wahr ?" Man könnte aber auch kürzer "*xxx* ?" schreiben, denn die umgebenden Wörter kann man sich dazudenken.

Selektionsfragen in umgangssprachlicher Form beginnen i.a. mit einem Fragewort: "Wer hat ... ?" oder "Wann wurde .... ?" oder "Wo steht ... ?" . Diese Formen kann man verallgemeinern zu der Form "Welche Individuentupel machen die Aussageform *P*(*Tupel der Argumentvariablen*) wahr ?", und dies wiederum kann man durch Weglassen der implizit assoziierbaren Wörter verkürzen auf " *P*(*Tupel der Argumentvariablen*) ? ".

Beispiele:

Behauptungsfrage:

(5) Männlich (ANNA) ? d.h. umgangssprachlich: "Ist ANNA männlich?"

Selektionsfragen:

(6) Vater–Sohn (PETER, $k$) ? d.h. "Welche Söhne hat PETER?"

(7) GLEICH ( ANNA, MutterVon ($k$) ) $\cdot$ Männlich ($k$) ? d.h. "Welche Söhne hat ANNA?"

Es wurde gesagt, daß die Ausführung eines Programms, welches aus einer Wissensbasis und einer Anfrage aufgebaut ist, darin bestehe, daß die Anfrage beantwortet wird und daß sich dabei die Antwort auf die Wissensbasis gründen müsse. Dies legt unmittelbar die Frage nahe, ob man aus dem Nichtgesagten etwas schließen dürfe. Man kann zu diesem Problem unmittelbar den albernen Witz assoziieren, bei dem jemand eine Reihe von Fakten über ein Schiff bekanntgibt, beispielsweise daß das Schiff 120 m lang sei, von Hamburg nach New York fahre und einer griechischen Reederei gehöre, worauf dann anschließend gefragt wird, wie alt der Kapitän sei.

Grundsätzlich sollten Systeme, die Programme aus Wissensbasis und Anfrage ausführen, derart gestaltet sein, daß sie aus dem Nichtgesagten keinerlei Schlüsse ziehen. Das bedeutet, daß solche Systeme auch in der Lage sein sollten, auf die Beschränktheit ihres Wissens hinzuweisen. Dies müßte sich dann derart äußern, daß ein solches System auf eine Behauptungsfrage auch die Antwort "Das weiß ich nicht." geben kann, und daß die Antwort auf eine Selektionsfrage im allgemeinen die Form hat: "Von den folgenden Individuentupeln.....weiß ich, daß sie das angefragte Prädikat erfüllen, von den folgenden Individuentupeln....weiß ich, daß sie das Prädikat nicht erfüllen, und bezüglich aller anderen möglichen Individuentupel weiß ich nicht, ob sie das Prädikat erfüllen."

Es ist jedoch gar nicht leicht, ein System zu bauen, welches diese Fähigkeiten hat. Die heutigen PROLOG–Systeme haben sie nicht.

## Funktionale Programme

Ein funktionales Programm ist eine funktionale Umschreibung und besteht deshalb im Normalfall aus zwei Teilen – einem Teil, der eine Funktion f( ... ) festlegt, und einem anderen Teil, der das Argumenttupel festlegt. Im Sonderfall der nullstelligen Funktionen besteht das Programm nur aus einem Teil, nämlich der Angabe der Funktion, die eine Konstante darstellt.

Die Klassifikation funktionaler Umschreibungen in primitiv funktionale und komponiert funktionale Umschreibungen ist auf die funktionalen Programme zu übertragen. In komponiert funktionalen Programmen kommen zwangsläufig definierende Funktionsumschreibungen vor. Für die Formulierung definierender Funktionsumschreibungen hat man eine große Vielfalt von Möglichkeiten. So ist es auch möglich, in definierenden Funktionsumschreibungen Anweisungen zu verwenden. Jede imperative Programmiersprache bietet die Möglichkeit, Funktionen unter Verwendung von Anweisungen zu formulieren.

Beispiel: Imperative Definition der Funktion f(a, b)=a+2b

```
define REAL function f (a:REAL, b:REAL)
    local variable accu:REAL
    begin
      accu := b
      accu := 2 * accu
      accu := accu + a
      return (accu)
    end
```

Wenn man von *rein funktionalen Programmen* spricht, schließt man die Möglichkeit aus, daß definierende Funktionsumschreibungen vorkommen, die mit Hilfe von Anweisungen formuliert sind.

Neben den funktionalen Programmen gibt es auch *scheinfunktionale Programme*. Diese sind dadurch gekennzeichnet, daß sie prozeßorientierte Abschnitte enthalten, die in die syntaktische Form von funktionalen Umschreibungen gekleidet sind.

Beispiel:

Funktionales Programm :

```
f(i) = DIV ( ADD(i, 1), SUB (12, DIV (DOUBLE(i), SQRT(i) ) ) )
  (i) = (25)
```

Die Ausführung liefert als Ergebnis die Zahl 13.

Strukturgleiches scheinfunktionales Programm:

```
f(i) = SEQ ( SET(i, 1), LOOP(12, SEQ (PRINT (i), INCR(i) ) ) )
  (i) = (irrelevant)
```

Bei der Ausführung werden nacheinander die natürlichen Zahlen 1 bis 12 gedruckt. Während *i* im funktionalen Programm eine Variable im mathematischen Sinne bezeichnet, ist *i* im scheinfunktionalen Programm die Bezeichnung einer Speicherzelle, und deren Anfangsbelegung ist für die Programmausführung irrelevant.

## Vergleichende Betrachtungen

Das sich aufgrund der beiden eingeführten Klassifikationskriterien ergebende Klassifikationsschema für Programme läßt sich in Form einer Matrix darstellen. Diese ist in Bild 5 gezeigt. In dieser Matrix sind zwei Felder ausgestrichen, denn die Formulierung von prozeßorientierten Programmen erfordert die Verwendung von Anweisungen, in denen Operationen verlangt werden, und Anweisungen stehen in rein funktionalen oder rein deklarativen Sprachen nicht zur Verfügung.

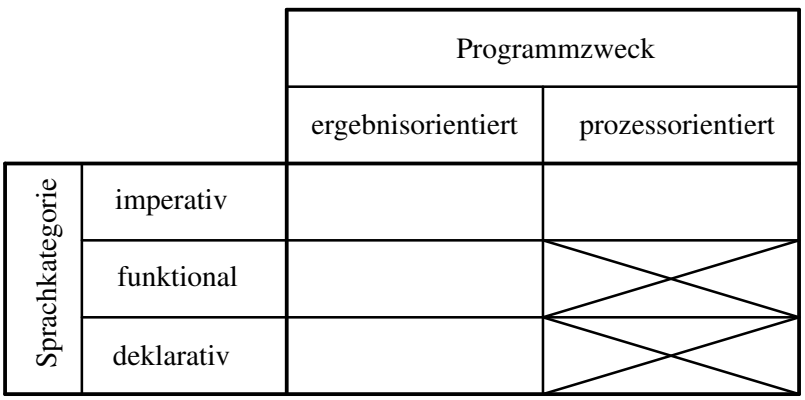


**Bild 5** Klassifikationsschema für Programme

In Sprachen, in denen keine Anweisungen zur Verfügung stehen, kann auch der Begriff der Speicherzelle nicht relevant sein. Speicherzellen braucht man als Orte, an denen die Zustände PRE und POST beobachtet werden. In diesem Sinne ist auch das Papier, welches bei der Ausführung einer Druckanweisung seinen Zustand ändert, eine Speicherzelle. Die Relevanz des Begriffs der Speicherzelle ist ein Merkmal imperativer Sprachen.

Praktisch gibt es bei allen Aufgaben, die durch Programmierung zu lösen sind, einen Bedarf an identifizierbaren Speicherzellen und an der Möglichkeit, explizit zu verlangen, daß bestimmte End– oder Zwischenergebnisse in solchen Speicherzellen abgelegt werden. Das Denken in PRE und POST ist sehr natürlich, und schon bei einfachsten Aufgaben ist dieses Denken unerläßlich – man denke an den Wunsch, berechnete Ergebnisse ausdrucken zu lassen.

Die imperativen Sprachen sind also die universellen Sprachen, und in diese Sprachen lassen sich funktionale oder deklarative Anteile ohne "Verrat an den Konzepten" einbringen. Denn rechts vom Zuweisungsoperator könnte grundsätzlich ein beliebiges ergebnisorientiertes Programm stehen:

Speichervariable := Ergebnisorientiertes Programm

Bei den bekannten imperativen Sprachen sind diese rechts vom Zuweisungsoperator zugelassenen ergebnisorientierten Programme auf primitiv funktionale Programme beschränkt, z.B.

v := SQRT ( ABS ( SIN ( (x+0.338) / 2.0 ) ) )

Solche Einschränkungen können aber lediglich mit praktischen Erwägungen begründet werden, d.h. es gibt dafür keine logisch zwingenden Gründe.

Im Gegensatz hierzu ist die Formulierung prozeßorientierter Programme mit funktionalen und deklarativen Sprachen kategoriell unmöglich. Anstatt nun aber, was nahe läge, die imperativen Sprachen funktional und deklarativ anzureichern, hat man alle funktionalen und deklarativen Sprachen imperativ angereichert und damit eine kategorielle Verwirrung gestiftet. Die sprachliche Form

PRINT (Ergebnis)

bleibt in der Kategorie der Anweisungen, auch wenn man sie als Funktion oder als Prädikat bezeichnet – so wie ein Löwe ein Tier bleibt, auch wenn man ihn "Hustensaft" nennt.

Neben der Beschränkung auf Ergebnisorientierung und dem Fehlen des Begriffs der Speichervariablen gibt es noch eine weitere Gemeinsamkeit bei den rein funktionalen und rein deklarativen Sprachen. Eine Funktionsformulierung und eine Prädikatsformulierung haben grundsätzlich die gleiche syntaktische Form, nämlich

Name (Argument$_1$, Argument$_2$, ... Argument$_m$) [4)]

wobei die Argumente entweder selbst auch wieder diese syntaktische Form haben oder elementare Individuenbezeichner sind.

Die Ausführung von Programmen, die mit solchen syntaktischen Formen formuliert sind, kann man sich sehr einfach als Belegung der Knoten des zugehörigen Ableitungsbaums vorstellen. Bild 6 zeigt hierzu ein Beispiel: Das funktionale Programm ist in diesem Beispiel eine einfache primitiv funktionale Umschreibung der Zahl 13. Im Ableitungsbaum kommen die Superzeichen P für Programm, F für Funktion und Z für Zahl vor. Die Grammatik selbst ist nicht angegeben; der Leser wird auf den Mosaikstein "Die Begriffswelt des Formalen" verwiesen.

---

4) Selbstverständlich ist diese syntaktische Form nicht zwingend. Eine äquivalente Form ist die sog. Listenform, die in der Programmiersprache LISP verwendet wird:

( Name, Argument$_1$, Argument$_2$, ... Argument$_m$ )

Die Gültigkeit der Aussagen bezüglich der Bedeutung des Ableitungsbaums für die Programmausführung hängt jedoch nicht von solchen syntaktischen Details ab.

Während es sehr leicht möglich ist zu zeigen, daß der Ableitungsbaum die Basis für die Ausführung funktionaler Programme darstellt, kann dies für den Fall deklarativer Programme im Rahmen dieses Aufsatzes nicht gezeigt werden. Möglicherweise erscheint es aber dem Leser auch ohne ausführliche Darstellung plausibel, daß für die Ablage von Zwischenergebnissen, die bei der Ausführung eines deklarativen Programms anfallen, der Ableitungsbaum die geeignete Struktur ist, da jedes dieser Zwischenergebnisse eindeutig einem Knoten des Ableitungsbaumes zugeordnet werden kann.

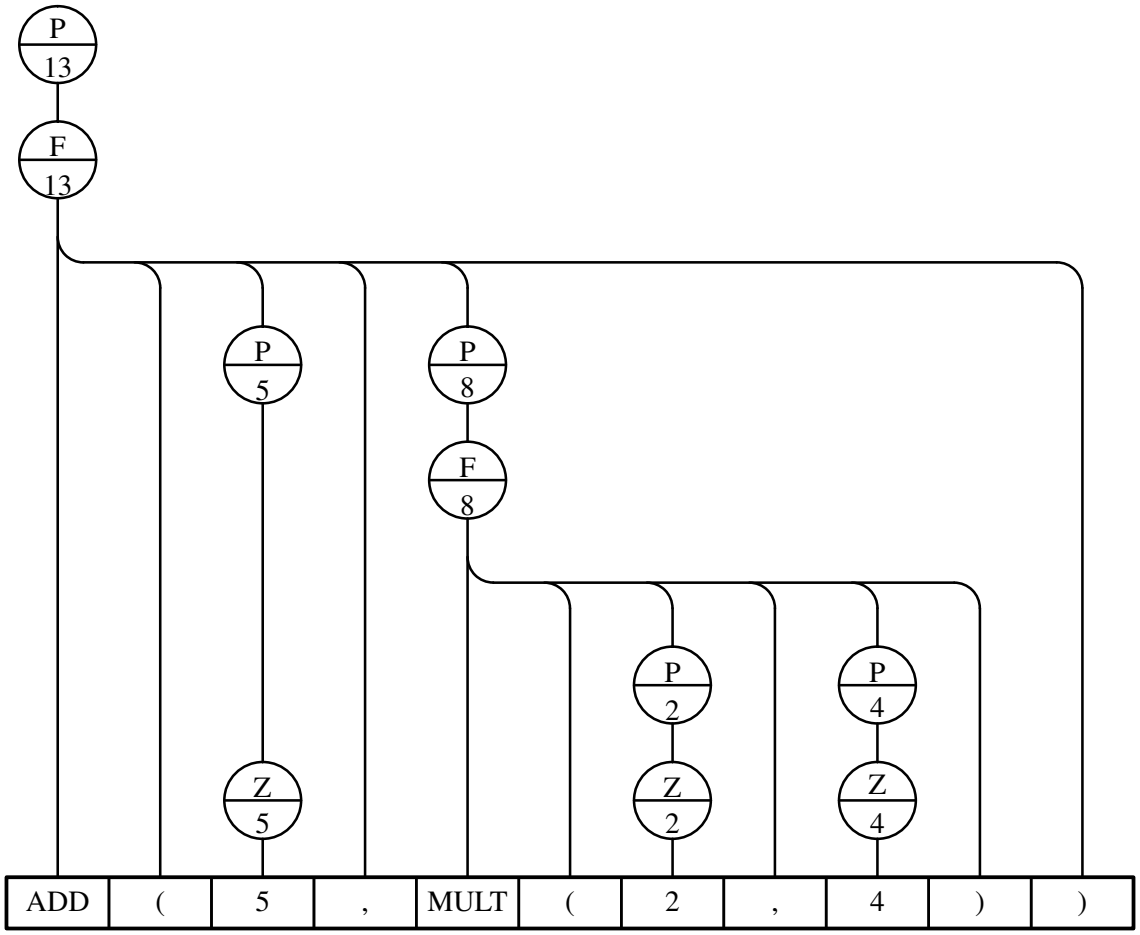


**Bild 6** Ableitungsbaum eines funktionalen Programms als Basis der Programmausführung

Dagegen ist im Falle imperativer Programme der Ableitungsbaum keine geeignete Basis für die Programmausführung. In diesem Fall entsteht der Ableitungsbaum nur als Zwischenstruktur bei der Programmübersetzung. Mit den Bildern 2, 3 und 4 wurde bereits gezeigt, daß das Petrinetz die geeignete Basis für die Ausführung imperativer Programme ist.

Nachdem nun die Gemeinsamkeiten funktionaler und deklarativer Programme aufgezeigt wurden, wozu man formal die Partitionsbildung

{ { imperativ } , { funktional, deklarativ } }

assoziieren kann, soll noch dargestellt werden, daß auch die andere Partitionsbildung

{ { imperativ, funktional } , { deklarativ } }

begründet werden kann, d.h. daß es auch bestimmte Gemeinsamkeiten bei imperativen und funktionalen Programmen gibt, die sie gegen die deklarativen Programme abgrenzen.

Sowohl einem imperativen Programm als auch einem funktionalen Programm kann der Programmausführende explizit entnehmen, was er zu tun hat. Zwar kommen nur in imperativen Programmen explizite Anweisungen vor, aber auch dem funktionalen Programm kann der Ausführende mühelos entnehmen, welche einzelnen Schritte er ausführen muß. Er muß nach der selbstverständlichen Regel verfahren, daß zuerst die Argumentbelegungen bestimmt werden müssen, bevor eine Funktionsauswertung erfolgen kann. Deshalb muß er mit der Berechnung der inneren Funktionen beginnen und sich sukzessive nach außen vorarbeiten, bis er zur Auswertung der äußersten Funktion kommt. Im Beispiel des Bildes 6 heißt dies, daß zuerst die Multiplikation ausgeführt werden muß, bevor addiert werden kann.

Im Gegensatz hierzu erhält derjenige, der ein deklaratives Programm ausführen soll, aus dem Programm keinerlei Hinweise darauf, welche einzelnen Schritte er nebenläufig oder nacheinander ausführen soll. Die Konstruktion einer Maschine, die deklarative Programme ausführen kann, liegt somit sehr viel weniger auf der Hand als die Konstruktion von Maschinen zur Ausführung imperativer oder funktionaler Programme. Man kann ein deklaratives Programm als "ein Gleichungssystem mit Unbekannten" ansehen und die Programmausführung als Auflösung dieses Gleichungssystems. Einem Gleichungssystem sieht man i.a. überhaupt nicht an, was man tun muß, um es nach den Unbekannten aufzulösen.

## Einordnung der Objekt–Orientierung

Es würde den Rahmen des vorliegenden Aufsatzes sprengen, wenn hier eine grundlegende Einführung in die Begriffswelt der Objekt–Orientierung gegeben werden sollte – hierzu wird auf den Mosaikstein "Abstrakte Datentypen versus Objektklassen" verwiesen. Es muß aber im vorliegenden Kontext doch kurz auf die Objekt–Orientierung eingegangen werden, weil die Frage naheliegt, ob denn die objekt–orientierten Programme nicht außerhalb der drei vorgestellten Sprachkategorien liegen und man deshalb die Tabelle in Bild 5 um eine vierte Zeile erweitern müsse.

Die Antwort auf diese Frage lautet: Die objekt–orientierten Programme gehören zur Kategorie der imperativen Programme, denn es handelt sich auch hier um Programme, die als Anweisungen formuliert sind. Das Merkmal *objekt–orientiert* bezieht sich auf die Art und Weise, wie die Anweisungen formuliert werden – genauer gesagt, wie in den Anweisungen die Operatoren und Operanden identifiziert werden. Die Objektorientierung ist eine Alternative zu einer anderen imperativen Formulierungsart, die man *operator–orientiert* nennen könnte. Dies wird im oben genannten Mosaikstein ausführlich behandelt.