

HANSER

Informatik-Handbuch

Peter Rechenberg, Gustav Pomberger

ISBN 3-446-40185-7

Leseprobe

Weitere Informationen oder Bestellungen unter
<http://www.hanser.de/3-446-40185-7> sowie im Buchhandel

2 Programmiersprachen

G. Goos, W. Zimmermann

2.1	Methodische Grundlagen	517
2.1.1	Abstrakte Datentypen	518
2.1.2	Grundlegende abstrakte Datentypen	521
2.1.3	Programmierparadigmen	525
2.2	Elemente von Programmiersprachen	526
2.2.1	Syntax, Semantik und Pragmatik	526
2.2.2	Syntaktische Eigenschaften Grundsymbole – Struktureller Aufbau von Programmen	527
2.2.3	Semantische Eigenschaften	529
2.3	Bindungen	529
2.3.1	Lebensdauer und Bindungen	529
2.3.2	Statische Bindung und Blockstruktur	531
2.4	Datentypen und Ausdrücke	532
2.4.1	Grundtypen	532
2.4.2	Zusammengesetzte Typen	533
2.4.3	Variablen in imperativen Sprachen, Zeigertypen	535
2.4.4	Vereinigungstypen und polymorphe Typen	536
2.4.5	Typäquivalenz	537
2.4.6	Ausdrücke	537
2.5	Sequentielle Ablaufsteuerung	539
2.5.1	Zuweisungen, bedingte Anweisungen und Schleifen Zuweisungen – Bedingte Anweisungen – Fallunterscheidung – Schleifen	540
2.5.2	Prozeduren und Funktionen Parameterübergabemechanismen	541
2.5.3	Ausnahmen	545
2.6	Modularität und Objektorientierung	546
2.6.1	Module	547
2.6.2	Klassen und Objekte	548
2.6.3	Vererbung	550
2.6.4	Generizität	552
2.7	Skriptsprachen Perl – Tcl/Tk – Python – PHP	554
2.8	Parallelität	558
2.9	Historische Entwicklung von Programmiersprachen	558
	Allgemeine Literatur	560
	Spezielle Literatur	560

A dark gray square containing the white text 'D2' in a serif font.

Eine Programmiersprache ist eine Notation für *Programme*, also für Beschreibungen von Berechnungen. Die Beschreibung soll für den menschlichen Leser verständlich und darüber hinaus auch effizient implementierbar sein. Die einfachsten Programmiersprachen sind *maschinennahe* Sprachen, die umkehrbar eindeutig der Befehlskodierung im Rechner entsprechen. Programmiersprachen, die sich mehr am menschlichen Verständnis ori-

entieren, heißen *höhere* Sprachen. Übersetzer transformieren Programme einer höheren Programmiersprache in maschinennahe Programme. Maschinennahe Programmiersprachen sind leicht zu implementieren, es ist aber aufwendig und fehleranfällig, in solchen Sprachen zu programmieren. Die heutigen höheren Programmiersprachen sind Abstraktionen der maschinennahen Programmierung, z.B. wenn arithmetische Ausdrücke in mathematischer Notation statt als Folge von Rechenoperationen angegeben werden. Andererseits geben die angebotenen Abstraktionen und Begriffe einen bestimmten Programmierstil und eine Programmiermethodik vor. Die vorhandenen Begriffe, mehr noch das Fehlen bestimmter Begriffe, beeinflussen die Denkweise der Programmierer und ihre Entwurfsmethoden. Höhere Programmiersprachen spiegeln daher das Verständnis wünschenswerter Programmier- und Entwurfsmethoden zum Zeitpunkt ihrer Entstehung wider.

Höhere Programmiersprachen klassifiziert man nach dem zugrundeliegenden Berechnungsmodell als imperativ, funktional und logisch. *Imperative* Sprachen beschreiben eine Berechnung als Folge von Zustandsübergängen einer Menge von Zustandsvariablen, die in erster Näherung den Speicher des benutzten Rechners repräsentieren. Alle maschinennahen Sprachen für v. Neumann-Rechner sind imperativ. Das Sprachelement zur Beschreibung eines einzelnen Zustandsübergangs heißt *Anweisung* oder *Befehl*. In *funktionalen* Sprachen beschreibt ein Programm die Berechnung des Ergebnisses einer mathematischen Abbildung, einer Funktion. Das Programm ist eine Menge von Funktionsdefinitionen und geschachtelten Funktionsaufrufen. Die Basis der funktionalen Programmiersprachen ist der λ -Kalkül. In *logischen* Programmiersprachen bestätigt man eine Formel der mathematischen Logik, indem man das Gegenteil, die Negation der Formel, zu widerlegen sucht. Die speziellen Werte der vorkommenden logischen Variablen, die zu dieser Widerlegung führen, bilden das Ergebnis der Berechnung. Alle drei Rechenmodelle sind zur Beschreibung einzelner Algorithmen geeignet. Zur Beschreibung reaktiver Systeme muß dem funktionalen oder logischen Rechenmodell zusätzlich ein Zustandsbegriff beigegeben werden. (Unter einem *reaktiven System* versteht man ein System, das in virtueller Zeit Eingabeströme von Daten auf Ausgabeströme abbildet, siehe [Manna 91].) Die Abgrenzung heutiger Programmiersprachen nach Rechenmodellen ist nur „im Prinzip“ möglich. Eine andere Klassifikation, die vor allem die imperativen Sprachen weiter unterteilt, orientiert sich an der Programmiermethodik und führt zu den Begriffen der *strukturierten, modularen* und *objektorientierten Programmiersprachen*.

Programmiersprachen sind Ingenieurprodukte. Bei ihrer Entwicklung und der Einsatzplanung muß man neben dem Rechenmodell, dem Anwendungsgebiet und der Programmiermethodik u. a. folgende Kriterien berücksichtigen:

- *Praktische Benutzbarkeit:* Die Sprache soll es Programmierern erlauben, die ihnen gewohnten Denk- und Ausdrucksweisen zu verwenden. Umständliche und fremdartige Ausdrucksweisen mit Rücksicht auf theoretische Rechenmodelle oder praktische Implementierungsanforderungen sind unerwünscht. Andererseits soll die Sprache die Benutzung von Spezialfunktionen, die der Rechner, das Betriebssystem, das Datenbanksystem usw. bereitstellen, nicht unnötig erschweren.
- *Verständlichkeit und Genauigkeit der Spezifikation:* Die Programmiersprache ist durch eine verbindliche Spezifikation vorgegeben: die Sprachbeschreibung, bei den gängigen Programmiersprachen fast immer eine internationale Norm. Je einfacher diese Beschreibung ist, desto weniger Mißverständnisse und Fehler gibt es beim Schreiben und beim Lesen von Programmen. Das Literaturverzeichnis nennt Standards und Normen der Programmiersprachen Algol 60 [Naur 63], Simula 67 [Dahl 68], Algol 68 [van Wijngaarden 75], Snobol 4 [Griswold 71], Pascal [Pascal 90], Fortran 77 [Fortran 77],

Pearl [Pearl 80], Smalltalk 80 [Goldberg 85], Modula-2 [Modula-2 96], COBOL [Cobol 85], C [C 99], C++ [C++ 98], Fortran 90 [Fortran 90], Fortran 95 [Fortran 95], SML [Milner 90], Eiffel [Meyer 92], Ada [Ada 95], Java [Gosling 96, Gosling 05], C# [C# 01, C# 05] und Fortran2003 [Fortran 04].

- *Leichte Implementierbarkeit:* Geringer Aufwand bei der Implementierung einer Programmiersprache verringert die Herstellungskosten. Er führt meist zu kürzeren Übersetzungszeiten und verringert damit die Kosten im Ediere-übersetze-teste-Zyklus. Vor allem verringert sich die Fehleranfälligkeit des Übersetzers und damit auch der übersetzten Programme. Oft führt leichte Implementierbarkeit auch zu effizienteren Zielprogrammen.
- *Portabilität der Programme:* Langlebige oder für einen großen Kundenkreis gedachte Programme müssen auf unterschiedlichen Rechnern laufen und mit unterschiedlichen Übersetzern übersetzt werden können. Manche dieser Rechner und Übersetzer können zum Zeitpunkt der Programmerstellung noch unbekannt sein. Dafür kommen nur international standardisierte Programmiersprachen in Betracht; Sprachelemente, die nur von einem einzelnen Übersetzer unterstützt oder die unterschiedlich interpretiert werden, und die Verwendung von Sprachen, auf die einzelne Hersteller ein Monopol besitzen, müssen vermieden werden. Insbesondere bei der Ein/Ausgabe, beim Aufruf von Betriebssystemfunktionen und bei der Verknüpfung von Programmteilen unterschiedlichen Ursprungs gibt es Schwierigkeiten, auf die bereits beim Entwurf der Programmiersprache Rücksicht genommen werden muß.
- *Kompatibilität:* Die Pflege großer Softwarebestände verbietet es, ältere Programmiersprachen in Zyklen von wenigen Jahren dem jeweils neuesten Erkenntnisstand der Programmiermethodik anzupassen. Es hemmt die Produktivität, wenn Programmierer laufend neue Sprachen erlernen oder mehrere Programmiersprachen nebeneinander benutzen sollen. Die „optimale Programmiersprache“ hängt nicht nur vom Einsatzzweck, sondern auch von der Historie, dem Umfeld und dem Ausbildungsstand ab.

Insbesondere die ersten drei Kriterien gehen nur teilweise Hand in Hand. Oft sind sie widersprüchlich. Eine Programmiersprache darf nicht isoliert nach nur einem Gesichtspunkt beurteilt werden.



2.1 Methodische Grundlagen

Programme in einer Programmiersprache beschreiben im einfachsten Fall *Algorithmen*, die eine Abbildung $f: E \rightarrow A$ von Eingaben E in Ausgaben A realisieren. Die Beschreibung soll sowohl für den menschlichen Leser als auch maschinell verständlich sein. Die Eingaben E und Ausgaben A sind *Daten*, d.h. Folgen von Bits, denen eine bestimmte Bedeutung beigelegt wird. Die Bitfolge heißt auch *Codierung* des Datums. In für Menschen lesbaren Programmen verwendet man dem jeweiligen Anwendungsgebiet angepaßte Codierungen, z.B. Zahlen im Dezimalsystem oder Texte in üblicher Schreibweise über einem vorgegebenen Alphabet, und überläßt die binäre Umcodierung der Implementierung der Programmiersprache. Die Bedeutung des Datums, soweit sie im Programm überhaupt erkennbar ist, ergibt sich dann oft aus der Schreibweise. Oder man kann sie explizit erklären, indem man dem Datum einen *Typ* zuordnet, der die mit dem Datum ausführbaren Operationen spezifiziert.

Bei einem Algorithmus setzt man voraus, daß seine Eingaben zu Beginn der Berechnung bekannt sind und am Ende die Ausgaben geliefert werden. Bei einem *reaktiven System* wird ein Strom von Eingaben, die nur schrittweise bekannt werden, zu einem Strom von

Ausgaben verarbeitet. Es kann mehrere Ein- und Ausgabeströme geben. Die Länge der Ströme ist unbeschränkt. Die Eingaben können von zuvor berechneten Ausgaben abhängen. Wieweit der Eingabestrom gelesen bzw. der Ausgabestrom geschrieben ist, wird im abstrakten Modell durch einen Eingabe- und einen Ausgabezeiger beschrieben.

Reaktive Systeme setzen sich aus Einzelalgorithmen zusammen. Wie diese Algorithmen realisiert sind, spielt im Gesamtzusammenhang keine oder eine untergeordnete Rolle; die einzelnen Algorithmen können als schwarzer Kasten behandelt werden. Dies ist das Prinzip der *prozeduralen Abstraktion*. Der Algorithmus heißt dann auch eine *Prozedur*. Eine solche Prozedur hat eine (prozedurale) *Schnittstelle*, nämlich die Spezifikation der zulässigen Eingaben, der *Argumente* der Prozedur und des *Resultats*, d.h. der Ausgaben. Innerhalb der Prozedur heißen die Argumente *Parameter*. In Programmiersprachen werden Argumente und Resultate heute nur nach Anzahl und Typ spezifiziert; weitere Angaben, etwa Konsistenzbedingungen, können in einigen neueren imperativen Sprachen wie etwa Eiffel als *Vor-* und *Nachbedingungen* spezifiziert werden. Jedoch fehlt zumeist die Verbindung mit einem automatischen Verifikationssystem, um diese Bedingungen vollständig zu prüfen.

Die Ausführung einer Prozedur wird im Gesamtprogramm durch einen *Prozeduraufruf* veranlaßt. Zur prozeduralen Abstraktion gehört an sich, daß dieser Aufruf *alle* Argumente angibt und die weitere Verwendung *aller* Resultate bestimmt. Eine Prozedur hat eine *Nebenwirkung*, wenn sie von sich aus auf Daten, die nicht in der Schnittstelle spezifiziert und nicht im Aufruf angegeben sind, zugreifen und diese womöglich verändern kann. Zu den Nebenwirkungen gehört auch das Weiterschalten des Eingabe- oder Ausgabezeigers, also das Lesen oder Schreiben von Daten auf externen Medien oder die Kommunikation mit anderen Programmen. Ferner sind alle potentiellen *Ausnahmen*, d.h. Fehler, die den Programmablauf beeinflussen, Nebenwirkungen, wenn ihr Einfluß über die Prozedur hinausreicht. Prozeduren ohne Nebenwirkungen, die nur ein einziges Resultat liefern, heißen *Funktionen*.

Der Begriff des Algorithmus und der Prozedur ist rekursiv: Prozeduren können selbst Prozeduraufrufe enthalten und unter anderem sich selbst (mit anderen Argumenten) aufrufen. Da auch einfache Operationen wie die Addition als Prozeduren angesehen werden können, besteht eine Programmausführung allgemein aus einer Folge von Prozeduraufrufen, deren Argumente entweder Eingaben oder Resultate vorangehender Aufrufe sind. Die nachfolgenden Programmierparadigmen zeigen das in unterschiedlicher Weise. Eine Programmiersprache beschreibt Algorithmen, indem sie schrittweise die Bedeutung der Elementaroperationen erklärt und dann angibt, wie man hieraus größere Prozeduren zusammensetzt.

2.1.1 Abstrakte Datentypen

Typen bilden heute in allen Programmiersprachen die Grundlage der Beschreibung von Daten. Ursprünglich wurde der Begriff Datentyp rein intuitiv verstanden. Arbeiten in der theoretischen Informatik (siehe z.B. [Wirsing 90]) legten die mathematische Grundlage für ein algebraisches Verständnis des Begriffs Datentyp, der hier dargelegt wird. Implementierungstechnische Gründe und die Rücksicht auf Programmierparadigmen und Denkgewohnheiten der Programmierer erlauben es aber nicht, die theoretisch saubere Formulierung unverändert in praktisch brauchbare Programmiersprachen zu übernehmen. Die Theorie abstrakter Datentypen verhilft vor allem zum systematischen Verständnis.

Ein *abstrakter Datentyp* (in diesem Abschnitt *Typ* genannt) besteht aus einer Menge von *Operatoren* $f: w_1 \times \dots \times w_n \rightarrow s$. Dabei sind w_1, \dots, w_n und s Typen. Der Operator f repräsentiert eine n -stellige Funktion, die Argumente t_1, \dots, t_n der Typen w_1, \dots, w_n auf einen

Typ s abbilden. $w_1 \times \dots \times w_n \rightarrow s$ heißt die *Stelligkeit* des Operators f . Ein nullstelliger Operator $c: \rightarrow s$ heißt *Konstante* des Typs s . Die Menge der Operatoren eines Typs T zusammen mit ihrer Stelligkeit heißt die *Signatur* von T .

Beispiel: Signaturen der Typen *BOOL* und *INT* (1)

Die beiden folgenden Signaturen beschreiben den Typ *BOOL* der Wahrheitswerte und den Typ *INT* der ganzen Zahlen. *true* und *false* sind Konstanten vom Typ *BOOL*; $0, 1, \dots$ Konstanten vom Typ *INT*. *INT* hat unendlich viele Konstanten. Die Operatoren von *BOOL* sind die üblichen logischen Operatoren und Vergleichsoperatoren, die von *INT* sind die üblichen arithmetischen Operatoren.

```
data type BOOL
  true, false:          → BOOL
  ¬:                    BOOL → BOOL
  ∧, ∨:                 BOOL × BOOL → BOOL
  =, ≠, <, >, ≥, ≤:    INT × INT → BOOL
end

data type INT
  ... -2, -1, 0, 1, 2, ...: → INT
  ⊕, ⊗, div, mod:        INT × INT → INT
end
```

Die *Terme* eines Typs T sind induktiv definiert: Eine Konstante vom Typ T ist ein Term; sind t_1, \dots, t_n Terme der Typen w_1, \dots, w_n und ist $f: w_1 \times \dots \times w_n \rightarrow s$ ein Operator, dann ist $f(t_1, \dots, t_n)$ ein Term vom Typ T . Sei X eine Menge (von Variablensymbolen), die disjunkt zu allen Typnamen und Operationssymbolen ist. Dann heißt $x \in X$ eine *Variable*. *Variablen* sind ebenfalls Terme eines Typs T . Terme ohne Variablen heißen *Grundterme*.

Dieser Variablenbegriff entspricht dem der mathematischen Logik und dem Begriff einer Unbestimmten in der Algebra. Er findet sich wieder in logischen und funktionalen Programmiersprachen. Er darf nicht verwechselt werden mit den Variablen imperativer Programmiersprachen, die ihren Wert in der Zeit ändern können.

Beispiel:

$x \wedge y$ ist ein Term vom Typ *BOOL* mit den Variablen x und y . $3 \leq 5 + 8$ ist ebenfalls ein Term vom Typ *BOOL*. Dieser Term ist ein Grundterm. $5 + 8$ ist ein Term vom Typ *INT*.

Intuitiv können Grundterme ausgewertet werden, z.B. $5 + 8$ zu 13. Dazu werden in abstrakten Datentypen *Gleichungen* $t_1 = t_2$ definiert, mit deren Hilfe man Terme auswertet oder vereinfacht. Eine Gleichung heißt oft ein *Axiom*. Dabei sind t_1 und t_2 Terme gleichen Typs und können Variablen enthalten. Ist letzteres der Fall, dann interpretiert man $t_1 = t_2$ als eine Menge von Gleichungen über Grundtermen, wobei jeweils alle Grundterme des Typs einer Variablen in die Gleichung eingesetzt werden. Die Gleichungen eines Datentyps definieren also eine Äquivalenzrelation. Meist verwendet man Gleichungen zur Definition der Bedeutung von Operatoren.

Beispiel: Der abstrakte Datentyp *BOOL* (2)

```
data type BOOL is
  true, false:          → BOOL
  ¬:                    BOOL → BOOL
  ∧, ∨:                 BOOL × BOOL → BOOL
equations x: BOOL ¬ true = false
  ¬ false = true
  true ∧ x = x
  false ∧ x = false
  false ∨ x = x
  true ∨ x = true
end
```

Der Datentyp *BOOL* definiert durch Gleichungen die Bedeutung der Operatoren \neg , \vee , und \wedge . Die Gleichung $true \wedge x = x$ steht für die unendliche Menge von Gleichungen $\{true \wedge t = t \in BOOL\}$. Durch die Gleichungen sind zwei Äquivalenzklassen definiert: in der einen Klasse sind alle Terme äquivalent zu *true*, in der anderen zu *false*.

Die *Konstruktoren* Γ eines abstrakten Datentyps *T* bilden die kleinste Menge von Operatoren mit Ergebnistyp *T*, so daß jede Äquivalenzklasse von *T* mindestens einen Grundterm *t* enthält, der nur Operatoren aus Γ benutzt. Die anderen Operatoren mit Ergebnistyp *T* in der Signatur heißen *Projektionen*.

Beispiel: Der abstrakte Datentyp *INTLIST* der Liste ganzer Zahlen (3)

```
data type INTLIST
  nil:          → INTLIST
  cons:  INT x INTLIST → INTLIST
  head:  INTLIST      → INT
  tail:  INTLIST      → INTLIST
  empty: INTLIST      → BOOL
equations a: INT, l: INTLIST
  head(cons(a,l)) = a
  tail(cons(a,l)) = l
  empty(nil)      = true
  empty(cons(a,l)) = false
end
```

Der Datentyp *BOOL* hat die Konstruktoren *true* und *false*. Der Datentyp *INT* hat die Konstruktoren $\dots, -2, -1, 0, 1, 2, \dots$. Der Datentyp *INTLIST* hat die Konstruktoren *nil* und *cons*. *nil* repräsentiert die leere Liste, *cons* fügt eine ganze Zahl in eine Liste ein. Damit hat *INTLIST* unendlich viele Äquivalenzklassen.

Der Datentyp *BOOLLIST* der Listen von Wahrheitswerten hat die gleiche Definition wie der Datentyp *INTLIST*. Man ersetzt überall *INT* durch *BOOL* und *INTLIST* durch *BOOLLIST*. Ebenso kann man Listen für andere Elementtypen *T* definieren. Um nicht jedesmal neue Definitionen einführen zu müssen, können abstrakte Datentypen mit *Typvariablen* parametrisiert werden. Ein *parametrisierter abstrakter Datentyp* $D(A_1, \dots, A_n)$ verwendet die Typvariablen A_1, \dots, A_n . Sie dürfen in der Definition der Signatur verwendet werden. Seien T_1, \dots, T_n abstrakte Datentypen. $D(T_1, \dots, T_n)$ ist ein mit T_1, \dots, T_n *instantiierter parametrisierter Datentyp*. $D(T_1, \dots, T_n)$ definiert einen abstrakten Datentyp, indem man in der Definition von *D* die Typvariablen A_i durch T_i , $i = 1, \dots, n$ ersetzt.

Beispiel: Der abstrakte Datentyp *LIST(T)* (4)

```
data type LIST(T)
  nil:          → LIST(T)
  cons:  T x LIST(T) → LIST(T)
  head:  LIST(T)   → T
  tail:  LIST(T)   → LIST(T)
  empty: LIST(T)   → BOOL
equations a:T, l:LIST(T)
  head(cons(a,l)) = a
  tail(cons(a,l)) = l
  empty(nil)      = true
  empty(cons(a,l)) = false
end
```

(4) zeigt den parametrisierten Datentyp *LIST(T)* der Listen, (3) den mit *INT* instantiierten parametrisierten Datentyp *LIST(INT)*.

Der Begriff der Konstruktoren verallgemeinert sich in naheliegender Weise auf parametrisierte Datentypen.

In funktionalen Sprachen kann man die Konstruktoren abstrakter Datentypen direkt festlegen und dann die Axiome definieren (siehe Kapitel D5). Allgemein ist ein *Typ in einer Programmiersprache* eine Implementierung eines abstrakten Datentyps. Wenn alle Konstruktoren eines Typs nullstellig sind, heißt der Typ *Grundtyp*, sonst *zusammengesetzter Datentyp*. Eine Sonderrolle spielen in imperativen Sprachen *Zeigertypen*.

Ein abstrakter Datentyp T ist ein *Untertyp* des Typs T' , wenn zwei Terme t, t' aus T genau dann äquivalent sind, wenn sie in T' äquivalent sind. Dieser Begriff des Untertyps ist die Grundlage der *polymorphen Typen* in objektorientierten Sprachen: Statt der Operationen des Obertyps T kann man die Operationen eines beliebigen Untertyps T' verwenden und erhält trotzdem ein Resultat in der gleichen Äquivalenzklasse, die sich auch mit den Operationen aus T ergeben hätte. Insbesondere kann T' eine Implementierung des abstrakten Datentyps T sein.

2.1.2 Grundlegende abstrakte Datentypen

Dieser Abschnitt stellt die Signaturen und Axiome wichtiger abstrakter Datentypen zusammen. Die Operationen sind als Konstruktoren und Projektionen klassifiziert. In einigen Fällen sind Hilfskonstruktoren notwendig, die nicht zur Signatur gehören, aber für die Formulierung der Gleichungen erforderlich sind. Die Formulierung der Gleichungen benutzt vielfach bedingte rekursive Aufrufe von Operationen.

Viele Kombinationen von Operationen, z.B. bei Listen *tail(nil)* oder *head(nil)*, sind nicht spezifiziert. In der praktischen Anwendung muß eine solche Kombination zu einer *Ausnahme* führen, die eine Fehlerbehandlung oder den Programmabbruch einleitet. In seltenen Fällen bedeutet Nicht-Spezifikation einer Kombination, daß es dem Programmierer freisteht, welches Resultat er liefert. Um Konsistenz und Vollständigkeit zu prüfen, hat sich in der Praxis eine tabellarische Erfassung der Definition abstrakter Datentypen bewährt [Parnas 94].

Der in (5) definierte Datentyp der natürlichen Zahlen ist abzählbar unendlich und daher wegen Ressourcenbeschränkungen ebenso wie der nachfolgende Typ *INT* nicht vollständig in maschinell ausführbaren Programmen realisierbar. Abstrakte Datentypen geben eine mathematische Idealisierung der Realität wieder. Unter Verifikationsgesichtspunkten, auf die hier nicht näher eingegangen wird, ist es Aufgabe des Programmierers, nicht der Programmiersprache oder ihrer Implementierung, diesen Unterschied zwischen Realität und Ideal zu berücksichtigen.

Beispiel: Der abstrakte Datentyp *NAT* der natürlichen Zahlen (5)

```

data type NAT
constructors
  0:    NAT
  succ: NAT → NAT
operations
  plus: NAT × NAT → NAT
  times: NAT × NAT → NAT
equations a,b: NAT
  plus(0,a)      = a
  plus(succ(a),b) = succ(plus(a,b))
  times(0,a)     = 0
  times(succ(a),b) = plus(times(a,b),b)
end

```

Beispiel: Der abstrakte Datentyp *INT* der ganzen Zahlen (6)

```

data type INT
constructors
  ... - 2, - 1, 0, 1, 2, ... :→ INT
operations
  ⊕, ⊗, div, mod: INT × INT → INT
  =, ≠, <, >, ≥, ≤ :INT × INT → BOOL
equations a,b:  INT
  0⊕a = a
  1⊗1 = 2
  ...
end

```

In gleicher Weise könnte man abstrakte Datentypen *REAL* und *COMPLEX* als Obertypen von *INT* spezifizieren. Wegen der Ressourcenbeschränkungen läßt sich jedoch die reelle Arithmetik immer nur mit beschränkter Genauigkeit durchführen. Dabei geht die Ober-Untertyp-Beziehung verloren.

Die nachfolgenden abstrakten Datentypen sind sämtlich parametrisiert; der Typparameter *T* beschreibt den Elementtyp dieser *Behältertypen*.

Der abstrakte Datentyp *Variable(T)* gibt *Variablen* im Sinne imperativer Sprachen wieder. Ihr Wert vom Typ *T*, einem weiteren abstrakten Datentyp, kann sich durch Schreiboperationen ändern. Variablen in diesem Sinne sind nur dann unterscheidbar, wenn sie unterschiedliche Werte haben. In der praktischen Umsetzung besitzt eine Variable in einer imperativen Sprache zusätzlich eine *Referenz* oder einen *Zeiger*. Er bestimmt die Identität der Variablen, so daß auch Variablen gleichen Werts unterschieden werden können. Dabei gibt es zwei Fälle:

- *Wertsemantik* oder *Kopiersemantik*: Die Referenz identifiziert umkehrbar eindeutig ein Exemplar des abstrakten Datentyps *Variable*. Sein Wert kann sich durch Schreiboperationen unabhängig von anderen Variablen ändern.
- *Referenzsemantik*: Mehrere Referenzen können auf dasselbe Exemplar zeigen. Eine Schreiboperation ändert gleichzeitig den Wert aller, durch ihre Referenzen unterschiedenen Variablen, die auch zuvor den gleichen Wert besaßen.

Beispiel: Der abstrakte Datentyp *VARIABLE(T)* (7)

```

data type VARIABLE(T)
constructors
  create:                → VARIABLE(T)
  write: T × VARIABLE(T) → VARIABLE(T)
operations
  read: VARIABLE(T)      → T
equations x,y:T, v:VARIABLE(T)
  write(x,write(y,v)) = write(x,v)
  read(write(x,v))   = x
end

```

Keller (8), Schlangen (9) und Sequenzen (10) sind die wichtigsten linearen Behältertypen.

Beispiel: Der abstrakte Datentyp *STACK(T)* der Keller (8)

```

data type STACK(T)
constructors
  create:                → STACK(T)
  push: T × STACK(T)    → STACK(T)
operations
  pop:  STACK(T) → STACK(T)
  top:  STACK(T) → T
  empty: STACK(T) → BOOL

```

```

equations a:T, s:STACK(T)
  empty(create) = true
  empty(push(a,s)) = false
  top(push(a,s)) = a
  pop(push(a,s)) = s
end

```

Beispiel: Der abstrakte Datentyp *QUEUE(T)* der Schlangen (9)

```

data type Queue(T)
constructors
  create: → QUEUE(T)
  enq: T × QUEUE(T) → QUEUE(T)
operations
  deq: QUEUE(T) → QUEUE(T)
  front: QUEUE(T) → T
  empty: QUEUE(T) → BOOL
equations a,b:T, s:QUEUE(T)
  empty(create) = true
  empty(enq(a,s)) = false
  front(enq(a,create)) = a
  front(enq(a,enq(b,s))) = front(enq(b,s))
  deq(enq(a,create)) = create
  deq(enq(a,enq(b,s))) = enq(a,deq(enq(b,s)))
end

```

SEQ(T) spezifiziert die Grundoperationen auf sequentiellen Dateien. Mit *read* und *write* werden Sequenzen gelesen oder geschrieben. Sequenzen haben implizit eine Position, die durch die Funktion *skip* definiert ist. *show* gibt das Element an dieser Position zurück. *reset* setzt diese Position auf das erste Element der Sequenz. *eof* gibt an, ob die Position die letzte in der Sequenz ist.

Beispiel: Der abstrakte Datentyp *SEQ(T)* der Sequenzen oder Dateien (10)

```

data type SEQ(T)
constructors
  createfile: → SEQ(T)
  write: SEQ(T) × T → SEQ(T)
  read: SEQ(T) × T → SEQ(T)
operations
  reset: SEQ(T) → SEQ(T)
  skip: SEQ(T) → SEQ(T)
  show: SEQ(T) → T
  eof: SEQ(T) → BOOL
  empty: SEQ(T) → BOOL
equations a,b:T; f:SEQ(T)
  reset(createfile) = createfile
  reset(write(f,a)) = read(reset(f),a)
  reset(read(f,a)) = read(reset(f),a)
  skip(read(createfile,a)) = write(createfile,a)
  skip(read(write(f,a),b)) = write(write(f,a),b)
  skip(read(read(f,a),b)) = read(skip(read(f,a),b))
  show(read(createfile,a)) = a
  show(read(write(f,a),b)) = b
  show(read(read(f,a),b)) = show(read(f,a))
  eof(createfile) = true
  eof(write(f,a)) = true
  eof(read(f,a)) = false
  empty(createfile) = true
  empty(write(f,a)) = false

```



```

empty(read(f,a))      = false
write(read(f,a),b)   = write(f,b)
end

```

(11) zeigt den abstrakten Datentyp der Binärbäume. Mit *left* bzw. *right* greift man auf den linken bzw. rechten Unterbaum zu, durch *node* werden Teilbäume mit einer neuen Wurzel zusammengefügt. Binärbäume sind nichtlineare Behältertypen.

Beispiel: Der abstrakte Datentyp *BINTREE(T)* der Binärbäume (11)

```

data type BINTREE(T)
constructors
  createtree:          → BINTREE(T)
  node: BINTREE(T) × T × BINTREE(T) → BINTREE(T)
operations
  left:  BINTREE(T) → BINTREE(T)
  right: BINTREE(T) → BINTREE(T)
  value: BINTREE(T) → T
  empty: BINTREE(T) → BOOL
equations a:T; t1,t2: BINTREE(T)
  left(node(t1,a,t2)) = t1
  right(node(t1,a,t2)) = t2
  value(node(t1,a,t2)) = a
  empty(createtree) = true
  empty(node(t1,a,t2)) = false
end

```

Mengen sind ebenfalls nichtlineare Behältertypen. In ihnen ist keine Ordnung der Elemente definiert; im Gegensatz zu Kellern, Listen, Sequenzen und Schlangen kann man nicht vom ersten oder letzten Element sprechen. In der Definition des abstrakten Datentyps (12) kann man dies an der Definition von *elem* erkennen. Die Axiome sind teilweise durch bedingte Gleichungen definiert.

Beispiel: Der Datentyp *SET(T)* der Mengen über *T* (12)

```

data type SET(T)
constructors
  emptyset:          → SET(T)
  elem: T × SET(T) → SET(T)
operations
  single:      T          → SET(T)
  insert:     SET(T) × T  → SET(T)
  member:     T × SET(T) → BOOL
  delete:     SET(T) × T  → SET(T)
  union:      SET(T) × SET(T) → SET(T)
  intersect:  SET(T) × SET(T) → SET(T)
  difference: SET(T) × SET(T) → SET(T)
  is_subset:  SET(T) × SET(T) → BOOL
  is_equal:   SET(T) × SET(T) → BOOL
  is_empty:   SET(T)       → BOOL
equations s,s':SET(T); x,x':T
  single(x)          = elem(x,emptyset)
  insert(s,x)        = union(single(x),s)
  member(x,emptyset) = false
  member(x,elem(x',s)) = if x=x' then true else member(x,s)
  delete(s,x)        = difference(s,single(x))
  union(emptyset,s)   = s
  union(elem(x,s),s') = if member(x,s') then union(s,s') else elem(x,union(s,s'))
  intersect(emptyset,s) = emptyset
  intersect(elem(x,s),s') = if member(x,s') then elem(x,intersect(s,s')) else intersect(s,s')
  difference(emptyset,s) = emptyset

```

```

difference(elem(x,s),s') = if member(x,s') then difference(s,s') else elem(x,difference(s,s'))
is_subset(s,s')          = is_empty(difference(s,s'))
is_equal(s,s')           = is_subset(s,s') ∧ is_subset(s',s)
is_empty(emptyset)      = true
is_empty(elem(x,s))     = false
end

```

2.1.3 Programmierparadigmen

Funktionale Sprachen unterstützen direkt die Definition und Programmierung abstrakter Datentypen. Ihre Grundlage ist der λ -Kalkül (siehe Kapitel D5). Logische Programmiersprachen kennen ebenfalls Terme als Typen. Die Grundlage ist Horn-Logik und Resolution (siehe Kapitel D6).

Beim *imperativen Programmierparadigma* betrachtet man, wie bereits erwähnt, eine Berechnung als Folge von Zustandsübergängen von Variablen im Sinne von (7). Solche Zustandsübergänge können sehr komplex sein und entsprechen dann unter Umständen umfangreichen Prozeduren (*prozedurale Abstraktion*). Sie können unter Bedingungen stehen oder, mit jeweils anderen Variablenwerten, in Schleifen ausgeführt werden. Die Sprachelemente *Prozedur* und *Prozeduraufruf*, *bedingte Anweisung* und *Schleife* sind aus theoretischer Sicht ausreichend, um alle Algorithmen zu formulieren. Praktisch ergeben sich die gleichen Elemente, wenn man Programmentwurf durch schrittweise Verfeinerung mit Verwendung von Vor- und Nachbedingungen zur Verifikation nach Hoare oder Dijkstra betreibt [Hoare 69, Dijkstra 76]. Man spricht vom Paradigma des *strukturierten Programmierens*, wenn für die Ablaufsteuerung nur diese Elemente eingesetzt werden. (Bedingte) *Sprünge*, also willkürliche Bestimmung der nächsten auszuführenden Anweisung, sind in diesem Paradigma nicht erlaubt. Eine Sonderrolle spielen *Ausnahmen*, d.h. die Formulierung von Alternativen der Programmausführung, um nach einem Fehler, z.B. Division durch Null, versuchter Zugriff auf eine nicht vorhandene Datei usw., wieder einen Zustand zu erreichen, in dem die Programmausführung ordnungsgemäß fortgesetzt und zu Ende geführt werden kann.

Prozedurale Abstraktion unterstellt die Verwendung des *Geheimnisprinzips*, in diesem Fall, um lokale Größen einer Prozedur nicht nach außen sichtbar werden zu lassen. Die einzelnen benannten Variablen haben einen *Gültigkeitsbereich*; nur in diesem Bereich bedeutet die Benennung die gewünschte Variable. Auf Gültigkeitsbereiche wird in Abschnitt 2.2.3 genauer eingegangen.

Beim Zusammensetzen größerer Programme reichen Prozeduren nicht als Strukturierungshilfe aus. Im Paradigma des *modularen Programmierens* besteht ein Programm aus Modulen A, B, C, \dots , die jeweils über eine genau festgelegte Schnittstelle, bestehend aus Typdefinitionen, Prozeduren und lokalen Variablen, verfügen. Nur die in dieser Schnittstelle angegebenen Größen p können auch außerhalb des Moduls mit Namen angesprochen werden. Dabei wird der Name p im allgemeinen mit dem Namen A des Moduls, zu dem p gehört, *qualifiziert*: $A.p$. Alle anderen Größen im Modul unterliegen dem Geheimnisprinzip und sind von außen nicht sichtbar. Dies kann insbesondere die Einzelheiten der Implementierung von Prozeduren und Typen an der Schnittstelle betreffen. In Programmiersprachen wie Modula-2 und Ada bestehen dazu die Module bzw. Pakete aus einem (Schnittstellen-)Definitions- und einem Implementierungsteil.

Modulares Programmieren erlaubt die Realisierung abstrakter Datentypen, indem an der Modulschnittstelle ein (oder mehrere) Typen T und Operationen mit Objekten dieser Typen zugänglich gemacht werden. Die Einzelheiten der Repräsentation der Objekte und damit der Definition von T unterliegen dem Geheimnisprinzip, um Repräsentationswech-

sel zu ermöglichen. Daher werden in modularen (und objektorientierten) Programmiersprachen nur die Namen T , T' zweier Typen verglichen, um Gleichheit festzustellen (*Namensgleichheit von Typen*). Die *strukturelle Gleichheit* wird nur für Felder benötigt.

Im *objektorientierten Programmieren* heißen die Module *Klassen* und definieren Schablonen zur Bildung oder Instantiierung von Objekten. Eine Klasse kann die Schnittstelle, die Typrepräsentation und die Operationen eines abstrakten Datentyps definieren. Die Begriffe Typ und Klasse fallen in den meisten objektorientierten Sprachen zusammen. Weitere Einzelheiten dazu finden sich in Abschnitt 2.6 und Kapitel D4.

2.2 Elemente von Programmiersprachen

Die Bedeutung eines Programms ergibt sich durch Zusammensetzen der Bedeutungen der darin vorkommenden Sprachelemente. Die Sprachdefinition gibt deren Interpretation und ihre Verknüpfungen an. Die Sprachelemente sind ausführbare Operationen; die Ausführung ändert den Stand der Berechnung. Man kann die Menge der möglichen Operationen als Befehlsvorrat einer (abstrakten) Maschine auffassen, deren Maschinensprache die beschriebene Programmiersprache ist.

Programmierer lesen die Sprachdefinition als Benutzerhandbuch zur Beantwortung der Fragen: Was bedeuten die Sprachelemente? Wozu können sie *sinnvoll* benutzt werden? Wie können sie *sinnvoll* kombiniert werden? Der Übersetzerbauer interpretiert die Sprachdefinition hingegen als Spezifikation, die er zu implementieren hat. Er fragt sich, was alles erlaubt ist, selbst dann, wenn eine Kombination oder Benutzung von Sprachelementen nicht sinnvoll erscheint.

2.2.1 Syntax, Semantik und Pragmatik

Programme sind heute meist Texte, also Zeichenfolgen. Die *Programmiersprache* definiert, welche Texte zulässige Programme sind und welche nicht. Die *Syntax einer Programmiersprache* ist eine Beschreibung einer Obermenge der zulässigen Programmtexte. Gewöhnlich definiert man die Syntax einer Programmiersprache mit Hilfe einer kontextfreien Grammatik in *erweiterter Backus-Naur-Form* (kurz: *EBNF*). Diese Methode wurde mit der *Backus-Naur-Form* (kurz: *BNF*) zur Beschreibung der Syntax der Sprache Algol 60 erstmals eingeführt. (14) zeigt eine solche Definition. Die linken Seiten der Produktionen führen Begriffe ein, die man als syntaktische *Elemente der Programmiersprache* bezeichnet. Sie prägen dem Programmtext eine *syntaktische Struktur* auf.

Die *Semantik einer Programmiersprache* beschreibt die Bedeutung der syntaktischen Sprachelemente, aus denen sich Programme zusammensetzen, unter Berücksichtigung des jeweiligen *Kontexts*. Dazu gehört auch die *statische Semantik*, d.h. Eigenschaften und Bedingungen, die ohne Ausführung eines Programms bestimmt werden können. Sie schränkt die Menge der zulässigen Programme ein.

Beispiel:

In vielen Programmiersprachen müssen Bezeichner vereinbart sein, wenn sie benutzt werden. Die Namensanalyse, d.h. die Bestimmung der zugehörigen Vereinbarung, gehört ebenso zur statischen Semantik wie die Bestimmung des Typs eines Objekts oder eines Rechenergebnisses in einer typisierten Sprache.

Die *dynamische Semantik* definiert die Ausführung der Sprachelemente. In der Terminologie der Sprachphilosophen gehört die statische Semantik zur Syntax. Bei Programmiersprachen sind die Abgrenzungen oft willkürlich.

Die *Pragmatik einer Programmiersprache* setzt Sprachelemente zu Konzepten außerhalb der Programmiersprache in Beziehung. Das Additionssymbol „+“ und der Begriff „Gleitpunktzahl“ sind gewöhnlich syntaktische Begriffe. Man könnte die Addition zweier Gleitpunktzahlen auf umständliche Weise zur Semantik einer Programmiersprache zählen. Gewöhnlich verweist man jedoch auf die Ausführung der Gleitpunktaddition durch den Rechner, einer Einheit außerhalb der Begriffswelt der Sprache. Auch Datentypen und andere theoretisch erklärbare Begriffe werden oft pragmatisch definiert. So beschreibt etwa Pascal den booleschen Typ *BOOL* so:

[Boolean] Values are the truth values denoted by the identifiers true and false (Pascal Report, Section 6.1.2 [Jensen 74])

2.2.2 Syntaktische Eigenschaften

Die kleinsten bedeutungstragenden syntaktischen Einheiten einer Programmiersprache heißen *Grundsymbole* (in Kapitel A3 *Terminalsymbole* genannt). Die Syntax zerfällt in zwei Teile: die Beschreibung des Aufbaus von Grundsymbolen aus einzelnen Zeichen, für die meist eine reguläre Grammatik oder ein regulärer Ausdruck genügt, und die kontextfreie Produktionsmenge für die syntaktische Struktur eines Programmtextes über dem Alphabet der Grundsymbole.

Grundsymbole

Grundsymbole sind *Schlüsselwörter*, *Bezeichner*, *Konstanten*, *Kommentare* und *Spezialsymbole*. Schlüsselwörter, z.B. *begin*, *end*, *while*, *if*, *then*, *else*, haben eine feste unabänderliche Bedeutung. Bezeichner sind frei wählbare Wörter. Konstanten(-bezeichner) sind Zahlen, Zeichen oder Zeichenreihen. Spezialsymbole (wie „:“, „:=“, „::“, „#“, „\$“) sind spezielle Zeichen oder Zeichenkombinationen, denen die Semantik eine Bedeutung zuordnet. (13) zeigt einige typische Grundsymbole, die durch reguläre Ausdrücke definiert sind.

Gebräuchliche Grundsymbole (13)

```
bez ::= Buchstabe (Buchstabe + Ziffer)*
zeichenreihe ::= "" 'Zeichen' ""
spezial ::= '=' + ':' + '#' + '$' + ':' + ';' + '<' + '+' + '*' + '/'
```

Meist unterscheiden sich Schlüsselwörter und Bezeichner in der Schreibweise nicht. Schlüsselwörter sind dann Bezeichner mit vordefinierter Bedeutung. In manchen Programmiersprachen bestimmt sogar der Kontext, ob es sich bei einem Bezeichner um ein Schlüsselwort handelt oder nicht (z.B. COBOL, PL/1). Algol 60 setzte ursprünglich eine unterschiedliche Schreibweise für Schlüsselwörter und Bezeichner voraus. Fortran benutzt eine gemischte Schreibweise: Schlüsselwörter wie *DO* oder *COMMON* schreibt man wie Bezeichner; Schlüsselwörter wie *EQ.* oder *LT.*, die in Ausdrücken vorkommen können, werden durch einen vor- und nachgestellten Punkt unterschieden.

Kommentare sind spezielle Grundsymbole. Sie sind für die maschinelle Interpretation eines Programms irrelevant. Der Programmierer kann damit Zusatzinformationen angeben, die nur den menschlichen Leser interessieren. In manchen Sprachen gibt es mehrere Formen von Kommentaren, darunter solche, die dem Übersetzer Implementierungshinweise geben, die dieser berücksichtigen kann, aber nicht muß. In Ada heißen solche Spezialkommentare nach ihrem einleitenden Schlüsselwort *Pragmas*.

Beispiel: Ausschnitt aus der Syntax einer Programmiersprache (14)

```
Klassendef ::= ['abstract'] 'class' Klassenkopf 'is' Klassenrumpf 'end' ';'
Klassenkopf ::= bez ['(' (bez ['<' Klasse] || ';' ) ')']
Klasse ::= bez ['(' (Klasse || ';' ) ')']
```

Klassenrumpf	::= (Vererbung ';')*(Attribut ';')*(Methode ';')*
Vererbung	::= 'subtype of' Klasse 'include' Klasse
Attribut	::= bez ':' Typ
Typ	::= (Klasse PolymorpheKlasse)
PolymorpheKlasse	::= '\$' Klasse
Methode	::= bez ['(' (par ';') * ')'] [':' Typ] 'is' Methodenrumpf 'end'
par	::= bez ':' Typ
Methodenrumpf	::= (Attribut ';') * (Anweisung ';') *
Anweisung	::= Zuweisung Methodenaufruf bedAnw Schleife Eingabe Ausgabe
Zuweisung	::= ObjDes ':=' Ausdruck
ObjDes	::= [(#' Klasse Zeichenkette) Methodenaufruf] (':' ObMethodenaufruf) *
ObMethodenaufruf	::= bez ['(' (ObjDes ';') ')']
Methodenaufruf	::= [Klasse ':'] ObMethodenaufruf
Ausdruck	::= ObjDes Vergleich Konstante unop Ausdruck Ausdruck binop Ausdruck '(' Ausdruck ')'
Vergleich	::= Ausdruck relop Ausdruck
relop	::= '<' '=' '>' '<>' '<=' '>='
unop	::= 'not' '-' '+'
binop	::= 'and' 'or' '+' '**' 'div' 'mod' '/'
bedAnw	::= 'if' Ausdruck then (Anweisung ';') * ('elsif' Ausdruck 'then' Anweisung ';') * ['else' (Anweisung ';') *] 'end'
Schleife	::= 'while' Ausdruck 'do' (Anweisung ';') * 'end'

Struktureller Aufbau von Programmen

(14) zeigt einen Ausschnitt der Syntax einer Programmiersprache, definiert durch eine kontextfreie Grammatik in *EBNF*. Da die Semantik von Programmen in Termen der syntaktischen Elemente definiert ist, ist es günstig, wenn die syntaktische Struktur eines Programms eindeutig festgestellt und in einem *Strukturbaum* wie in Bild 1 festgehalten werden kann. Die Definitionen der meisten Programmiersprachen erfüllen diese Forderung nicht; die kontextfreie Grammatik ist mehrdeutig. Ein Übersetzer muß diese Forderung nachträglich erfüllen, indem er die Grammatik entsprechend transformiert oder indem er Regeln der statischen Semantik zur Auflösung von Mehrdeutigkeiten benutzt.

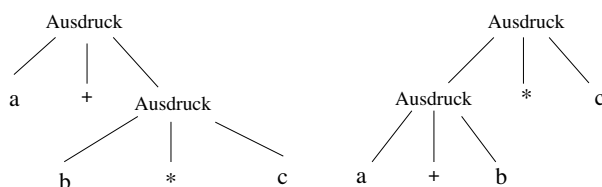


Bild 1 Syntaxbäume für $a + b * c$

Beispiel:

Die Definition eines Ausdrucks durch die *EBNF* in (14) regelt nicht eindeutig, welcher der beiden Strukturbäume in Bild 1 dem Ausdruck $a + b * c$ zugeordnet ist. Die Grammatik läßt sich jedoch so transformieren, daß unter Berücksichtigung der üblichen *Vorrangregeln* der linke Strukturbaum in Bild 1 gewählt wird.

Beispiel:

In Fortran und Ada ist syntaktisch nicht feststellbar, ob der Ausdruck $a(i,j)$ eine indizierte Variable oder einen Funktionsaufruf mit 2 Argumenten darstellt. Dies hängt davon ab, ob a nach den Regeln der statischen Semantik eine indizierbare Variable benennt oder nicht.

2.2.3 Semantische Eigenschaften

Es gibt drei verschiedene Formen der Definition der Semantik von Programmiersprachen: operationale, denotationale und axiomatische Semantik. Die *denotationale Semantik* begreift ein Programm als Abbildung seiner Eingaben in seine Ausgaben; diese Abbildung wird induktiv über die Sprachelemente definiert. Die *operationale Semantik* beschreibt die Wirkung eines Programms als schrittweise Zustandsänderung der abstrakten Maschine, die durch die Programmiersprache gegeben ist. Die *axiomatische Semantik*, z.B. in [Hoare 73], geht davon aus, daß die Ein- und Ausgaben bestimmte Vor- und Nachbedingungen erfüllen. Den Sprachelementen entsprechen Axiome über die Bedingungen, die der Zustand vor oder nach Ausführung des Elements erfüllt; mit ihrer Hilfe wird induktiv nachweisbar, daß die Vorbedingung des Gesamtprogramms durch die Programmausführung in die Nachbedingung überführt wird. (Diese Formulierung unterstellt Vorwärtsanalyse unter Benutzung Hoarescher Logik. Mit dem Kalkül der schwächsten Vorbedingungen schließt man von den Nachbedingungen rückwärts.)

Die axiomatische Semantik eignet sich für Programmieranleitungen; da sie aber nur sinnvolle Kombinationen von Sprachelementen erfaßt, gibt sie dem Übersetzerbauer keine ausreichende Auskunft in Zweifelsfällen. Funktionale Sprachen werden erfolgreich mit denotationaler Semantik definiert. Die Sprachdefinitionen aller gängigen imperativen Programmiersprachen verwenden operationale Semantik, indem sie einen Interpretierer beschreiben, der die Zustandsänderungen schrittweise ausführt. Für manche Sprachen, z.B. *Lisp 1.5* [McCarthy 85] und *PL/1*, gibt es solche Interpretierer entweder als Programm oder als formales System. In der Praxis gibt man die denotationale bzw. operationale Semantik jedoch zumeist informell durch Sätze in natürlicher Sprache wieder. Im folgenden wird beispielhaft die operationale Semantik behandelt.



2.3 Bindungen

Alle Methoden zur Semantikdefinition handeln von den Datenobjekten während der Programmausführung und den für sie definierten Operationen. *Objekte* sind konstante Werte, Variablen oder Resultate von Operationen (Zwischenergebnisse). Die Menge der Variablen heißt *Zustandsraum eines Programms*. Zusammen mit ihren Werten bilden sie den *Programmzustand*. Operationen werden auf einem Zustand durchgeführt und ändern ihn. Eine solche Zustandsänderung heißt *Nebenwirkung (side effect)*.

2.3.1 Lebensdauer und Bindungen

Die Menge der möglichen Variablen ist abzählbar unendlich. Theoretisch kann man von dieser unendlichen Menge als Zustandsraum ausgehen und die Variablen und Objekte als immerwährend existierend voraussetzen. Praktisch genügt es, sich auf die Objekte zu beschränken, die im Programm tatsächlich angesprochen werden können. Jedes solche Objekt hat eine bestimmte *Lebensdauer (extent)*, nämlich den Teil der Programmausführung, während dessen es benutzt werden kann. Man unterscheidet:

- *Persistente Objekte*: Objekte, die auch vor Beginn oder nach dem Ende der Ausführung eines Programmlaufs existieren, z. B., weil sie zu einer Datenbank gehören.
- *Objekte beschränkter Lebensdauer*: Diese Objekte sind im Programmtext durch Bezeichner benannt. Sie werden bei Einführung des Bezeichners gebildet und können gelöscht werden, wenn der zugeordnete Bezeichner ungültig wird. Die explizite Einführung eines Bezeichners und damit eines Objekts heißt *Vereinbarung*.
- *Objekte unbeschränkter Lebensdauer*: Diese Objekte sind in den heutigen Programmiersprachen zumeist *anonym*; sie werden durch eine explizite Operation *neue Variable (new, allocate)* gebildet, welche die *Referenz* auf die Variable als Ergebnis liefert. Die Variable lebt mindestens so lange, wie diese Referenz als Wert einer anderen lebenden Variablen vorkommt. Objekte unbeschränkter Lebensdauer könnten auch durch eine Vereinbarung eingeführt sein.

Im Programmtext finden sich keine Objekte, sondern Bezeichner und Operatoren (Operationssymbole) zur Benennung von Objekten und Operationen. Die Zuordnungen Bezeichner – Objekt und Operator – Operation heißen *Bindungen*. Die Bindung ist nur in sehr einfachen maschinennahen Sprachen umkehrbar eindeutig. In allen höheren Programmiersprachen kann ein Bezeichner oder ein Operator verschiedene Objekte oder Operationen benennen, abhängig vom Kontext, in dem der Bezeichner oder der Operator vorkommt. Man unterscheidet folgende Fälle:

- *Statische Bindung*: Sie wird durch eine Vereinbarung im Programm festgelegt. Es gibt einen Programmteil, gegeben durch eine syntaktische Struktur, den *Gültigkeitsbereich (scope)* der Bindung, in dem die Zuordnung gültig ist. Gültigkeitsbereiche können geschachtelt sein. Die Bindungsregeln entsprechen den Regelungen über freie und gebundene Variablen in logischen Formeln oder (geschachtelten) Integralen in der Analysis. Da die Bindung nur die Kenntnis der syntaktischen Struktur voraussetzt, gehören die Bindungsregeln zur statischen Semantik. Der Bezeichner in der Vereinbarung heißt *Bezeichnerdefinition (defining occurrence of an identifier)*; alle anderen Vorkommen des Bezeichners heißen *Anwendungen (des vereinbarten Bezeichners) (applied occurrences)*. Im Übersetzerbau ist die Auswertung der statischen Bindungen als *Namenanalyse* bekannt.
- *Dynamische oder polymorphe Bindung*: Die Bindung von Operatoren, die *Operator-identifizierung*, ist oft vom Typ der Operanden abhängig: $a + b$ bezeichnet eine ganzzahlige oder eine Gleitpunktaddition, abhängig davon, ob a, b ganze Zahlen oder Gleitpunktzahlen benennen. Kann a, b durch die statische Semantik bereits ein Typ zugeordnet werden, so liegt statische Bindung vor; bei dynamischer Bindung geschieht dies erst während der Ausführung.
- *Dynamische Bindung bei Vereinbarung*: Insbesondere bei rekursiven Programmstrukturen werden Bezeichner oft in zwei Schritten gebunden: Durch statische Bindung wird den Anwendungen eine Bezeichnerdefinition zugeordnet. Wird die Vereinbarung mehrfach ausgeführt und dadurch mehrere Objekte gebildet, so wird während der Ausführung durch dynamische Bindung jeweils das in der Schachtelung jüngste Objekt zugeordnet.
- *Dynamische Bindung bei Anwendung*: Man findet sie in APL und bei interpretativer Ausführung von Lisp: Ein Bezeichner benennt während der Ausführung das jeweils jüngste unter diesem Bezeichner eingeführte Objekt. Der syntaktische Kontext der Anwendung des Bezeichners wird im Unterschied zum vorigen Fall nicht berücksichtigt. Diese Regelung hat sich, insbesondere bei Programmänderungen, als äußerst fehleranfällig erwiesen.

- *Bindung bei Programmstart*: Sie stellt einen Zwischenschritt zwischen statischer und dynamischer Bindung dar. In COBOL werden bei Programmstart die vom Betriebssystem verwalteten Dateien den in der *environment section* des Programms vereinbarten Dateibezeichnern zugeordnet. Danach ist diese Bindung unveränderlich.

Vereinbarungen spielen also eine Doppelrolle: Sie führen zur Bildung von Objekten und sind damit ausführbare Operationen. Außerdem spezifizieren sie eine Bindung Bezeichner – benanntes Objekt, die oft statisch ausgewertet werden kann.

2.3.2 Statische Bindung und Blockstruktur

Die Zusammenfassung lokaler Vereinbarung und eines Anweisungsteils heißt *Block*. Der Vereinbarungsteil eines Blocks kann ebenso wie der Anweisungsteil (Unter-)Blöcke enthalten. (15) zeigt einen Block *B* mit Unterblock *C*. Die Blockstruktur ist in allen Programmiersprachen Grundlage der statischen Gültigkeitsbereichsregeln und kann durch geschachtelte Rechtecke mit den in ihnen enthaltenen Vereinbarungen wie in Bild 2 dargestellt werden. Ein Bezeichner *x* heißt *sichtbar* an einer Programmstelle, wenn er in einem der die Programmstelle umgebenden Blöcke vereinbart ist. Ist *x* im innersten Block vereinbart, dann heißt *x* *lokaler Bezeichner*, sonst heißt *x* *global*. In (15) sind im Anweisungsteils des Blocks *B* die Bezeichner *B*, *x*, *y* und *C* sichtbar. Im Anweisungsteil des Blocks *C* sind die Bezeichner *B*, *C*, *x*, *y* und *z* sichtbar. Man beachte, daß die beiden *y* zwei verschiedene Objekte bezeichnen: das *y* im Anweisungsteil von *C* ist der Vereinbarung in *C* zugeordnet, das im Anweisungsteil von *B* der Vereinbarung in *B*.

Beispiel: Blöcke in Ada

(15)

```

B: declare  x: FLOAT;
           y: BOOL;
  C: declare y,z: FLOAT;  -- y verdeckt das y von B
  begin
    z := x; y := z; x := z + y;  -- y ist das in C vereinbarte y, x ist in B vereinbart
  end C
begin
  y := (x=3);  -- y ist das in B vereinbarte y
end B

```

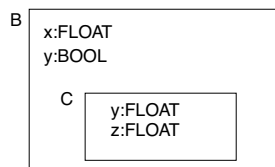


Bild 2 Graphische Darstellung der Blöcke in (15)

Die *Umgebung* einer Programmstelle ist die Menge aller an ihr sichtbaren Bezeichner samt deren Vereinbarungen. Vereinbart ein innerer Block ebenfalls einen Bezeichner *x*, dann wird die Vereinbarung von *x* des äußeren Blocks *verdeckt*. In (15) verdeckt die Vereinbarung von *y* im Block *C* die Vereinbarung von *y* im Block *B*.

D₂

Sprachen, die explizit oder implizit Blöcke als Konzept kennen, heißen *blockstrukturierte Programmiersprachen*. Die meisten blockstrukturierten Programmiersprachen können unbeschränkt viele Blöcke ineinander schachteln. In Fortran sind jedoch nur drei erlaubt.

2.4 Datentypen und Ausdrücke

Variablen und ihre Werte können nach verschiedenen Gesichtspunkten klassifiziert werden. *Typen* partitionieren die Menge der Werte nach den auf sie anwendbaren Operationen. im Sinne der Theorie abstrakter Datentypen in Abschnitt 2.1.1.

Der Typ einer Variablen definiert die Menge ihrer möglichen Werte. Technisch ist jeder Wert durch eine Bitfolge codiert; der Typ definiert die Interpretation solcher Bitfolgen als ganze oder Gleitpunktzahlen, als boolesche Werte, Texte, Gruppen solcher Werte usw. Hängt die Interpretation und damit der Typ von der jeweils ausgeführten Operation ab, so heißt die Programmiersprache *typfrei*, sonst *typisiert* oder *typgebunden*. Typisierte Sprachen können in zwei Richtungen weiter klassifiziert werden: Besitzt jede Variable und jeder Wert einen festen, unveränderlichen Typ, so heißt die Sprache *statisch typisiert*. Kann sich der Typ einer Variablen durch Zuweisung eines neuen Wertes ändern, heißt sie *dynamisch typisiert*. Die Typbindung heißt *stark*, wenn *nur* die für den jeweiligen Typ definierten Operationen zulässig sind. Sie heißt *schwach*, wenn es Lücken in der Typkontrolle gibt.

Beispiel:

Die Programmiersprache *Pascal* besitzt an sich statische, starke Typbindung. Beim Umgang mit Verbunden mit Varianten wird jedoch nicht geprüft, ob der Variantenzeiger tatsächlich richtig gesetzt ist, wenn auf eine Komponente einer Variante zugegriffen wird. Hier ist die Typisierung nicht lückenlos. Zu den Sprachen mit schwacher Typbindung gehören auch C, COBOL, Fortran, Eiffel, Lisp und Prolog.

Beispiel:

Bei der Datenkommunikation werden Bitfolgen ohne Rücksicht auf ihre Bedeutung umcodiert und übertragen. Zur effizienten Berechnung einer Haschfunktion aus einem Bezeichnertext werden Teile der Codierung des Textes als ganze Zahl uminterpretiert. Eine starke, lückenlose Typbindung ist in diesen Fällen hinderlich.

Beispiel:

Funktionale und manche objektorientierte Sprachen wie Sather und Java sind zwar stark typisiert. Jedoch läßt sich bei polymorphen Aufrufen der Typ nicht statisch aus dem Programmtext ableiten. Hier liegt dynamische Typbindung vor. Die vorgenannten Sprachen garantieren durch statische Analyse, daß ein korrektes Programm zur Laufzeit keine Typfehler liefert. In Smalltalk gibt es keine Möglichkeit, durch statische Analyse vorab Typfehler vollständig auszuschließen.

2.4.1 Grundtypen

Auf Objekte der Grundtypen kann nur als Ganzes zugegriffen werden. Im wesentlichen sind in allen Programmiersprachen die Grundtypen *INT*, *FLOAT*, *CHAR* und *BOOL* für ganze Zahlen, Gleitpunktzahlen, Zeichen und Wahrheitswerte gebräuchlich.

Der Typ *INT* der *ganzen Zahlen* ist im allgemeinen auf den für die Zielmaschine verfügbaren Ausschnitt der ganzen Zahlen beschränkt. Deshalb gibt es eine kleinste ganze Zahl *minint* und eine größte ganze Zahl *maxint*; bei arithmetischen Operationen kann Überlauf

auftreten. Manche Programmiersprachen fordern in diesem Fall eine Fehlermeldung (Algol 68, Pascal, Modula-2, Modula-3 und Ada); andere erlauben auch die Implementierung als Ring ohne Überlaufmeldungen (C, C++). Abgesehen vom Überlauf verhalten sich die arithmetischen Operationen bis auf die Division *div* und die Berechnung des Restes *mod*, wie in der Mathematik üblich. In vielen Programmiersprachen wird bei $x \text{ div } y$ immer zur 0 hin gerundet. Beispielsweise ergibt in *Pascal* die Berechnung $-3 \text{ div } 2 = -1$ und $3 \text{ div } 2 = 1$. Die Berechnung des Restes ist der Definition der Division angepaßt.

Ähnlich dem Typ der ganzen Zahlen bietet der Typ *FLOAT* der *Gleitpunktzahlen* die üblichen arithmetischen Operationen an. Die Gleitpunktarithmetik ist letztlich durch die Zielmaschine definiert, die heute meist die IEEE-Norm erfüllt. Neben den arithmetischen Operationen gibt es noch eine Operation *inttoflt*: $INT \rightarrow FLOAT$, die ganze Zahlen in Gleitpunktzahlen konvertiert. Manche Programmiersprachen bieten zusätzlich zum Typ *FLOAT* den Typ *DOUBLE* der *Gleitpunktzahlen mit doppelter Genauigkeit* an sowie Konvertierungsoperationen zwischen diesen beiden Typen.

Der Typ *CHAR* beschreibt den Zeichensatz der Zielmaschine, z.B. die Zeichen nach ISO 8859-1. Die Zuordnung der Zeichen zu ganzen Zahlen ist in manchen Programmiersprachen durch Operationen *ord*: $CHAR \rightarrow INT$ und *char*: $INT \rightarrow CHAR$ explizit zugänglich. Dann können auch Zeichen mit < oder > miteinander verglichen werden.

In manchen Programmiersprachen wird *true* mit 1 und *false* mit 0 identifiziert. Mit dieser Einbettung in den Typ *INT* ist auch ein Vergleich zweier Werte vom Typ *BOOL* mit < oder > möglich. In C gibt es keinen Datentyp *BOOL*, sondern nur diese Einbettung in *INT*.

In einigen Sprachen, darunter Pascal, Modula und Ada, gibt es zusätzlich Ausschnitts- und Aufzählungstypen:

Ausschnittstypen definieren eine Teilmenge aufeinanderfolgender ganzer Zahlen oder Einzelzeichen. (16) zeigt die Definition eines Ausschnittstyps in Ada. Die Operationen der Ausschnittstypen werden wie beim Typ *INT* oder *CHAR* ausgeführt, nur tritt Überlauf schon dann ein, wenn der Ausschnitt verlassen wird.

Beispiel: Ausschnittstypen in Ada (16)

```
subtype INTAUSSCHNITT is INTEGER range 3..45;
subtype CHARAUSSCHNITT is CHAR range 'A'..'Z'
```

Aufzählungstypen zählen ihre Werte explizit auf. (17) zeigt die Definition von Aufzählungstypen in Ada. Auf den Aufzählungstypen sind eine Ordnung < entsprechend der Reihenfolge der Aufzählung der Werte und damit die Operationen *succ* und *pred* definiert, die den nächsten größeren bzw. kleineren Wert liefern. In C werden die Werte eines Aufzählungstyps als ganzzahlige Konstanten interpretiert.

Beispiel: Aufzählungstyp in Ada (17)

```
type FARBE is (rot, grün, blau)
```

2.4.2 Zusammengesetzte Typen

Gegeben seien k Ausschnittstypen I_1, I_2, \dots, I_n und ein Typ T . Dann ist die Menge der Abbildungen $RT: I_1 \times \dots \times I_n \rightarrow T$ ein k -stufiger *Feldtyp (array type)* mit Indexmengen I_1, I_2, \dots, I_n und Elementtyp T . Ein Feld a dieses zusammengesetzten Typs besteht aus $|I_1| \times \dots \times |I_n|$ Elementen. Die Elemente sind durch Indizierung $a[i_1, \dots, i_k]$, $i_j \in I_j$, zugänglich. Außer in den Sprachen Pascal und Modula sind nur Ausschnitte des Typs *INT* der ganzen Zahlen als Indexmengen zugelassen. Fortran legt die Untergrenze der Ausschnitte einheitlich mit 1, C, C++ und andere Sprachen mit 0 fest. In diesen Fällen wird der Ausschnitt durch seinen Umfang charakterisiert.

Die Anzahl k der Stufen und der Elementtyp T gehören auf jeden Fall zum Feldtyp. Bezüglich des aktuellen Umfangs unterscheidet man nach dem Bindezeitpunkt drei Fälle:

- *Statisches Feld*: Die Indexmengen I_j gehören zum Typ. Ihre Unter- und Obergrenzen müssen im Programm durch statisch berechenbare Konstanten gegeben sein. Statische Felder gibt es z.B. in Fortran, Pascal, Modula-2 und C. Sie bilden die einzigen Feldtypen, die auch theoretisch sauber behandelt werden können.
- *Dynamisches Feld*: Die Grenzen gehören nicht zum Typ. Sie werden bei Bildung eines Feldobjekts, z.B. durch eine Vereinbarung, festgelegt und sind dann fest. Beispiele sind die Feldtypen in Algol 60, PL/1 und Ada, sowie die offenen Felder in Pascal und Modula.
- *Flexibles Feld*: Die Grenzen gehören nicht zum Typ. Sie können sich bei jeder Zuweisung an eine Feldvariable ändern. Flexible Felder wurden zuerst in Algol 68 eingeführt. Sehr häufig werden in C Zeigervariablen wie flexible Felder benutzt.

Beispiel: Felddeklarationen in verschiedenen Programmiersprachen (18)

a: array[1..100, 1..100] of real;	Pascal und Modula
double a[100][100];	C
DOUBLE a(100,100);	Fortran
flex [1:100, 1:100] real a;	Algol 68
a: array (INTEGER range<>, INTEGER range<>) of FLOAT	Ada

(18) zeigt die Typvereinbarungen zweidimensionaler Matrizen in verschiedenen Programmiersprachen. Aus mehrstufigen Feldern lassen sich in manchen Sprachen Teilfelder niedrigerer Stufe ausblenden. Zum Beispiel erlaubt Fortran Vektoroperationen für Felder: $a(0:9) + b(0:9)$ bezeichnet die elementweise Addition der ersten 10 Elemente der Felder a und b .

Spezialfälle einstufiger Feldtypen sind Texte und (in eingeschränkter Form) Mengen:

Zeichenreihen (strings) oder *Texte* sind Folgen von Einzelzeichen. Sie können verkettet werden. In Sprachen wie Pascal kann auf die Einzelzeichen von Texten wie bei Feldern mit Index zugegriffen werden. Snobol und zahlreiche Makrosprachen machen Texte sogar zu den grundlegenden Werten, mit denen gearbeitet wird. Bei Texten statisch fester Länge ist die Konkatenation problematisch. Textverarbeitungssprachen wie Snobol benutzen daher flexible Felder zur Repräsentation von Texten; in Pascal ist der Typ *text* als *file of char*, also als Spezialfall des abstrakten Datentyps *SEQ(CHAR)* definiert.

Einstufige Felder mit dem Elementtyp *Bool* sind charakteristische Funktionen über ihrer Indexmenge I und beschreiben daher eine Teilmenge dieser Indexmenge. In Pascal und Modula hat man hierfür eine spezielle Notation *set(I)* eingeführt und erlaubt als Grundoperationen auch die üblichen Mengenoperationen Vereinigung, Durchschnitt, Differenz und Test, ob Element vorhanden.

Verbundtypen definieren kartesische Produkte von Typen T_1, T_2, \dots, T_k . (19) zeigt die Definition von Verbundtypen in verschiedenen Programmiersprachen. Die Variablen t_1, \dots, t_k bezeichnen die k Komponenten eines Verbundtyps. Die Projektion auf die i -te Komponente eines Verbundobjekts x , eine *Komponentenselektion*, wird in den meisten Programmiersprachen mit $x.t_i$ bezeichnet. Mögliche Werte eines Verbundobjekts sind Tupel von Werten der Einzeltypen. Der Verbundwert kann durch Einzelzuweisung der Komponenten bestimmt werden, siehe Abschnitt 2.5.1. In manchen Sprachen, z.B. Ada und C, können die Tupel explizit als Aggregate angegeben werden. Die Komponentenbezeichner t_i und die Komponententypen sind Teil des Verbundtyps. In vielen funktionalen Sprachen sind die Komponenten allerdings unbezeichnet und können nur durch Positionsangabe – erste, zweite, ... Komponente des Tupels – ausgewählt werden.

Beispiel: Verbundtypen in verschiedenen Programmiersprachen (19)

```
type T = record t1:T1; ...; tk:Tk; end;    Pascal, Modula und Ada
typedef struct{T1 t1; ...; Tk tk} T      C
```

2.4.3 Variablen in imperativen Sprachen, Zeigertypen

Eine *Variable* in einer imperativen Programmiersprache ist ein Exemplar des abstrakten Datentyps $Variable(T)$ aus (7). Außer in Lisp, funktionalen und objektorientierten Sprachen wird durchgängig *Wertsemantik* verwendet. Die Variable ist damit ein Tripel (*Referenz*, *Behälter*, *Wert*) mit der *Referenz* als unveränderlichem Kennzeichen. Der *Behälter* ist die eigentliche Variable, sein Inhalt ist ihr *Wert*. Die Variable heißt eine *Konstante*, wenn der Wert unveränderlich ist. Wenn man zwischen Variablen und Konstanten nicht unterscheiden will, spricht man von *Größen*, *Angaben* oder *Objekten*. Wird ein solches Objekt durch einen Bezeichner *bez*, eine indizierte Benennung $a[i]$ oder eine Komponentenselektion $x.t$ angesprochen, so sind diese Benennungen an eine Zugriffsfunktion gebunden, deren Berechnung den Zeiger und dann bei Bedarf den Wert des Objekts liefert. In BCPL und C wird der Zeiger und der Wert auch als *left hand* und *right hand value* der Variablen bezeichnet. Der Übergang zum Wert einer durch Zeiger gegebenen Variablen heißt *Dereferenzieren*.

In älteren Programmiersprachen wie Fortran, COBOL oder Algol 60 treten Zeiger und das Dereferenzieren nicht explizit in Erscheinung. Algol 68, Pascal und alle neueren Sprachen kennen Zeiger als explizite Werte eines *Zeigertyps* *pointer to T* mit *T* als dem *Bezugstyp*. Dereferenzieren setzt voraus, daß das angesprochene (Bezugs-)Objekt existiert (Problem der *dangling references*); die meisten Sprachen beschränken die Wertmenge von Zeigertypen daher auf Referenzen von Objekten unbeschränkter Lebensdauer. Neben dem Dereferenzieren können *anonyme* Objekte des Typs *T* erzeugt werden. Die Erzeugung eines anonymen Objekts liefert einen Zeiger auf dieses Objekt. Es ist auch üblich, leere Zeiger einzuführen, die auf kein Objekt verweisen. Die leeren Zeiger heißen *nil* (z.B. Pascal, Modula) oder *void* (z.B. C, C++). Der Vergleich zweier Zeiger liefert genau dann *true*, wenn die Zeiger auf das gleiche anonyme Objekt verweisen oder beide Zeiger leer sind. In C wird zur Erklärung von Referenzen auf die Implementierung mit Hilfe von Adressen zurückgegriffen: Zusätzlich zum Dereferenzieren sind auch die für Adressen zulässigen Verknüpfungen mit ganzen Zahlen erlaubt: Ist *r* eine Referenz und *i* eine ganze Zahl, so haben $r + i$ und die indizierte Benennung $r[i]$ die gleiche Bedeutung; das Problem der *dangling references* wird ignoriert.

Mit Zeigertypen lassen sich rekursive Datenstrukturen definieren, in denen Referenzen auf den Gesamttyp eines zusammengesetzten Objekts zugleich als Elemente vorkommen. (20) zeigt die Definition eines Typs *BINOP*, mit dem zwei Ausdrücke über binäre Operatoren verknüpft werden können.

Beispiel: Zeigertypen in Ada (20)

```
type BINOP is record left, right: access BINOP; operator: STRING; end;
```

Funktionale Sprachen und im Standardfall objektorientierte Sprachen wie Smalltalk, Eiffel und Java betrachten jeden Typ als Zeigertyp; sie verwenden also *Referenzsemantik*. Der Übergang zum Wert des Objekts, das Dereferenzieren, erfolgt nur, wenn der Kontext dies zwingend erfordert.

2.4.4 Vereinigungstypen und polymorphe Typen

Ein *Vereinigungstyp* beschreibt eine disjunkte Vereinigung von Typen T_1, \dots, T_k . Objekte dieses Typs können wahlweise Werte eines der Typen T_i annehmen. (21) zeigt die Vereinbarung einer Variablen x mit Vereinigungstyp. Auch C und C++ besitzen Vereinigungstypen. In diesen Sprachen wird nicht geprüft, ob die Operationen mit einem Objekt eines solchen Typs auch für den aktuellen Wert und seinen Typ legal sind.

Beispiel: Vereinigungstypen in Algol 68 (21)

```
mode T = union(T1, ..., Tk);
T x;
```

Sei T_i ein Verbundtyp mit Feld t_i und x eine Variable vom Typ $\text{union}(T_1, \dots, T_k)$. Dann ist der Zugriff $x.t_i$ auf die Komponente t_i von x nur dann erlaubt, wenn T_i der Typ des aktuellen Wertes von x ist.

Andere Programmiersprachen erlauben an Stelle von Vereinigungstypen variante Verbunde (Pascal, Modula, Ada). (22) zeigt, wie in Ada ein Typ *OP* mit den Varianten *BINOP* und *CONST* (Konstante) definiert wird. Ähnliche Definitionen erlauben Pascal und Modula. Zur Definition eines varianten Verbundes sollte eine Komponente *diskr*: *A* eines Aufzählungstyps mit den entsprechenden Varianten, der *Varianteanzeiger*, vorhanden sein. Der Wert dieser Komponente legt die Variante des aktuellen Werts fest. Zusätzlich kann man Komponenten, die allen Varianten gemeinsam sind, im gemeinsamen Teil vor der Variante definieren. Wegen der varianten Verbunde müssen in Pascal, Modula und Ada Typprüfungen zur Laufzeit erfolgen. Sonst wären diese Programmiersprachen statisch typisiert. In Pascal und Modula kann der Varianteanzeiger sogar fehlen. In Ada kann man bei der Vereinbarung einer Variablen angeben, welche Variante ausgewählt wird: $x: \text{BINOP}(\text{kind} \Rightarrow \text{op})$ gibt an, daß bei x die Variante *op* ausgewählt wird. Die Variable x kann dann nur Werte dieser Variante enthalten.

Varianten Verbund in Ada (22)

```
type BINOP(kind:(const,op)) is
  record
    case kind is
      when const => val:T
      when op => left,right: access BINOP; operator:('*','+');
    end case;
  end record;
```

Ein Vereinigungstyp heißt *polymorph*, wenn die Interpretation der auf ein Objekt angewandten Operatoren oder Prozeduren vom Typ des aktuellen Werts abhängt. Dies entspricht der polymorphen Bindung von Operationssymbolen in Abschnitt 2.3.1. Das polymorphe Auswahlverfahren kann ein- oder mehrstufig sein: In Smalltalk, Eiffel, C++ und Java hängt die Auswahl der Operation nur vom ersten Operanden ab, was auch syntaktisch unterschieden wird: $a + b$ ist eine Kurzschreibweise für $a.\text{plus}(b)$ mit a als ausgezeichnetem ersten Operanden. In C++ muß der Operator oder die Prozedur explizit als virtuell vereinbart sein. In CLOS hängt die Auswahl von sämtlichen Operanden (oder Prozedurargumenten) ab. Die Elementtypen eines polymorphen Vereinigungstyps heißen auch *Untertypen* des polymorphen Typs.

Polymorphe Vereinigungstypen gibt es in allen objektorientierten Sprachen. In funktionalen Sprachen gibt es das gleiche Konzept, jedoch werden die polymorphen Typen nicht explizit benannt, sondern ihre Elementtypen aus dem Kontext bestimmt.

2.4.5 Typäquivalenz

Bei der Übergabe von Objekten als Prozedurparameter oder allgemeiner bei der Übergabe eines Objekts zwischen verschiedenen Modulen oder Klassen einer modularen oder objektorientierten Sprache wechselt der Objekttyp im allgemeinen nicht. Es muß daher, wie bereits in Abschnitt 2.1.3 erwähnt, festgestellt werden können, ob und wann die auf beiden Seiten einer solchen Schnittstelle erklärten Typen als gleich gelten. In der Nachfolge von Pascal verwendet man dazu in allen neueren Sprachen *Namensgleichheit*: Zwei Typen gelten als gleich, wenn sie gleich benannt sind. Damit wird zugleich das Geheimnisprinzip an Modulgrenzen gewahrt: Ein Modul kann Objekte benutzen, deren Repräsentation, wie sie in der Typdefinition zum Ausdruck kommt, verborgen bleibt. Bei *Strukturgleichheit* sind zwei Typen äquivalent, wenn ihre Algebren als abstrakte Datentypen betrachtet isomorph sind. Namensgleichheit bereitet bei Feldern Schwierigkeiten: Wird etwa ein Statistikpaket und ein Modul mit Routinen der linearen Algebra zusammengesetzt, so kommen in beiden Teilen Typen für Vektoren und Matrizen vor, die zwar austauschbar, aber eventuell unterschiedlich benannt sind. Daher benutzt man als Ausnahme von der Regel für Felder Strukturäquivalenz.

Namens- oder Strukturgleichheit bestimmen jeweils eine Äquivalenzrelation auf der Menge der Typen eines Programms. In (23) sind die Typen von u, v, w, x, y, z alle strukturgleich, während die Typen von x und y bzw. von u und v namensgleich sind. Auch die beiden Typen C und D sind strukturgleich. Die Typen von x und z bzw. u und w sind natürlich nicht namensgleich. Bei der Strukturgleichheit dynamischer und flexibler Felder wird die Anzahl der Stufen und der Elementtyp berücksichtigt, nicht jedoch die Anzahl der Elemente.

Beispiel: Typäquivalenz

(23)

```

type A = record a: integer; b: real end;
type B = record a: integer; b: real end;
x: A;
y: A;
z: B;
u,v: record a: integer; b: real end;
w: record a: integer; b: real end;
type C = record i: integer; j: pointer to C end;
type D = record i: integer; j: record i: integer; j: pointer to D end;

```

D₂

2.4.6 Ausdrücke

Ausdrücke berechnen (Zwischen-)Ergebnisse durch Anwendung von Operationen auf ihre Operanden. Im Sinne von Abschnitt 2.1.1 kann man Ausdrücke syntaktisch auch als Terme mit Variablen betrachten, wobei die Signatur durch die Sprachdefinition und das Programm (siehe Abschnitt 2.5.2) festgelegt ist.

Ein Ausdruck regelt zunächst die Reihenfolge der Operandenzugriffe und der Operationen. Diese beiden Reihenfolgefestlegungen sind deutlich zu unterscheiden: Da in der Rechnerarithmetik weder für ganze noch für Gleitpunktzahlen das Assoziativgesetz gilt, legen die meisten Programmiersprachen explizit fest, daß Ausdrücke wie $a + b + c$ von links nach rechts ausgewertet werden, also zuerst $a + b$ berechnen und danach c addieren. Daraus folgt jedoch nicht, daß zuerst auf den Operanden a , danach auf b und zum Schluß auf c zugegriffen werden *muß*. Vielmehr stellen es die meisten Programmiersprachen frei, in welcher Reihenfolge die Operanden bereitgestellt werden.

Beispiel:

In Pascal (und ebenso in Ada und vielen anderen Sprachen) muß zwar ein Ausdruck $i + j + f(i, j)$ linksassoziativ als $(i + j) + f(i, j)$ interpretiert werden. Sollte jedoch die Funktion f durch

```
function f (var a,b: integer) : integer; begin a := a + 1; b := b - 1; f := a + b end;
```

mit Nebenwirkungen auf die Werte der Argumente i, j vereinbart sein, so läßt sich das Ergebnis nicht vorhersagen: Der Funktionsaufruf $f(i, j)$ könnte sowohl vor wie auch nach den Zugriffen auf die Werte der Variablen i, j stattfinden. Es bleibt offen, ob die Nebenwirkungen des Aufrufs auf die Werte von i, j bereits eingetreten sind oder noch nicht. Praktisch ist dies nicht zu beanstanden, da unter solchen Umständen das Gebot der sauberen und verständlichen Programmierung auch für den menschlichen Leser verletzt ist. Die Idealforderung, daß Funktionsaufrufe keine Nebenwirkungen haben dürfen, kollidiert in objektorientierten Sprachen mit den Konsequenzen des Geheimnisprinzips und in allen Sprachen mit den praktischen Bedürfnissen beim Programmtest und findet sich daher in keiner der gängigen Programmiersprachen.

Die zulässigen Operationen ergeben sich aus den Typisierungsvorschriften der Sprache. Die Anwendung der Operationen ist durch Regeln definiert, bei denen die Operatoren der Ausdrücke auf Operationen auf dem entsprechenden Typ abgebildet werden.

Beispiel:

Die Funktion *eval* definiert die Auswertung von Ausdrücken, wie sie typisch für viele Programmiersprachen ist. In imperativen Programmiersprachen, ist sie immer als Berechnung in einem bestimmten Zustand zu verstehen. Die folgende Gleichung zeigt einen Ausschnitt einer Definition, wobei $e_1 = eval(E_1)$ und $e_2 = eval(E_2)$ ist.

$$\begin{aligned} eval(x) &= content(x) \\ eval(E_1 + E_2) &= e_1 \oplus e_2 && \text{falls } (type(e_1) = INT) \wedge (type(e_2) = INT) \\ &= inttoflt(e_1 \diamond e_2) && \text{falls } (type(e_1) = INT) \wedge (type(e_2) = FLTD) \\ &= e_1 \diamond inttoflt(e_2) && \text{falls } (type(e_1) = FLTD) \wedge (type(e_2) = INT) \\ &= e_1 \diamond e_2 && \text{falls } (type(e_1) = FLTD) \wedge (type(e_2) = FLTD) \end{aligned}$$

content ergibt den Wert einer Variablen, *type* den Typ eines Wertes, \oplus definiert die Addition auf ganzen Zahlen und \diamond definiert die Addition auf Gleitpunktzahlen. Wichtig ist in diesem Zusammenhang die Unterscheidung zwischen den Operationen der Typen *INT* und *FLT* und dem Operatorsymbol $+$. Erstere gehören zu den entsprechenden Algebren, letzteres ist Teil der Syntax der Programmiersprache.

Operatoren in Ausdrücken, die durch *eval* abhängig vom Typ der Operanden auf Operationen bestimmter Typen abgebildet werden, nennt man *überladen*. Ist der Wert eines Operanden vom Typ T , wird aber ein Wert eines Typs T' erwartet, dann wird der gegebene Wert an den Typ *angepaßt*.

Beispiel:

Der Operator „+“ in Ausdrücken ist überladen. Er wird abhängig vom Typ der Operanden auf die Addition ganzer Zahlen \oplus oder auf die Addition von Gleitpunktzahlen \diamond abgebildet. Ist einer der Operanden von „+“ vom Typ *INT* und der andere vom Typ *FLT*, dann wird der Operand vom Typ *INT* mit *inttoflt* automatisch an *FLT* angepaßt. Sowohl das Überladen arithmetischer Operatoren als auch das Anpassen von ganzen Zahlen an Gleitpunktzahlen ist in den meisten Programmiersprachen gebräuchlich. Die Anpassung erfolgt oft automatisch, kann aber auch durch explizite Konvertierungsoperationen wie *inttoflt* oder *fltoint* (mit Rundung) explizit geschrieben werden.

In stark typisierten funktionalen Sprachen werden in der Signatur meist keine expliziten Typen, sondern nur Typvariablen angegeben. Der Typ der Operationen, Variablen und Konstanten wird durch *Typinferenz* aus dem Programmtext bestimmt. Jede Typvariable wird dabei durch den allgemeinsten zulässigen parametrisierten Typ ersetzt.

Boolesche Ausdrücke berechnen Wahrheitswerte. In den meisten Programmiersprachen ist dies durch *Kurzauswertung* definiert, die gar nicht alle Operanden berechnet.

$$\begin{aligned} \text{eval}(E_1 \wedge E_2) &= \text{false} && \text{falls } \text{eval}(E_1) = \text{false} \\ &= \text{eval}(E_2) && \text{falls } \text{eval}(E_1) = \text{true} \\ \text{eval}(E_1 \vee E_2) &= \text{true} && \text{falls } \text{eval}(E_1) = \text{true} \\ &= \text{eval}(E_2) && \text{falls } \text{eval}(E_1) = \text{false} \end{aligned}$$

Ada bietet für die Kurzauswertung (*and then, or else*) und für die vollständige Auswertung (*and, or*) verschiedene Operatoren an.

2.5 Sequentielle Ablaufsteuerung

Nach den theoretischen Erkenntnissen über *while*-Sprachen genügen bedingte Ausdrücke bzw. Anweisungen zusammen mit Schleifen oder rekursiven Funktionen, um beliebige Algorithmen auszudrücken. In imperativen Sprachen ist zusätzlich die Zuweisung erforderlich. Das Prinzip der prozeduralen Abstraktion führt zwingend zur Einführung von Prozeduren (oder Funktionen) und entsprechenden Aufrufen. Die Ablaufsteuerung definiert, wie diese Anweisungen ausgeführt werden und damit der Gesamt Ablauf eines Programms gesteuert wird.

In maschinennahen Sprachen sind unbedingte und bedingte Sprünge sowie Unterprogramm Sprünge das wesentliche Hilfsmittel der Ablaufsteuerung. Allerdings sind Sprünge nur dann für den Programmierer sinnvoll, wenn der Befehlszähler des Rechners explizit wie eine Variable behandelt werden kann, und Sprünge demnach als zustandsverändernde Zuweisungen an den Befehlszähler angesehen werden. Dies ist nur naheliegend, wenn endliche Automaten als Programmiermodell verwendet werden (siehe hierzu [Knuth 74]). Ferner kann die unbeschränkte Verwendung von Sprüngen bei der Programmanalyse zu irreduziblen Ablaufgraphen führen und damit den Übersetzer bei der Programm- und Datenflußanalyse erheblich beschränken. Auf Sprünge wird im folgenden nicht eingegangen.

In imperativen, höheren Sprachen ist die Zuweisung die wesentliche, zustandsverändernde Operation. Die Zustandsänderung ist eigentlich eine Nebenwirkung. Andere Sprachelemente können durch die in ihnen enthaltenen Zuweisungen ebenfalls Nebenwirkungen hervorrufen.

Schleifen setzen zwingend Variablen wie in imperativen Sprachen voraus. In funktionalen und logischen Programmiersprachen ohne einen solchen Variablenbegriff gibt es daher nur bedingte Ausdrücke und rekursive Funktionen; Schleifen treten nur als Abkürzungen für rechtsrekursive Funktionen auf.

Zur Ablaufsteuerung in parallelen Programmen siehe Kapitel D4, zur Ablaufsteuerung in logischen Programmen Kapitel D6.

2.5.1 Zuweisungen, bedingte Anweisungen und Schleifen

Zuweisungen

Nach einer Zuweisung $x := e$ besitzt die Variable x das Ergebnis der Berechnung des Ausdrucks e als Wert. Der Typ des Resultats von e muß an den Typ von x automatisch anpaßbar sein. Diese Definition ist typisch für zustandsorientierte Programmiersprachen. In COBOL werden Zuweisungen anders geschrieben. Bild 3 zeigt Beispiele

Bemerkung:

Zuweisungen wie $x := x + 3$ können in manchen Programmiersprachen abgekürzt werden, z.B. in C zu $x += 3$, in Algol 68 zu $x +:= 3$ und in Modula 3 zu $INC(x, 3)$. Einige Sprachen (z.B. Algol 60 und PL/1) erlauben auch Zuweisungen $l_1, \dots, l_k := e$ mit mehreren Zielen. Die *parallele Zuweisung* $(x, y) := (a_1, a_2)$ kommt in gängigen Programmiersprachen nicht vor.

Anweisung	Bedeutung
MOVE a TO b	$b := a$
ADD 3 TO x	$x := x + 3$
MULTIPLY x BY y GIVING z	$z := x * y$

Bild 3 COBOL-Zuweisungen.

Bedingte Anweisungen

Beim Ausführen einer bedingten Anweisung *if b then S₁ else S₂* wird zuerst die boolesche Bedingung b berechnet. Ergibt sie *true*, dann werden die Anweisungen S_1 ausgeführt, sonst die Anweisungen S_2 . Fehlt der *else*-Zweig und ist das Resultat der Auswertung von b *false*, dann wird gar keine Anweisung ausgeführt. Bei geschachtelten bedingten Anweisungen *if b then if b' then S₁ else S₂* bleibt unklar, ob die Nein-Alternative zur inneren oder äußeren Bedingung gehört, wenn die Klammerung nicht ausdrücklich festgelegt ist (Problem des *dangling else*).

Fallunterscheidung

In einer Fallunterscheidung wie in (24) muß der Fallausdruck e ein Ergebnis eines der Typen *INT*, *BOOL* oder *CHAR* liefern. Die Fallmarken l_1, \dots, l_k müssen Konstanten desselben Typs wie das Resultat von e und alle verschieden sein. In Pascal, Ada und Modula darf das Resultat von e auch einen Aufzählungstyp haben. Fortran kennt keine Fallunterscheidung. In C und C++ werden Fallunterscheidungen mit dem Schlüsselwort *switch* eingeleitet. Die Fallunterscheidung hat dann die Bedeutung der bedingten Anweisung

$\text{if } e=l_1 \text{ then } S_1 \text{ else if } e=l_2 \text{ then } S_2 \text{ else... else if } e=l_k \text{ then } S_k \text{ else } S_0$

wobei e nur einmal berechnet wird. Die Reihenfolge der Fälle ist nur dann wichtig, wenn gleiche Fallmarken oder andere Typen, z.B. Texte für Fallmarken, zugelassen sind.

Beispiel: Fallunterscheidung

(24)

```

case e of
  l1: S1
  ...
  lk: Sk
else S0
end;
```

Objektorientierte Sprachen und andere Sprachen mit Vereinigungstypen bieten Fallunterscheidungen nach Typen an, mit denen für polymorphe Variablen unterschiedliche Anweisungen ausgeführt werden, abhängig vom Typ des augenblicklichen Variablenwertes.

Schleifen

Die Bedeutung der Schleife *while b loop S* wird auf die bedingte Anweisung

```
if b then S; while b loop S;
```

zurückgeführt: Wenn die boolesche Schleifenbedingung *b* wahr ist, wird *S* ausgeführt und die Schleife wiederholt, sonst endet die Schleife. Die Schleife heißt *leer*, wenn *b* bereits zu Beginn falsch ist. Manche Programmiersprachen bieten eine spezielle Anweisung zum Beenden einer Schleife an, z.B. C und C++ die Anweisung *break* und Ada die Anweisung *exit*. Bei geschachtelten Schleifen kann man die einzelnen Schleifen auch benennen und dann mit *exit name* bestimmen, welche Schleife verlassen werden soll.

Bei Verwendung von *break* oder *exit* kann man auf die While-Bedingung verzichten. Pascal und Modula bieten Schleifen *repeat S until b* an, bei denen die Bedingung erst am Schleifenende geprüft wird.

Die Anzahl der Wiederholungen ist zu Beginn unbekannt. Namentlich zur Bearbeitung von Feldern benötigt man zusätzlich *Zählschleifen* *for i := e₁ to e₂ step c do S*, um Anweisungen mit wechselndem Index, aber fester Anzahl von Wiederholungen auszuführen. Man findet sie etwa in Pascal, Modula, C, Fortran mit wechselnder Syntax und subtilen Unterschieden der Semantik. Zum Beispiel könnte die Bedeutung dieser Schleife durch

```
i: integer;
i := e1;
while i ≤ e2 loop S; i := i + c
```

gegeben sein. Die letzte Addition *i + c* könnte dann allerdings zum Überlauf führen; auch würde der Schleifenzähler *i* nach Ende der Schleife immer einen Wert $i > e_2$ haben. Der Schleifenzähler ist entweder eine lokale Konstante, deren Wert der Programmierer in der Schleife nicht verändern kann; oder es handelt sich um eine Variable, deren Wert nach Schleifenende in vielen Programmiersprachen explizit als undefiniert bezeichnet ist.

Dem Geheimnisprinzip der modularen und objektorientierten Programmierung folgend bieten CLU und Sather *Iteratoren* an, mit denen Ströme von Elementen einer Datenstruktur in einer Schleife bearbeitet werden können [Liskov 77, Murer 96]. In der Schleife bleibt unbekannt, woher die Elemente stammen, oder wo sie abgelegt werden. Schleifen mit Iteratoren sind Verallgemeinerungen von Zählschleifen; da beispielsweise das letzte Element eines Binärbaumes nicht als solches zu erkennen ist, bricht die Schleife erst ab, wenn der Iterator nach dem letzten Element eines Eingabestroms nochmals aufgerufen wird. CLU erlaubt nur einen Iterator pro Schleife. In C++ und vergleichbaren Sprachen erreicht man Ähnliches mit einer Gruppe (*first, next, test*) von Funktionen, wobei zahlreiche Beschränkungen zu beachten sind.

2.5.2 Prozeduren und Funktionen

Funktionen $f(x_1:T_1; \dots x_n:T_n): T$ is *S end*, $n \geq 0$, implementieren Operationen mit der Signatur $f:T_1 \times \dots \times T_n \rightarrow T$ im Sinne von Abschnitt 2.1.1. Fehlt der Ergebnistyp *T*, so spricht man von *eigentlichen* oder *echten Prozeduren*. Der Begriff *Prozedur* wird gemeinhin als Oberbegriff für echte Prozeduren, (Funktions-)Prozeduren und auch für Operationen wie die ein- oder zweistellige Subtraktion gebraucht. Bei einem *Prozeduraufruf* $f(a_1, \dots, a_n)$ mit den *Argumenten* a_1, \dots, a_n werden die Parameterspezifikationen zu initialisier-

ten Vereinbarungen $x_i : T_i := a_i$ ergänzt und diese dem Rumpf S vorangestellt. Die Initialisierung heißt *Parameterübergabe*. Der so ergänzte Rumpf, ein Block, wird dann an Stelle des Aufrufs ausgeführt; bei Funktionen ist sein Ergebnis auch das Ergebnis des Funktionsaufrufs. Zur Bestimmung des Ergebnisses erlaubt Pascal Zuweisungen $f := \text{ergebnis}$ an den Funktionsbezeichner, der somit eine lokale Variable des Rumpfes vom Ergebnistyp vereinbart. (25) zeigt eine Funktion *eval* in Pascal, die mit Hilfe der Typvereinbarungen in (22) definiert ist und Ausdrücke auswertet. In vielen anderen Sprachen gibt es eine Anweisung *return*, die die Ausführung der Funktion abschließt, und zusätzlich in der Form *return ergebnis* das Funktionsergebnis liefert.

Beispiel: Prozedur in Pascal (25)

```
function eval(t:BINOP):T is
begin
  if t.kind = const then eval := t.val;
  else if t.kind = op and t.operator = '+' then eval := eval(t.left) + eval(t.right)
  else if t.kind = op and t.operator = '*' then eval := eval(t.left) * eval(t.right)
end;
```

In funktionalen Sprachen können die Parameter x_i durch *Muster (pattern)* angegeben werden, worauf hier nicht eingegangen wird. Die Angabe der Parametertypen T_i und des Ergebnistyps T unterbleibt, die Typen werden durch Typinferenz bestimmt. Es können mehrere Funktionsvereinbarungen zum gleichen Funktionsbezeichner f angegeben sein, von denen bei Aufruf die erste Vereinbarung gewählt wird, für die die Argumente auf die angegebenen Parametermuster passen. Bei einem Aufruf entspricht der Initialisierung eine Substitution der Argumente a_i an Stelle der Parameter x_i überall im Rumpf.

Die prozedurale Abstraktion ist rekursiv. Daher erlauben manche imperative Programmiersprachen wie Pascal, Modula und Ada auch im Rumpf weitere, lokale Prozedurvereinbarungen. Aus der Sicht des modularen und objektorientierten Programmierens definiert eine Funktion eine Operation auf den lokalen Daten des Moduls oder Objekts. In C, C++ und Fortran gibt es daher nur einstufige Funktionsvereinbarungen. Ada und C++ erlauben auch Operatoren (z.B. „+“, „*“) als Funktionsnamen, um diese Operatoren auch für andere Operandentypen zu definieren. Die passende Prozedurvereinbarung wird statisch ausgewählt; die Operatordefinition ist überladen, siehe Abschnitt 2.4.6.

Steuersprachen, aber auch Programmiersprachen wie Ada, erlauben Aufrufe mit *Schlüsselwortparametern* der Form $f(a_1, \dots, a_k, x_{j_1} = a_{j_1}, \dots, x_{j_r} = a_{j_r})$. Hier werden die ersten k Argumente den ersten k Parametern zugeordnet. Die restlichen Argumente gehören zu einer Auswahl x_{j_1}, \dots, x_{j_r} der verbleibenden Parameter. Etwa noch übrigen Parametern wird ein Ersatzwert (*default value*) zugeordnet; wie dieser bestimmt wird, ist unterschiedlich geregelt. Schlüsselwortparameter haben sich bewährt, wenn aus den Werten der Argumente folgt, daß bestimmte weitere Argumente nicht benötigt werden.

Die Reihenfolge der Berechnung der Argumente ist unterschiedlich geregelt: Bei Berechnung von links nach rechts können z.B. zuerst Feldgrenzen übergeben und zur Initialisierung weiterer Parameter verwandt werden. Implementierungstechnisch ist es günstiger, wenn die Reihenfolge unspezifiziert bleibt und im Belieben des Übersetzers steht.

Eine Funktion p heißt *rekursiv*, wenn während ihrer Ausführung p nochmals aufgerufen wird. Auch indirekte Rekursion, bei der p eine Funktion q aufruft, deren Ausführung p aufruft, ist möglich. Fast alle Programmiersprachen erlauben rekursive Funktionen. Verboten ist sie jedoch in COBOL und Fortran 77. (Fortran 90 erlaubt Rekursion.)

Diese Ausführungen gelten für (*Funktions-*)*Prozeduren*, die ein Ergebnis liefern, aber nur *Eingabeparameter* besitzen. Im Idealfall sind diese Prozeduren nebenwirkungsfrei (siehe Abschnitt 2.4.6). Zusätzlich kann es *Ausgabeparameter* geben, an die in der Prozedur

Teilergebnisse zugewiesen werden. Echte Prozeduren besitzen stets Ausgabeparameter. Ist ein Eingabeparameter zugleich auch Ausgabeparameter, so heißt er ein *transienter* oder *Übergangsparameter*.

Das Argument für einen Ausgabe- oder transienten Parameter muß eine Variable sein, der durch den Prozeduraufruf ein neuer Wert zugewiesen werden kann. Die meisten imperativen Sprachen erlauben überdies Zuweisungen an die in der Prozedur sichtbaren globalen Variablen, auch wenn diese nicht in der Argumentliste erscheinen. Zuweisungen an Ausgabeparameter und an globale Variablen bilden Nebenwirkungen des Aufrufs.

Parameterübergabemechanismen

In imperativen Sprachen unterscheidet man sechs verschiedene Parameterübergabemechanismen: Wertaufwurf, Ergebnisaufwurf, Wert-und-Ergebnisaufwurf, Referenzaufruf, Namensaufruf und faulen Aufruf.

Beim *Wertaufwurf* (*call by value*) wird der Parameter mit dem Argument, dem Wert eines Ausdrucks, wie zuvor beschrieben initialisiert. Bei typisierten Sprachen muß der Typ des Arguments an den Typ des formalen Parameters anpaßbar sein.

Beim *Ergebnisaufwurf* (*call by result*) wird nach Beendigung der Funktion der Wert des Parameters an das Argument, eine Variable, zugewiesen. Bei typisierten Sprachen muß der Parametertyp an den Argumenttyp anpaßbar sein. Der *Wert-und-Ergebnisaufwurf* ist eine Kombination des Wertaufwurfs und des Ergebnisaufwurfs für transiente Parameter.

Beim *Referenzaufruf* (*call by reference*) wird der formale Parameter mit einem Zeiger auf das Argument, eine Variable, initialisiert. Der Typ des Arguments muß äquivalent zum Typ des formalen Parameters sein. Argument und Parameter bezeichnen im Rumpf einer solchen Prozedur dasselbe Objekt; alle lesenden oder schreibenden Zugriffe auf den Parameter beziehen sich auf dieses nicht-lokale Objekt. Wenn zwei verschiedene Bezeichner x und y dasselbe Objekt bezeichnen, dann ist x ein *Alias* von y .

Beim *Namensaufruf* (*call by name*) wird im Prozedurrumpf der Parameter durch den Aufruf einer Funktion ersetzt, die dem Argument entspricht. Ist letzteres eine Variable, so liefert die Funktion einen Zeiger auf das Objekt, und es wird weiter wie beim Referenzaufruf verfahren; ist das Argument ein Ausdruck, so liefert die Funktion den Wert des Ausdrucks. Bei typisierten Sprachen dürfen hierbei keine Typfehler auftreten. Der Zeiger auf das Argument bzw. dessen Wert wird bei jedem Zugriff auf den Parameter neu bestimmt. Durch Nebenwirkungen kann sich das Ergebnis ändern. Die konsequente Ausnutzung dieses Umstands ist nach ihrem Erfinder als *Jensen-Trick* (*Jensen's device*) bekannt, vgl. (26): Wird diese Funktion mit $sum(i,n,a[i],b[i])$ aufgerufen, so liefert sie das innere Produkt $a_1b_1 + \dots + a_nb_n$. Die Fortschaltung des Schleifenzählers i ist eine globale Nebenwirkung, die bei der wiederholten Berechnung von $a[i]$, $b[i]$ berücksichtigt wird.

Beim *faulen Aufruf* (*call by need*) wird der Wert des Arguments, ein Ausdruck, erst berechnet, wenn er benötigt wird. Der Typ des Arguments muß an den Typ des formalen Parameters anpaßbar sein. Dieses Verfahren wird z.B. implizit bei der Kurzauswertung boolescher Ausdrücke verwandt.

Beispiel: Der Jensen-Trick in Algol 60 (26)

```

real procedure sum(i,n,x,y);
value n; integer i,n; real x,y;
begin real s; s := 0;
  for i := 1 step 1 until n do s := s + x*y;
  sum := s;
END;
```



Die meisten Programmiersprachen bieten mehrere Parameterübergabemechanismen an. Pascal und Modula erlauben Wert- und Referenzaufruf (*var*-Parameter), Ada bietet den Wertaufufr, den Ergebnisaufruf und die Kombination an (*in*-, *out*- bzw. *in out*-Parameter); Referenzaufruf wird für Felder eingesetzt. C kennt nur Wertaufufr. Fortran benutzt bei Variablen Referenz- und bei anderen Ausdrücken Wertaufufr; der Fortran-Standard sagt ebenso wie der Ada-Standard, daß Programme, bei denen Wert-/Ergebnisaufruf und Referenzaufruf unterschiedliche Resultate erbringen, unzulässig sind. Bei Programmiersprachen, die Zeiger als Werte zulassen, läßt sich ein Referenzaufruf simulieren, indem man mit dem Wertaufufr einen Zeiger übergibt; dies wird in C und allen objektorientierten Sprachen, insbesondere auch in C++ ausgenutzt. Namensaufruf kommt nur in Algol 60 vor.

Funktionale Sprachen sind nebenwirkungsfrei, so daß es keinen Unterschied zwischen einfacher und mehrfacher Berechnung eines Ausdrucks gibt. Zwar ist die Parameterübergabe durch Substitution definiert; *gierige funktionale Sprachen* (*eager evaluation*) berechnen die Argumente allerdings mit Wertaufufr (z.B. ML), *faule funktionale Sprachen* (*lazy evaluation*) benutzen faulen Aufruf (z.B. Haskell).

Bild 4 zeigt die verschiedenen Ergebnisse des Programms in (27) bei den verschiedenen Parameterübergabemechanismen. Bei faulem Aufruf ist das Ergebnis das gleiche wie bei Wertaufufr, da beide Argumente benötigt werden.

Beispiel: Ein Programm zur Demonstration der Parameterübergabemechanismen (27)

```
m:INT := 1;
n:INT;
p(??? j:INT; ??? k:INT) : INT is
  j := j + 1; m := m + k; return j + k;
end;
begin n := p(m, m + 3);end;
```

Mechanismus	<i>m</i>	<i>n</i>	<i>j</i>	<i>k</i>	Bemerkung
Wertaufufr	5	6	2	4	
Wert-/Ergebnisaufruf	2	6	2	4	Zweiter Parameter mit Wertaufufr. Reiner Ergebnisaufruf beim ersten Argument ist nicht sinnvoll.
Referenzaufruf	6	10	6	4	Erster Parameter mit Referenzaufruf. Zweiter Parameter mit Wertaufufr.
Namensaufruf	7	17	7	10	
Fauler Aufruf	5	6	2	4	Beide Argumente werden benötigt.

Bild 4 Effekt der verschiedenen Parameterübergabemechanismen im Programm in (27)

Näheres über den faulen Aufruf siehe Kapitel D5.

Beispiel: Ein Programm mit Prozedurparametern

(28)

```
procedure o;
  var n,k: integer;
  procedure p(procedure f; var j:integer);
    var i: integer;
```

```

procedure q
begin
  n := n + 1; if n = 4 then q end;
  n := n + 1; if n = 7 then (* Stelle 2 *) j := j + 1 end;
  i := i + 1
end q;

begin (* p *)
  i := 0; n := n + 1; if n=2 then p(q,i) else j := j + 1 end;
  if n=3 then (* Stelle 1 *) f end;
  i := i + 1;
end p;

procedure skip; begin end skip;

begin (* o *)
  n :=1; k := 0; p(skip,k)
end o;
    
```

Prozedurvereinbarungen kann man als Vereinbarungen konstanter Objekte ansehen. Zum Beispiel in ML, Haskell, Pascal, Modula können Prozeduren dann auch als Argumente von Prozeduren auftreten. Der Typ dieses Prozedurobjekts ist die Signatur der Prozedur. Die Prozedurvereinbarung bindet etwaige globale Größen aus ihrer Umgebung. Bei der Übergabe von Prozeduren muß neben der Prozedur daher auch diese Umgebung übergeben sein. Das Paar (*Umgebung, Prozedur*) nennt man die *Hülle (closure) der Prozedur*. Bei Aufruf eines Prozedurparameters wird zuerst diese Umgebung wieder hergestellt und dann die Prozedur ausgeführt. (28) zeigt ein Programm mit Prozedurparametern (in der Sprache Pascal). Bild 5 zeigt die Umgebungen gemäß Abschnitt 2.3.2, wenn dieses Programm die Stelle 1 bzw. 2 erreicht.

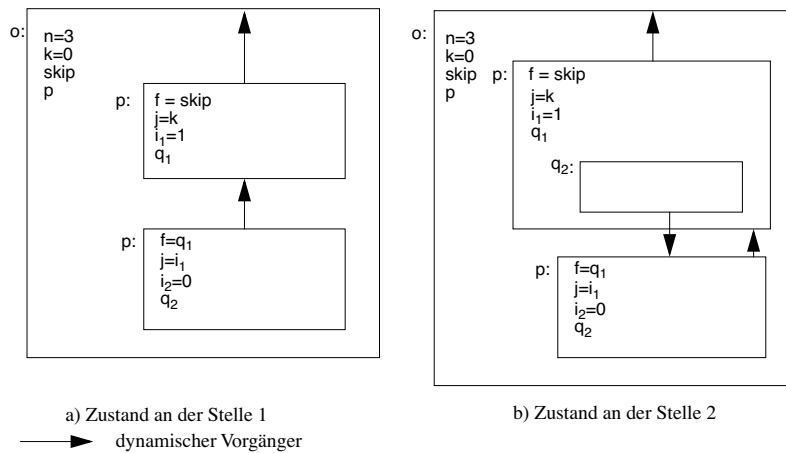


Bild 5 Zwei Zustände des Programms in (28) im Umgebungsmodell

2.5.3 Ausnahmen

Ausnahmen signalisieren, daß eine Vorbedingung einer Operation verletzt wurde, und die Operation daher nicht ordnungsgemäß zu Ende geführt werden konnte. Der Fehler kann

durch die Hardware signalisiert werden, z.B. bei Division durch 0 und arithmetischem Überlauf. Die Operation könnte aber auch ein Prozeduraufruf gewesen sein, oder der Fehler wurde durch das Laufzeitsystem der Implementierung der Programmiersprache erkannt, z.B. bei fehlerhafter Indizierung von Feldern. Ziel ist es in allen Fällen, einen konsistenten Zustand herzustellen, in dem die Programmausführung fortgesetzt werden kann; andernfalls wird der Programmablauf abgebrochen. Heutige Programmiersprachen stellen dazu eine *Raise*-Anweisung zur programmgesteuerten Auslösung von Ausnahmen und ein spezielles Sprachelement, die *Ausnahmebehandlung*, zum Abfangen von hardware- oder softwaregesteuerten Ausnahmen zur Verfügung. Eine Ausnahmebehandlung bezieht sich immer auf eine einzelne oder eine Gruppe von Fehlerursachen. Wird ein Fehler innerhalb einer Prozedur nicht abgefangen, so wird der Prozeduraufruf beendet und an der Aufrufstelle die Suche nach einer geeigneten Ausnahmebehandlung fortgesetzt. Dieses Verfahren wird fortgesetzt, bis eine Ausnahmebehandlung oder das Ende der Programmausführung erreicht ist. Ausnahmen wurden zuerst in PL/1 eingeführt. In C gibt es keine Ausnahmen; der gleiche Zweck wird durch explizite Programmierung und Abfrage unterschiedlicher Fehlercodes bei Prozedurrückkehr erreicht. (29) zeigt ein Beispiel für Ausnahmebehandlung in Ada. Bei einem Aufruf $f(n,0)$ löst die Division die Ausnahme *Constraint_Error* aus. In diesem Fall wird unmittelbar die Ausnahmebehandlung $x := 0$ ausgeführt.

Beispiel: Ausnahmebehandlung in Ada (29)

```

procedure f(n,m:INTEGER) is
  x := n / m; x := x + 1;
exception
  when Constraint_Error => x := 0;
  when others => x := maxint;
end;
```

Ist e eine vom Programmierer definierte Ausnahme, so kann sie in Ada wie in (30) mit *raise e* ausgelöst werden. Wird die Prozedur *invert(A)* mit einer nicht-quadratischen Matrix A aufgerufen, dann löst dieser Aufruf die Ausnahme *Non_Square_Matrix* aus. Wenn A singular ist, dann löst der Aufruf von *invert(A)* die Ausnahme *Singular_Matrix* aus.

Beispiel: Ausnahmefinition und -auslösung in Ada (30)

```

function invert(A:array( INTEGER range <>, INTEGER range <>) of REAL) is
  Non_Square_Matrix: exception, Singular_Matrix: exception;
  if A'Length ≠ A'Length(2) then raise Non_Square_Matrix
  else if determinant(A) = 0 then raise Singular_Matrix
  else ...
  end if;
end; invert
```

In CLU, Modula-3 und Java gehören Ausnahmen zur Signatur einer Prozedur oder Funktion. CLU, Modula-3 und ML erlauben Ausnahmen mit Parametern. Beispielsweise kann man eine Ausnahme mit der auszugehenden Fehlermeldung parametrisieren.

2.6 Modularität und Objektorientierung

Die Sprachelemente in Abschnitt 2.5 unterstützen zwar das strukturierte Programmieren, jedoch nicht das Programmieren im Großen. Die hier vorgestellten Sprachelemente erlauben es, Teile eines Programms zu Einheiten zusammenzufassen, auf die nur über eine Schnittstelle zugegriffen werden kann; die Einzelheiten der Implementierung unterliegen

dem Geheimnisprinzip aus Abschnitt 2.1.3 und sind an der Schnittstelle verborgen. Man spricht von einer *öffentlichen* und einer *privaten Sicht* auf den Programmteil.

2.6.1 Module

Ein *Modul* ist ein schwarzer Kasten, mit dem über eine Schnittstelle kommuniziert werden kann. *Modulare Programmiersprachen* wie Modula und Ada bieten Module als Sprachkonzept an. Die *Schnittstelle* eines Moduls besteht aus Typ- und Variablenvereinbarungen, Signaturen von Prozeduren usw. Die *Implementierung eines Moduls* besteht aus der Implementierung der Prozeduren der Schnittstellen (eventuell mit Hilfe zusätzlicher Typen und Variablen). Zusätzlich gibt es *import*- (in Ada: *with*- und *use*-)Anweisungen, um den Zugriff auf die Schnittstelle anderer Module ganz oder teilweise zu ermöglichen.

Beispiel:

(31) zeigt die Schnittstellendefinition eines Moduls in Ada, das Ausdrücke aufbaut, Zeichenketten in Ausdrücke zerteilt und Ausdrücke auswertet. Module heißen in Ada *Pakete*. Die Typvereinbarung *type OP is private* sorgt dafür, daß die Einzelheiten des Typs dem Benutzer des Pakets verborgen bleiben. Für Objekte eines solchen privaten Typs ist außerhalb des Pakets nur die Gleichheit und die Zuweisung definiert. Mit *limited private* kann selbst dies verboten werden. Die vollständige Typvereinbarung ist im privaten Teil der Paketvereinbarung enthalten. (Da der private Teil eigentlich zur Implementierung gehört, ist er an dieser Stelle unlogisch, aber bei getrennter Übersetzung technisch erforderlich; in Modula gibt es keinen solchen privaten Teil, dafür können öffentlich sichtbare Typvereinbarungen nur Zeigertypen definieren.) (32) zeigt die Implementierung des Moduls mit der Schnittstelle in (31).

Beispiel: Schnittstellendefinition eines Moduls in Ada (31)

```
package Ausdruck is
  type OP is private;
  function create_const(i:INTEGER) return access OP;
  function create_mult(l,r:access OP) return access OP;
  function create_add(l,r:access OP) return access OP;
  function eval(x:access OP) return INT;
  function parse(s:STRING) return access OP;
private
  type OP(kind:(const,op)) is record
    case kind is
      when const => val:INTEGER
      when op => left,right: access OP; operator:('*','+');
    end case;
  end record;
end Ausdruck;
```

Beispiel: Implementierung des Moduls in (31) (32)

```
package body Ausdruck is
  function create_const(i:INTEGER) return access OP is
  begin return( new OP'(kind => const, val =>i))
  end create_const;

  function create_mult(l,r:access OP) return access OP is
  begin return(new OP'(kind => op, left=>l, right=>r, operator='*'))
  end create_mult;

  function create_add(l,r:access OP) return access OP is ... end create_add;
```



```

function eval(x:access OP) return INT is
begin
  if x.kind = op then return(x'Access.val)
  elsif x.operator = '+' then return( eval(x'Access.l) + eval(x'Access.r))
  else return (eval(x'Access.l) *eval(x'Access.r))
  end if;
end eval;
end Ausdruck;

```

Bemerkung:

Der Modulbegriff ist an sich rekursiv: Module können zu größeren Modulen zusammengefaßt werden. Ada und Modula-2 erlauben daher Untermodule im Sinne syntaktischer Schachtelung. Praktisch hat sich diese Schachtelung nicht bewährt, weshalb sie in Modula-3 nicht aufgenommen wurde.

Andere Teile des Programms können die Schnittstelle eines Moduls *A* auf zwei Arten benutzen. Entweder sie importieren (Teile von) *A* – dann gehört die Schnittstelle von *A* zur Umgebung, in der sie importiert wird – oder die Schnittstellen von *A* werden qualifiziert angesprochen.

Beispiel:

(33) und (34) zeigen zwei Programmstücke in Ada. Das erste benutzt die Schnittstelle des Moduls *Ausdruck* (31) qualifiziert, das zweite importiert die Schnittstelle. In Ada kann die Schnittstelle eines Moduls nur insgesamt importiert werden. In Modula können auch Teile der Schnittstelle importiert werden.

Beispiel: Benutzung des Ada-Moduls aus (31) qualifiziert (33)

```

function werte_aus(s:STRING) return INTEGER is
  x:Ausdruck.OP;
begin x := Ausdruck.parse(s); return(Ausdruck.eval(x));
end werte_aus;

```

Beispiel: Benutzung des Ada-Moduls aus (31) importiert (34)

```

function werte_aus(a:STRING) return INTEGER is
  use Ausdruck;
  x:OP
begin x := parse(s); return(eval(s));
end werte_aus;

```

Die Implementierung eines Moduls ist im allgemeinen genau einer Schnittstellendefinition zugeordnet. Um verschiedene Sichten auf die gleiche Datenstruktur zu definieren, möchte man manchmal einer Implementierung eines Moduls verschiedene Schnittstellen zuordnen. Dies erlaubt etwa Modula-3.

2.6.2 Klassen und Objekte

In modularen Programmiersprachen werden abstrakte Datentypen durch Module realisiert. Die Modulschnittstelle muß einen Typ *T* definieren, der die Repräsentation der Objekte dieses abstrakten Datentyps bestimmt. Exemplare des abstrakten Datentyps können dann durch Vereinbarung (oder anonyme Erzeugung) von Objekten des Typs *T* gebildet werden. In objektorientierten Sprachen wird, beginnend mit Simula 67, der Modul mit der Typdefinition von *T* identifiziert, indem man die Prozedurvereinbarungen für die Operationen mit Objekten dieses Typs in die Typvereinbarung verlegt. Diese erweiterte Typvereinbarung heißt dann eine *Klassenvereinbarung*. Statt vom Typ eines Objekts spricht man von der *Klasse* des Objekts.

Eine Klasse *class A is m₁; ...; m_n; end*; besteht aus Merkmalen *m₁; ...; m_n*. Ein *Merkmal* (in C++, Java und C#: *Mitglied (member)*) ist eine Attributvereinbarung oder eine Methodenvereinbarung. Eine *Attributvereinbarung* ist eine Variablenvereinbarung *x:T* mit einem Typ, d.h. einer Klasse, *T*. Eine *Methodenvereinbarung* ist eine Prozedur- oder Funktionsdefinition *p(x₁: T₁, ..., x_n: T_n) is S end*; bzw. *p(x₁: T₁, ..., x_n: T_n): T is S end*, wobei die *T₁, ..., T_n, T* Typen, d.h. Klassen, definieren. Programmiersprachen, die Klassen in der hier vorgestellten Form anbieten, heißen *objektgestützt* oder *objektbasiert*. Können in einer objektgestützten Programmiersprache zusätzlich Klassen von anderen Klassen im Sinne von Abschnitt 2.6.3 erben, dann heißt sie *objektorientiert*. Beispiele objektorientierter Programmiersprachen sind Simula 67, C++, Smalltalk, Eiffel, Java, C#, Sather und Fortran2003.

Eine Klasse definiert zugleich einen Typ. Eine Klasse *A* ist eine *Referenzklasse*, wenn der durch *A* definierte Typ ein Zeigertyp ist, sonst ist *A* eine *Wertklasse*. Wenn eine Wertklasse *A* lediglich die Attribute *a₁: T₁; ...; a_n: T_n* enthält, dann definiert *A* den gleichen Typ wie *record a₁: T₁; ...; a_n: T_n end* in Abschnitt 2.4.2; im Fall einer Referenzklasse handelt es sich um den Typ *pointer to record a₁: T₁; ...; a_n: T_n end*. Auf die Komponenten einer Variablen *x* vom Typ *A* kann wie in Abschnitt 2.4.2 beschrieben zugegriffen werden. Enthält die Klasse *A* zusätzlich Methoden, dann gehören diese ebenfalls zu dem durch *A* definierten Typ. Eine Variable *x* vom Typ *A* kennt dann auch Prozeduren und Funktionen, die ausgeführt werden können. Die Ausführung einer solchen Prozedur (Funktion) kann die Attribute von *x* ändern. Man spricht deshalb auch in Anlehnung an den in Abschnitt 2.2.3 definierten Zustandsbegriff vom *Zustand des Objekts x*. Für Referenzklassen können anonyme Objekte wie in Abschnitt 2.4.2 erzeugt werden.

Beispiel: Definition der Klasse *CONST*

(35)

```
class CONST is -- Referenzklasse
  value: INT;
  create_const(i:INT):CONST is return(#CONST{value := i}); end;
  eval: INT is return(value) end;
  parse(s:STRING):INT is ... end;
end;
```

Das Attribut ist *value*. Die Methoden sind *create_const*, *eval* und *parse*. Mit *#CONST{Initialisierung}* wird ein Objekt der Klasse *CONST* erzeugt und initialisiert. Sei *x* ein Objekt der Klasse *CONST*. Mit *x.eval* wird die Konstante *x* ausgewertet. Der Zustand von *x* ist der Wert der Komponente *value*.

Bemerkung:

In einer objektorientierten Sprache sollten *alle* Datentypen wie in Smalltalk einheitlich durch Klassen definiert sein. Konsequenterweise sind dann alle Klassen Referenzklassen, weshalb dieser Begriff nicht extra erwähnt zu werden braucht. Eiffel, Sather, C# und Fortran2003 teilen die Klassen in Wert- und Referenzklassen, um effizienter mit den elementaren (Wert-)Klassen *INT*, *FLOAT* usw. umgehen zu können. C++ und Java rechnen die einfachen Typen *byte*, *short*, *int*, ..., *float*, *double*, *bool* nicht den Klassen zu. C#, Fortran2003 und Sather kennen auch nicht-elementare Wertklassen mit Methoden. Auch die Feldklassen spielen oft eine Sonderrolle. Diese Unterschiede sind alle effizienz- oder historisch bedingt, können aber das Verständnis und die Implementierung erschweren, insbesondere bei generischen Klassen, siehe Abschnitt 2.6.4.

Bisher wurde eine Klasse als Typdefinition angesehen. Man kann eine Klasse auch als Moduldefinition verstehen: Die *Schnittstelle einer Klasse* besteht aus den Merkmalen samt deren Signaturen. Die Schnittstelle einer Klasse kann in den meisten Programmiersprachen auf bestimmte Merkmale eingeschränkt werden, indem manche Merkmale als

D₂

Interna einer Klasse vereinbart werden und damit nach außen nicht sichtbar sind, oder indem umgekehrt nur bestimmte Merkmale als öffentlich (*public*) ausgezeichnet werden. Eine *abstrakte Klasse* oder *Schnittstellenklasse* definiert die Signatur einer Klasse, die Implementierung aber nicht vollständig. Daher kann es keine Objekte einer abstrakten Klasse geben.

Der Begriff des *package* in Java erlaubt es, Klassen, Typen, Schnittstellenbeschreibungen, d.h. abstrakte Klassen, usw. in Module zu gruppieren. In C# stehen zu diesem Zweck *namespaces* zur Verfügung. Auch die Import-Anweisung aus Modulen zum unqualifizierten Zugriff auf Klassen und Typen aus anderen Modulen ist vorhanden. An sich gehören diese Möglichkeiten in eine *Konfigurationssprache*, mit der auch Bausteine aus verschiedenen Programmiersprachen verbunden werden können.

2.6.3 Vererbung

Unter *Vererbung* versteht man die Übernahme der Merkmale einer (*Ober-*)Klasse *A* in eine (*Unter-*)Klasse *B*. Die Unterklasse besitzt zusätzlich zu ihren eigenen Merkmalen auch alle Merkmale der Oberklasse. Dabei können aus der Oberklasse geerbte Merkmale, wie nachfolgend erklärt, durch Merkmale der Unterklasse überschrieben werden. Vererbung dient verschiedenen Zwecken (siehe Kapitel D4).

Aus Programmiersprachensicht benutzt man Vererbung einerseits zur Wiederverwendung des Codes der Oberklasse, andererseits, um verhaltensgleiche Objekte zu kennzeichnen, oder um die in Kapitel D4.3.4 beschriebene Beziehung „*B* is a *A*“ zu realisieren. Die im Entwurf benötigte Beziehung der Verhaltensgleichheit „(ein Objekt des Typs) *B* kann überall an Stelle (eines Objekts von) *A* benutzt werden“ läßt sich übersetzungstechnisch nicht vollständig prüfen. Sie wird zur Beziehung „*B* ist *konform* zu *A*“ abgeschwächt: *B* heißt *konform* zu *A*, wenn ein Objekt der Klasse *B* überall an Stelle eines Objekts der Klasse *A* verwendet werden kann, ohne daß dies zu einem Typfehler führt. Die Beziehung „*B* is a *A*“ bedeutet dagegen „*B* ist *spezieller* als *A*“: die Menge der Objekte von *B* ist eine Teilmenge der Objekte von *A*. Bei dieser Beziehung können Übersetzer nur noch dann Typsicherheit garantieren, wenn *A* eine abstrakte Klasse ist, die *überall* durch *B* ersetzt wird.

Technisch bedeutet Konformität: Zu jedem Merkmal *m* von *A* muß es ein gleichbenanntes Merkmal *m* von *B* geben mit:

- Wenn *m* in *A* ein Attribut vom Typ *T* ist, dann ist *m* in *B* ein Attribut vom Typ *T'* und $T = T'$.
- Wenn *m* in *A* eine Methode der Stelligkeit $T_1 \times \dots \times T_n$ ist, dann ist *m* in *B* eine Methode der Stelligkeit $U_1 \times \dots \times U_n$ und T_i ist für $i = 1, \dots, n$ Untertyp von U_i (U_i ist *kontravariant* zu T_i).
- Wenn *m* in *A* eine Methode der Stelligkeit $T_1 \times \dots \times T_n \rightarrow T$ ist, dann ist *m* in *B* eine Methode der Stelligkeit $U_1 \times \dots \times U_n \rightarrow U$, T_i ist für $i = 1, \dots, n$ Untertyp von U_i und U ist Untertyp von T (U ist *kovariant* zu T).

Dabei ist, wie in den meisten objektorientierten Sprachen, vorausgesetzt, daß die Argumente durch Wertaufwurf übergeben werden, siehe Abschnitt 2.5.2. Ist das Attribut *m* nur lesbar, z.B. eine Konstante, so läßt sich die erste Forderung $T = T'$ zu „*T'* ist *kovariant* zu *T*“ abschwächen.

Beispiel:

Smalltalk kennt keine Konformitätsforderung; wird sie jedoch nicht eingehalten, so kann dies zu Laufzeitfehlern führen. In Eiffel sind kovariante Argumenttypen von

Methoden erlaubt; erst der Binder prüft beim Zusammensetzen von Programmen, ob hierdurch Fehler auftreten können. C++ und Java fordern sogar mehr als die obigen Bedingungen: bei Methoden müssen die Argumenttypen gleich sein.

Die Schnittstelle einer Klasse A ist die Vereinigung ihrer Merkmale mit denen der geerbten Klassen. Erbt die Klasse B die Klasse A und definiert B ein Merkmal m , dessen Signatur konform zur Signatur eines Merkmals von A ist, dann *überschreibt* die Definition von m in B die Definition von m in A . Diese Definition ist ähnlich der Verdeckungsregel bei der Definition von Gültigkeitsbereichen in Abschnitt 2.3.2. Die Bindung eines Merkmals m ist nicht mehr eindeutig, wenn eine Klasse C zwei Klassen A und B erbt, die beide das Merkmal m definieren und dieses in C nicht überschrieben wird. *Smalltalk* umgeht dieses Problem, in dem es nur *Einfachvererbung* zuläßt. Programmiersprachen mit *Mehrfachvererbung* müssen definieren, wie solche Konflikte beseitigt werden. Daneben kann es verschiedene Definitionen von m geben, die nicht zueinander konform sind, z.B. weil sie unterschiedliche Parameteranzahl haben; man sagt dann, m sei *überladen*.

Beispiel:

Eiffel erfordert bei Mehrfachvererbung Umbenennung oder Ausblendung von Merkmalen der geerbten Klassen, so daß durch Mehrfachvererbung entstehende Konflikte vermieden werden. C++ erlaubt ebenfalls Umbenennung oder Ausblendung von Merkmalen. In C++ muß bei Verwendung eines nicht eindeutig zuordenbares Merkmals dessen Herkunft mit angegeben werden. Java und C# erlauben mehrfaches Erben nur von Schnittstellenklassen.

Sei A eine Oberklasse mit den Unterklassen B_1, B_2, \dots . In einer Variablenvereinbarung x : A wird dann in den meisten objektorientierten Sprachen A als die Typmenge $T = (A, B_1, B_2, \dots)$ angesehen. Ist A selbst eine abstrakte Klasse, so gehört sie nicht zur Typmenge T . Im Sinne von Abschnitt 2.4.2 ist der (polymorphe) Typ A die disjunkte Vereinigung der Typmenge T . x kann Objekte aller Typen aus T als Wert annehmen. Es sind jedoch nur die in A definierten Merkmale m für x definiert. Bei einem Attributzugriff $x.m$ oder einem Methodenaufruf $x.m(\dots)$ wird das zum aktuellen Objekt x gehörige Merkmal benutzt. Dies ist die in Kapitel D4.5 beschriebene *dynamische Bindung*. x heißt eine *polymorphe Variable*, die Zugriffe $x.m$ oder $x.m(\dots)$ heißen *polymorphe Zugriffe*. In C++, Java, C#, Sather und Fortran2003 ist automatisch garantiert, daß in den Unterklassen das Merkmal m mit den gewünschten Typeigenschaften existiert. In *Smalltalk* oder *Eiffel* ist dies nur sicher, wenn die ausgewählte Unterklasse B konform zur Oberklasse A ist.

Beispiel:

(36) zeigt die Klassen *AUSDRUCK*, *ADDOP*, *MULOP* und *BINOP*. Die Klassen *AUSDRUCK* und *BINOP* sind Schnittstellenklassen. Die Klasse *CONST* aus (35) wird erweitert, indem sie von *AUSDRUCK* erbt. Die Klassen *BINOP*, *ADDOP*, *MULOP* und *CONST* sind konform zur Klasse *AUSDRUCK*, die Klassen *ADDOP* und *MULOP* sind Spezialisierungen der Klasse *BINOP*. Die Definition von *operator* in den Klassen *MULOP* und *ADDOP* überschreibt die in der Klasse *BINOP*. Ähnlich legen die Definition von *eval* und *parse* in den Klassen *CONST*, *ADDOP* und *MULOP* die Implementierung dieser Funktionen fest. Damit haben diese drei Klassen die gleiche, durch die Klasse *AUSDRUCK* definierte abstrakte Schnittstelle.

Vererbung und Polymorphie

(36)

```
abstract class AUSDRUCK is
  parse(s:STRING):AUSDRUCK is abstract;
  eval: INT is abstract;
end;
```

D₂

```

abstract class BINOP is
  inherit AUSDRUCK
  left,right: AUSDRUCK
  operator: STRING;
end;
class MULOP is
  inherit BINOP
  operator:STRING := "**";
  eval: INT is return(left.eval * right.eval); end;
  parse(s:STRING):MULOP is ... end;
end;
class ADDOP is
  inherit BINOP
  operator: STRING := "+";
  eval: INT is return(left.eval + right.eval); end;
  parse(s: STRING): ADDOP is ... end;
end;

```

2.6.4 Generizität

Ähnlich wie abstrakte Datentypen in Abschnitt 2.1.1 parametrisiert werden können, kann man Module und Klassen parametrisieren. Solche Module (Klassen) $A(T_1, \dots, T_n)$ heißen *generisch*. Sie dürfen in der Schnittstelle und der Implementierung die Parameter T_1, \dots, T_n verwenden. Üblicherweise stehen die generischen Parameter für Typen. Eine Instanz $A(U_1, \dots, U_n)$ eines generischen Moduls (generische Klasse) definiert ein Modul (Klasse), wobei in der Definition von $A(T_1, \dots, T_n)$ die generischen Parameter T_1, \dots, T_n textuell durch die Typen U_1, \dots, U_n ersetzt werden.

Beispiel: Generische Klassen

(37)

```

abstract class AUSDRUCK(T) is
  parse(s:STRING):$AUSDRUCK(T) is deferred;
  eval:T is deferred;
end;
abstract class BINOP(T) is
  inherit AUSDRUCK(T)
  left,right:$AUSDRUCK(T)
  operator:STRING;
end;
class MULOP(T) is
  inherit BINOP(T)
  operator:STRING := "**";
  eval:T is return(left.eval * right.eval); end;
  parse(s:STRING):MULOP is ... end;
end;
class ADDOP(T) is
  inherit BINOP(T)
  operator:STRING := "+";
  eval:T is return(left.eval + right.eval); end;
  parse(s:STRING):ADDOP is ... end;
end;
class CONST(T) is
  value:T;
  create_const(i:T):CONST is return(#CONST{value := i}); end;
  eval:T is return(value) end;
  parse(s:STRING):CONST(T) is ... end;
end;

```

Die Klasse *AUSDRUCK* in (36) behandelt nur ganzzahlige Ausdrücke. Eine Klasse *FLTAUSDRUCK*, die Gleitpunktausdrücke erlaubt, sieht ebenso aus, wenn man überall *INT* durch *FLOAT* ersetzt. Die generischen Klassen in (37) liefern mit der Instantiierung *AUSDRUCK(INT)*, *BINOP(INT)*, *ADDOP(INT)*, *MULOP(INT)* und *CONST(INT)* die Klassen aus (35) und (36) zurück; jetzt ist aber auch *AUSDRUCK(FLOAT)* erlaubt und behandelt Gleitpunktausdrücke.

Nicht alle modularen und objektorientierten Programmiersprachen erlauben die Definition generischer Module oder Klassen (z.B. Modula, Fortran2003 sowie die älteren Versionen von Java und C#). In C++ spricht man von *Schablonen (templates)*. In Ada können auch Prozeduren generisch sein, neben Typen sind auch Objekte, Pakete und Prozeduren als generische Parameter und Argumente zugelassen.

Die in der Praxis wichtigsten generischen Klassen sind die sogenannten *Behälterklassen*, die den Behältertypen aus Abschnitt 2.1.2 entsprechen und Datenstrukturen *B* mit Elementen vom Typ *T* definieren, wobei *T* weitgehend unabhängig von *B* gewählt werden kann. Das einfachste Beispiel von Behälterklassen sind Felder *array(T)* mit Elementen vom Typ *T*.

Wenn in einer Behälterklasse Methoden *m* definiert sind, die spezielle Eigenschaften des Elementtyps voraussetzen (z.B. daß auf dessen Werten eine Ordnung definiert ist), so beschränkt dies die Menge der zulässigen Elementtypen. Man spricht von *beschränkter Generizität*. Haskell erlaubt solche Beschränkungen durch Typklassen. In Java lassen sich die Parameter generischer Klassen durch Schnittstellenklassen beschränken, in C# zusätzlich noch durch Klassen.

Beispiel:

Die generischen Klassen *AUSDRUCK(T)*, *BINOP(T)*, *ADDOP(T)* und *MULOP(T)* erwarten, daß auf den Instanzen für *T* ein Operator *+* und *** definiert ist. Für *T* lassen sich also nicht alle Typen einsetzen.

In C# wie in den meisten anderen objektorientierten Programmiersprachen mit Generizität definiert jede Instanz einer generischen Klasse einen eigenen Typ. In Java hingegen definiert eine generische Klasse *A<T>* auch einen Typ *A*, an den zugewiesen werden darf. Bei der Erzeugung von Bytecode werden alle Methodenparametertypen *T* durch die Typschranke (im Extremfall *Object*) ersetzt. Für Behälterklassen *A<T>* mit Elementen vom Typ *T* kann beispielsweise nicht mehr garantiert werden, daß tatsächlich ein Objekt einer Behälterklasse ausschließlich Elemente vom Typ *T* enthält.

Beispiel:

Betrachtet man die generische Klasse *List<T>* in Java, so enthält diese beispielsweise eine Methode *insert(T x)*. Dann ist folgendes Programmfragment möglich:

```
List l = new List<int>;
List<String> m;
...
l.insert(2);
l.insert("a");
m=l;
```

Sowohl die Anweisungen *l.insert("a")* als auch die Zuweisung *m = l* sind in Java erlaubt. Wie man sieht, wird die Zeichenkette "a" in ein Objekt vom Typ *List<int>* eingefügt. Nach der Zuweisung *m = l* enthält die Liste *m* sowohl Zeichenketten als auch ganze Zahlen, obwohl *m* vom Typ *List<String>* ist. Der Java-Übersetzer meldet an solchen Stellen nur eine Warnung, die zudem noch ausgeschaltet werden kann. Grund für diese Entscheidung ist die Kompatibilität mit Altcode. In C# wie in den meisten anderen objektorientierten

Sprachen mit Generizität hingegen gibt es keinen Typ `List`; `l` müßte mit `List<int>` deklariert werden. Damit werden dann die Anweisungen `l.insert("a")` und `m = l` illegal.

Generizität ist, wie man hieran sieht, auch ein Konzept zur Wiederverwendung von Programmen. In funktionalen Sprachen können auch Funktionsdefinitionen parametrisiert sein, was (38) zeigt.

Beispiel: Parametrisierte Funktion in Haskell, die Typen T , U und V sind beliebig (38)

```
compose :: (T -> U) -> (U -> V) -> V
compose f g = \x. f(g x)
```

Erst beim Aufruf einer parametrisierten Funktion wird bestimmt, mit welchen (parametrisierten) Typen die Parameter instantiiert werden. Man sagt dann auch, daß der Typ bzw. die Funktion *polymorph* ist. Diese Art von Polymorphie heißt auch *parametrische Polymorphie*. Die meisten stark typisierten funktionalen Sprachen kennen parametrische Polymorphie. Die vorkommenden Instanzen generischer Parameter können mit *Typinferenz* durch den Übersetzer bestimmt werden.

2.7 Skriptsprachen

Mit *Skriptsprachen* fügt man Programmkomponenten zu größeren Programmen auf eine andere Weise als beim modularen oder objektorientierten Programmieren zusammen: Die Komponenten sind selbständige Programme P_1, P_2, \dots , die in unterschiedlichen Sprachen geschrieben sein können oder sogar nur übersetzt vorliegen; ein Skriptprogramm S faßt die Programme P_i als Grundoperationen auf, deren Operanden (Eingaben) die Ergebnisse (Ausgaben) anderer solcher Operationen sind. S fügt diese Operationen zusammen, indem es die Übergabe der Daten von einer zur anderen Operation übernimmt. Dazwischen könnten die Daten konvertiert werden, um sie in eine für die nächste Operation geeignete Form zu bringen. Skriptsprachen verwendet man also vor allem, um ein neues Programm aus vorhandenen Programmen zusammenzusetzen; das Skript liefert den Kleister zwischen den vorgegebenen Programmen.

Skriptsprachen haben die typischen Eigenschaften imperativer Sprachen: sie verfügen über Variablen, an die man zuweisen kann, bedingte Anweisungen, Schleifen und Prozeduren; andere Skriptprogramme können als Unterprogramme, teilweise sogar mit paralleler Ausführung, aufgerufen werden.

Voraussetzung für das Zusammenwirken von Skriptsprache und Programmkomponenten sind Datentypen, die von allen beteiligten Programmen verstanden werden. Meist sind dies Zeichenketten (*Perl, Tcl/Tk, Python*) oder Bytefolgen (*shell script*) unbeschränkter Länge.

Beispiel:

Das bekannteste Beispiel einer Skriptsprache sind die verschiedenen Versionen von Kommandosprachen unter dem Betriebssystem Unix, die *shell scripts*. In der Bourne shell (und ähnlich der *bash* unter Linux) werden Daten zwischen Programmen pufferweise über eine Röhre (*pipe*) übergeben, indem man sie auf die Standardausgabe schreibt und diese vom nächsten Programm als Standardeingabe lesen läßt; die beiden beteiligten Programme P_1 und P_2 laufen dabei als Coroutinen, symbolisch dargestellt mit „|“ als Verknüpfung: $P_1 | P_2$. Stattdessen könnten sie auch nacheinander laufen, wenn man die Aus- bzw. Eingabe in eine Datei umlenkt: $P_1 > \text{dateiname}$; $P_2 < \text{dateiname}$. Die Ein-/Ausgaben sind dabei Bytefolgen, die man eventuell auch als Zeichenketten schreiben kann. Da es nur diesen einen Datentyp gibt, vereinbart man Variablen

durch die Zuweisung `shellvariable = Text_als_Wert` ohne Typ und kann dann mit der Schreibweise `$shellvariable` darauf zugreifen.

Ebenso wie shell scripts sind auch alle anderen Skriptsprachen auf die sofortige Ausführung von Operationen oder Programmen *S* eingerichtet und werden daher meist interpretiert. Skriptsprachen sind also charakterisiert durch

- interpretative Ausführung oder rudimentäre Übersetzung
- Typfreiheit bzw. nur wenige Datentypen
- Einbettung von Betriebssystemkommandos
- komfortable Operationen auf Datentypen
- Fremde Programme können aufgerufen werden (Importieren von Funktionalität)

Daneben haben Skriptsprachen klassische Steuerstrukturen wie Schleifen und bedingte Anweisungen, Prozeduraufrufe. Perl und Python unterstützen Modularität und Objekt-orientierung.

Historisch haben Skriptsprachen zwei Wurzeln: Kommandozeilen-Interpreter und ihren Gebrauch zur Steuerung des Stapelbetriebs, wie oben am Beispiel der *Unix shell* dargestellt, und Editoren wie dem *sed* unter Unix, mit denen man Texte manipulieren kann, indem man Teilzeichenketten, die durch reguläre Ausdrücke spezifiziert sind, sucht und ersetzt. *awk* unter Unix ist ein komfortablerer Interpreter für solche Ersetzungsaufgaben.

Perl. Die Skriptsprache *Perl* bietet die Möglichkeiten von shell script in erweiterter Fassung und ist zudem plattformunabhängig. Programme werden über Zeichenketten verknüpft. Perl bietet eine Reihe komplexer Operationen auf Zeichenketten, die von anderen Skriptsprachen übernommen wurden:

- Finden/Ersetzen von Teilzeichenketten, die auf reguläre Ausdrücke passen;
- Zuweisungsoperatoren für reguläre Muster;
- Übersetzung von Zeichen gemäß einer Tabelle.

Perl bietet weitere Datentypen, von denen *assoziative Felder* hervorzuheben sind. Diese sind eine Menge von Paaren (Schlüssel, Zeichenkette). Auf die Zeichenketten eines assoziativen Feldes kann über ihre Schlüssel zugegriffen werden. Assoziative Felder können vereinigt werden. Außerdem hat Perl einen einfachen Modulmechanismus. Nebenläufige Ausführung der verknüpften Programme ist möglich (mit *fork*). Außerdem stehen aus UNIX bekannte Mechanismen zur Verknüpfung von Programmen zur Verfügung, z.B. Pipes.

Beispiel: (39)

Das folgende Perl-Skript benutzt assoziative Felder. In der ersten Zeile wird ein assoziatives Feld durch Angabe der Schlüssel-Wert-Paare definiert. Mit der zweiten Zeile wird eine Zahl eingelesen. Der Operator *chop* entfernt das letzte Zeichen einer Zeichenkette; im Beispiel das Zeichen für eine neue Zeile. Bei der Auswertung des Ausdrucks in der bedingten Anweisung wird die Zeichenkette `$x` automatisch in eine Zahl konvertiert. Die Ausgabe in der fünften Zeile greift dann auf den Wert eines assoziativen Feldes über den Schlüssel zu.

```

%map = ('1','Maier','2','Müller','3','Schmidt');
print "Eingabe einer Zahl (1, 2 oder 3):";
chop($x = <STDIN>);
if (1 <= $x && $x <=3){
    print "Name für $x ist $map{$x}\n"
} else {
    print "falsche Eingabe\n" }

```

Tcl/Tk. Die Skriptsprache *Tcl/Tk (tool command language with widget toolkit)*, entwickelt von Jan Ousterhout, dient dazu, existierende Programme untereinander und mit grafischen Benutzerschnittstellen zu verknüpfen. Programme kommunizieren untereinander mittels Zeichenketten. Tcl kennt Prozeduren und Funktionen, Parameter werden durch Namensaufruf übergeben. Wegen der Verknüpfung mit graphischen Benutzerschnittstellen können mit Tcl/Tk verknüpfte Programme nebenläufig ausgeführt werden. Die Ereignisbehandlung ist primitiv und analog zur Ausnahmebehandlung.

Beispiel:

(40)

Das folgende Tcl/Tk-Skript zeigt, wie Fenster und Kommandos miteinander verknüpft werden. Das Skript besteht aus einem Tk-Teil, der das Aussehen eines Fensters definiert, und aus einem Tcl-Teil (Prozedur *Print*). Beide Teile definieren die Interaktion zwischen dem Fenster und dem Tcl-Teil. Das Tk-Kommando *button* erzeugt einen Knopf und wird mit einem Tcl-Kommando verknüpft, das beim Drücken dieses Knopfes ausgeführt wird. Dieses Kommando kann auch wie im Beispiel eine Tcl-Prozedur sein. Das Tk-Kommando *entry* definiert im Fenster ein Eingabefeld mit dem Namen *.f.digit*. Im Rumpf der Prozedur *Print* kann auf dieses Feld zugegriffen werden. Bei der Ausführung von *.f.digit get* wird der aktuelle Wert des Feldes (als Zeichenfeld!) gelesen. Das Tk-Kommando *text* definiert im Fenster ein Ausgabefeld mit dem Namen *.output*. Bei der Ausführung von *.output insert end* wird das entsprechende Argument ans Ende des Ausgabefeldes geschrieben.

```

wm title . "Auswahl"
wm minsize . 100 100
frame .buttons -borderwidth 15
pack .buttons -side top -fill x
button .buttons.quit -text Schließen -command exit
button .buttons.input -text Eingabe -command Print
pack .buttons.quit .buttons.input -side right
frame .f;
pack .f -side top
label .f.l -text "Eingabe einer Zahl (1, 2 oder 3)"
entry .f.digit -width 5 -relief sunken
pack .f.l .f.digit -side left
text .output -width 50 -height 20 -bd 5 -relief raised
pack .output -side top

proc Print {} {
    set list {Maier Müller Schmidt}
    set c [.f.digit get]
    if {$c >= 1 && $c <= 3} {
        .output insert end [lindex $list [expr $c - 1]]
        .output insert end "\n"
    } else {
        .output insert end "falsche Eingabe\n"
    }
}

```

Python. Die Skriptsprache *Python*, 1991 entwickelt von G. van Rossum in den Niederlanden, ist objektorientiert und bietet alle Möglichkeiten von Perl, vermeidet aber Perls häufig kryptische Schreibweisen, indem sie sich syntaktisch und in einigen semantischen

Eigenschaften an Modula-3 anlehnt. Zusätzlich gibt es mehrere verschiedene Datentypen, darunter auch flexible und assoziative Felder. Die Gruppierung von Anweisungen, z.B. zu einem Schleifen- oder Prozedurrumpf, wird in Python nicht durch Klammerung, sondern durch Einrückung der Zeilen gekennzeichnet; Zeilenwechsel und Einrücken haben also syntaktische Bedeutung. Wie Perl bietet auch Python eine Schnittstelle zu Tk, um graphische Benutzerschnittstellen zu programmieren. Die Objektorientierung begünstigt auch den Anschluß anderer Bibliotheken.

Beispiel: (41)

Das folgende Python-Skript hat dieselbe Semantik wie das Tcl/Tk-Skript in (40). Man sieht, daß Python eine objektorientierte Sprache ist. Die Klassen *App*, *Button*, *Frame*, *Label* usw. werden durch das Paket *Tkinter* importiert. Konstruktoren ruft man wie in Java mit dem Klassennamen auf, z.B. in der Klasse *App* der Aufruf *f = Frame(master)*. Die Methode *out* hat wie das Perl-Skript in (39) ein assoziatives Feld *map*, das ein Objekt der (vordefinierten) Klasse *Mappings* ist.

```
from Tkinter import *
class App:
    def __init__(self, master):

        buttons = Frame(master)
        buttons.pack(side=TOP)
        self.quit = Button(buttons, text="Schließen", fg="red", command=buttons.quit)
        self.quit.pack(side=RIGHT)

        self.input = Button(buttons, text="Eingabe", command=self.out)
        self.input.pack(side=RIGHT)

        f = Frame(master)
        f.pack(side=TOP)
        self.l = Label(text="Eingabe einer Zahl (1, 2 oder 3)")
        self.l.pack(side=TOP)
        self.digit=Entry(f, width=5, relief="sunken")
        self.digit.pack(side=LEFT)
        self.output = Text(master, width=50, height=20, bd=5, relief="raised")
        self.output.pack(side=BOTTOM)

    def out(self):
        map = { "1":"Maier", "2":"Müller", "3":"Schmidt"}
        x = self.digit.get()
        if map.has_key(x) : self.output.insert(END, map[x] + "\n")
        else: self.output.insert(END, "Falsche Eingabe\n")

root = Tk()
app = App(root)
root.mainloop()
```



PHP. PHP wurde als Skriptsprache für die Erstellung dynamischer Webseiten entwickelt (das Akronym PHP/FI stand ursprünglich für *personal homepage/forms interpreter* und steht heute für *hypertext preprocessor*). PHP war zunächst eine Weiterentwicklung von Perl, die mittlerweile um objektorientierte Konzepte wie Klassen und Vererbung erweitert wurde (PHP 5). Diese objektorientierten Erweiterungen sind syntaktisch an Java 1.4 orientiert, allerdings ohne die starke Typisierung beizubehalten. Für weitere Details verweisen wir auf Kapitel E10.

Abgesehen von shell scripts werden alle Skriptsprachen mindestens teilweise übersetzt; der Interpretierer sieht einen attribuierten Syntaxbaum. Wegen der intensiven Nutzung

vor allem von Perl im Internet kann man einige Skriptsprachen zur Effizienzsteigerung inzwischen auch vollständig übersetzen.

2.8 Parallelität

Die Sprachelemente aus Abschnitt 2.4 und Abschnitt 2.5 erlauben nur die Formulierung sequentieller Programme. Für die Formulierung paralleler Programme sei auf das Kapitel D7 verwiesen. Die Strukturierungskonzepte aus Abschnitt 2.6 sind mit den hier vorgestellten Konzepten kombinierbar.

2.9 Historische Entwicklung von Programmiersprachen

Die erste imperative Programmiersprache, die das Prinzip des strukturierten Programmierens unterstützte, war der *Plankalkül* von Konrad Zuse aus dem Jahr 1944, der allerdings erst 1972 veröffentlicht wurde, siehe [Zuse 72]. Anfangs der fünfziger Jahre entstanden unter Namen wie *Autocode* oder *Formelcode* erste höhere Programmiersprachen, die über arithmetische Ausdrücke und Zuweisungen, bedingte Sprünge, Zählschleifen und Prozeduren, noch meist ohne Parameter, verfügten. Diese Sprachen gingen dann in der ersten Fassung der Sprache *Fortran* (*FORmula TRANslator*) von 1954 auf. Unter Mitarbeit führender Fortran-Entwickler entstanden ab 1958 die verschiedenen Fassungen der *ALGOrithmic Language* Algol: *Algol 58*, *Algol 60* und *Algol 68*. In Amerika hatten die Algol 58-Dialekte *Neliac* und *Jovial* jahrzehntelang Verbreitung. In Europa war in den sechziger Jahren neben Fortran vor allem Algol 60 zur Formulierung numerischer Aufgaben verbreitet. Algol 60 war anfangs die einzige Programmiersprache, in der die *Association for Computing Machinery (ACM)* Algorithmen veröffentlichte. Der Algol-60-Bericht [Naur 63] gilt auch heute noch als klassisches Musterbeispiel eines präzisen, aber in natürlicher Sprache geschriebenen Sprachberichts.

Aus der Arbeit [McCarthy 60] entstand die Programmiersprache *Lisp* [McCarthy 81, McCarthy 85], aus der die heutigen funktionalen Programmiersprachen hervorgingen. Lisp und der zugrundeliegende λ -Kalkül hatten aber auch großen Einfluß auf Algol 68, auf weitere imperative Sprachen und insbesondere auf deren Semantik.

Um die gleiche Zeit entstand die auch heute noch am weitesten verbreitete Programmiersprache *COBOL* (*COmmon Business Oriented Language*). Es wird oft übersehen, daß die in COBOL vorgesehene Dezimalarithmetik (BCD-Arithmetik) wegen der vorgeschriebenen dezimalen Rundung für Geldberechnungen unabdingbar ist und nicht durch Rechnen im Binärsystem ersetzt werden kann. Insbesondere die Anbindung von Datenbanken an Programmiersprachen, die in Form der „Sprachen der vierten Generation“ (*4GLs*, *4th Generation Languages*) später zu einer eigenständigen Form von Programmiersprachen führte, verdankt ihren Ursprung der Sprache COBOL. Die auch heute noch verbreitete Programmiersprache *PL/I*, die 1965 von Mitarbeitern der Firma IBM entwickelt wurde, war der Versuch, die Stärken von COBOL mit denen der Fortran/Algol-Entwicklung zu kombinieren. Ihre formale Beschreibung mit *VDL* (*Vienna Definition Language*) bildete den Ausgangspunkt der Spezifikationsmethode *VDM* (*Vienna Definition Method*).

Auch die zuerst in *Pascal* [Wirth 71] auftauchenden Verbunde (siehe Abschnitt 2.4.2) und die Fallunterscheidung gehen eigentlich auf COBOL zurück. Wirth beschrieb Pascal zu Recht als die auf das Wesentliche reduzierte, verallgemeinerte Fassung von Algol 60. Pascal bildete nicht nur die Grundlage der Methodik des strukturierten Programmierens, sondern hat bis in die syntaktische Formulierung hinein zahlreiche Programmiersprachen bis

heute beeinflusst. Die interpretative Implementierung von Pascal mit Hilfe des *P-Code*s war der Ausgangspunkt für die Entwicklung von *Turbo-Pascal* und Vorbild für die Bytecode-Implementierungen von Smalltalk und Java.

In der zweiten Hälfte der sechziger Jahre entwickelte Christopher Strachey, Professor in Oxford, die experimentelle Sprache *CPL* (*Christopher's Personal Language*). Auf sie gehen die Sprachen *BCPL*, eine der ersten maschinenunabhängigen Systemimplementierungssprachen und *C*, und damit in der Folge auch *C++*, *Java* und *C#* zurück. Insbesondere das Rechnen mit *left hand values*, also mit Zeigern, geht auf *CPL* zurück. Seinen Erfolg verdankt *C* nicht nur dem Umstand, daß ab 1971 das Betriebssystem UNIX und seine Dienstprogramme in *C* geschrieben wurden, sondern vor allem der Tatsache, daß *C* es allen Programmierern, die aus Maschinensprachen oder COBOL in die Systemprogrammierung mit „höheren“ Programmiersprachen umstiegen, erlaubte, ihre bisherige Berufserfahrung in einfacher Weise weiterzuverwerten, während in Pascal und seinen Nachfolgern ein rigoreses Umdenken gefordert wurde.

Mitte der siebziger Jahre entwickelte Niklaus Wirth die Sprache *Modula* als Grundform modularer Programmiersprachen, auf der dann die Sprachen *Ada* und *Modula-3* aufbauten [Cardelli 92]. Die von Jean Ichbiah entwickelte Sprache *Ada* verdankt im übrigen viel der zeitgleich mit *Modula* und ebenfalls von Ichbiah entworfenen Sprache *LIS* (Langage d'Implementation des Systèmes) [Ichbiah 74]. Zu den modularen Programmiersprachen gehört auch die Sprache *CLU* von Barbara Liskov [Liskov 77, Liskov 79], die zwar keine weite Verbreitung fand, deren Implementierer aber danach und aufbauend auf Ideen von *CLU* die Architektur des *X-Window*-Systems entwarfen.

Die Entwicklung objektorientierter Programmiersprachen beginnt mit dem Entwurf von *SIMULA 67* durch Ole-J. Dahl, B. Myrhaug und K. Nygaard in Oslo in den Jahren 1965/67, siehe [Dahl 68]. Diese geniale Programmiersprache ist eine Erweiterung von *Algol 60* zur Lösung ereignisgesteuerter Simulationsaufgaben. Sie nahm nahezu alle Ideen der nächsten beiden Jahrzehnte über modulare und objektorientierte Programmstrukturen vorweg, wengleich zunächst unklar blieb, warum diese Ideen so erfolgreich werden könnten. Die Entwicklung von *Smalltalk* 1974/80 durch Adele Goldberg und ihre Mitarbeiter in XEROX PARC baut auf Ideen von Alan Kay und *Simula 67* auf. Sie erklärt implizit einen Teil der Ideen, indem sie sie auf das Entwurfsprinzip *Kooperation von Objekten* (im Gegensatz zur hierarchischen Dekomposition von Problemen) zurückführt; damit wird im übrigen klar, daß Objektorientierung vor allem eine Entwurfsmethodik ist und die üblicherweise als wesentlich genannten Begriffe Klassen, Vererbung und Polymorphie (zusammen mit Generizität) nur Implementierungshilfsmittel für dieses Entwurfsprinzip sowie für den Entwurf wiederverwendbarer Software sind. *C++*, entworfen von Bjarne Stroustrup [Stroustrup 98], ist bis hin zum Begriff der virtuellen Methoden eine Adaption der Begriffswelt von *Simula 67* an die Begriffswelt von *C*; lediglich die Methodik des Überladens von Methoden und die (nachträglich eingefügten) Schablonen zur Beschreibung generischer Klassen bieten wesentlich Neues. *Java* vereinfacht diese Begriffswelt, um Typsicherheit zu erreichen. Während Sprachen wie *Smalltalk*, *C++*, *Java* und *Eiffel* ausschließlich Referenzklassen kennen, erweitert *C#* die Sprache *Java* um Werteklassen, ohne die Eigenschaft der Typsicherheit zu verletzen. Die Sprache *Eiffel* von Bertrand Meyer ist eine Reformulierung der objektorientierten Ideen mit den Begrifflichkeiten von *Pascal/Modula/Ada*; auch hier tritt das Entwurfsprinzip *Kooperation von Objekten* in Form des *vertraglichen Entwurfs* (*design by contract*) klar hervor. Viele der objektorientierten Entwurfsideen, auf denen Vererbung beruht, wie zum Beispiel die Relation „*A* is a *B*“, finden sich bereits in [Minsky 75] und zeigen die nahe Verwandtschaft der objektorientierten Ideenwelt zu Begriffen aus dem Bereich der künstlichen Intelligenz. Der dort eingeführte Begriff der Ähnlichkeit von *frames* hat im objektorientierten Bereich noch

keine passende Anwendung gefunden. Die meisten heute in der Industrie gebräuchlichen Programmiersprachen sind standardisiert. Die Sprachen Java und C# sind prominente Ausnahmen. Die Zukunft wird zeigen, inwieweit sich auch diese Sprachen einer Standardisierung unterziehen werden. Neuere Entwicklungen im Bereich der Programmiersprachen reduzieren Sprachkonzepte auf ihren Kern und lagern viele Konzepte in Bibliotheken aus.

Die historische Entwicklung und viele Einzelheiten über Programmiersprachen findet man in den Büchern [Sammet 69, Wexelblat 81, Bergin 96]. Das Buch [Horowitz 87] enthält den Nachdruck zahlreicher interessanter Artikel, darunter des Algol-60-Berichts.

Allgemeine Literatur

- Abelson, H.; Sussman, G. J.: Structure and interpretation of computer programs. 2nd ed. Cambridge, Mass.: MIT Press 1997
- Aho, A. V.; Sethi, R.; Ullman, J. D.: Compilers: principles, techniques, and tools. Reading, Mass.: Addison-Wesley 1996
- Goos, G.: Vorlesungen über Informatik. Band 1: Grundlagen und funktionales Programmieren. 2. Aufl. Berlin: Springer 1997
- Goos, G.: Vorlesungen über Informatik. Band 2: Objektorientiertes Programmieren und Algorithmen. 2. Aufl. Berlin: Springer 1999
- Louden, K. C.: Programming languages – principles and practice. Boston: PWS-Kent Publishing Company 1993
- Nielson, H. R.; Nielson, F.: Semantics with applications – a formal introduction. New York: Wiley 1992
- Sethi, R.: Programming languages – concepts and constructs. 2nd ed. Reading, Mass.: Addison-Wesley 1989
- Stansifer, R.: Theorie und Entwicklung von Programmiersprachen. München: Prentice-Hall 1995
- Tennent, R. D.: Principles of programming languages. Englewood Cliffs: Prentice-Hall 1981
- Waite, W. M.; Goos, G.: Compiler Construction. Heidelberg: Springer 1985

Spezielle Literatur

- [Ada 95] Ada 95 Reference Manual. International Standard ANSI/ISO/IEC-8652: 1995
- [Bergin 96] Bergin, T. J.; Gibson, R. G.: History of programmig languages. Reading, Mass.: Addison-Wesley 1996
- [C 99] C, International Standard ISO/IEC 9899:1999
- [C# 01] C# Language Reference Manual: <http://msdn.microsoft.com/library/default.asp>, 2001
- [C# 05] C# Language Specification, Standard ECMA 334, 2005
- [C++ 98] C++, International Standard ISO/IEC 14882:1998
- [Cardelli 92] Cardelli, L.; Donahue, J.; Glassman, L.; Jordan, M.; Kalsow, B.; Nelson, G.: Modula-3 language definition. ACM Sigplan Notices 27 (1992) 15–42
- [Cobol 85] American National Standard for information systems – programming language – COBOL / Secretariat, Computer and Business Equipment Manufacturers Association. New York: ANSI X3.23-1985, ISO 1989–1985
- [Dahl 68] Dahl, O.-J.; Myrhaug, B.; Nygaard, K.: Simula 67 – common base language. Oslo: norwegisches Rechenzentrum 1968
- [Dijkstra 76] Dijkstra, E. W.: A discipline of programming. Englewood Cliffs, N.J.: Prentice Hall 1976
- [Fortran 77] Fortran. ANSI X3.9-1978
- [Fortran 90] Fortran 90. ISO/IEC 1539: 1991 (E) und ANSI X3.198-1992

- [Fortran 95] Fortran: Base Language, International Standard ISO/IEC 1539-1:1997
- [Fortran 04] Fortran: Base Language, International Standard ISO/IEC 1539-1:2004
- [Goldberg 85] Goldberg, A.; Robson, D.: Smalltalk-80: The language and its implementation. Reading, Mass.: Addison-Wesley 1985
- [Gosling 96] Gosling, J.; Joy, B.; Steele, G.: The Java language specification. Reading, Mass.: Addison-Wesley 1996
- [Gosling 05] Gosling, J.; Joy, B.; Steele, G.; Bracha, G.: The Java Language Specification, Third Edition. Reading, Mass.: Addison-Wesley 2005
- [Griswold 71] Griswold, R. E.; Poage, J. F.; Polonsky, I. P.: The Snobol4 programming language. 2nd ed. Englewood Cliffs, N.J.: Prentice-Hall 1971
- [Hoare 69] Hoare, C. A. R.: An axiomatic basis for computer programming. Communications of the ACM 12 (1969) 576–580, 583
- [Hoare 73] Hoare, C. A. R.; Wirth, N.: An axiomatic definition of the programming language Pascal. Acta Informatica 2 (1973) 335–355
- [Horowitz 87] Horowitz, E. (Hrsg.): Programming languages: a grand tour. 3rd ed. Rockville, Md: Computer Science Press 1987
- [Ichbiah 74] Ichbiah, J. D.; Rissen, J. P.; Heliard, J. C.; Cousout, P.: The system implementation language LIS, reference manual. CII Honeywell-Bull, Louveciennes, France, Technical Report 4549 E/EN, 1974
- [Jensen 74] Jensen, K.; Wirth, N.: Pascal user manual and report. New York: Springer 1974
- [Knuth 74] Knuth, D. E.: Structured programming with go to statements. ACM Computing Surveys 6 (1974) 261–301
- [Liskov 77] Liskov, B.; Snyder, A.; Atkinson, R.; Schaffert, J. C.: Abstraction mechanisms in CLU. Communications of the ACM 20 (1977) 564–576
- [Liskov 79] Liskov, B.: Exception handling in CLU. IEEE Transactions on Software Engineering (1979) 545–558
- [Manna 91] Manna, Z.; Pnueli, A.: The temporal logic of reactive and concurrent systems: specification. New York: Springer 1991
- [McCarthy 60] McCarthy, J.: Recursive functions of symbolic expressions and their computation by machine. Part I. Communications of the ACM 3 (1960) 184–195
- [McCarthy 81] McCarthy, J.: History of Lisp. In: [Wexelblatt 81] 1981
- [McCarthy 85] McCarthy, J.; Abrahams, P. W.; Edwards, D. J.; Hart, T. P.; Levin, M. J.: Lisp 1.5 programmer's manual. 2nd ed. Cambridge, Mass.: MIT Press 1985
- [Meyer 92] Meyer, B.: Eiffel: the language. Englewood Cliffs, N.J.: Prentice-Hall 1992
- [Milner 90] Milner, R.; Tofte, M.; Harper, R. E.: The definition of standard ML. Cambridge, Mass.: MIT Press 1990
- [Modula-2 96] Modula-2, Base Language, International Standard ISO/IEC 10514-1:1990
- [Murer 96] Murer, S.; Omohundro, S.; Stoutamire, D.; Szyperski, C.: Iteration abstractions in Sather. ACM Transactions on Programming Languages and Systems 18 (1996) 1–15
- [Naur 63] Naur, P.: Revised report on the algorithmic language Algol 60. Communications of the ACM 6 (1963) 1–17
- [Ousterhout 97] Ousterhout, J. K.: Tcl and the Tk Toolkit. Reading, Mass.: Addison-Wesley 1997
- [Parnas 94] Parnas, D. L.; Madey, J.; Iglewski, M.: Precise documentation of well-structured programs IEEE Transactions on Software Engineering 20/12 (1994) 948–976
- [Pascal 90] Pascal, International Standard ISO/IEC 7185:1990
- [Pearl 80] Programmiersprache Pearl. DIN 66253, 1980
- [PHP] PHP Manual (deutsch), <<http://www.php.net/manual/de/>><http://www.php.net/manual/de/>
- [Sammet 69] Sammet, J.: Programming languages – history and fundamentals. Englewood Cliffs, N. J.: Prentice-Hall 1969

- [Stroustrup 98] Stroustrup, B.: The C++ programming language. 3rd ed. Reading, Mass.: Addison-Wesley 1998
- [van Rossum 99] van Rossum, G.: Python reference manual. Corporation for National Research Initiatives (CNRI) (URL: <http://www.python.org/doc/current/ref/ref.html>) 1999
- [van Wijngaarden 75] van Wijngaarden, A.; Mailloux, B. J.; Peck, J. E.; Koster, C. H. A.; Sintzoff, M.; Lindsey, C. H.; Meertens, L. G. L. T.; Fisker, R. G.: Revised report on the algorithmic language Algol 68. *Acta Informatica* 5 (1975) 1–236
- [Wexelblat 81] Wexelblat, R. L. (Hrsg.): History of programming languages. New York: Academic Press 1981
- [Wirsing 90] Wirsing, M.: Algebraic specification. In: Leeuwen, J. v. (ed.): Handbook of theoretical computer science. Vol B: Formal models and semantics. Amsterdam: Elsevier 1990, 675–788
- [Wirth 71] Wirth, N.: The programming language Pascal. *Acta Informatica* 1 (1971) 35–63
- [Wirth 83] Wirth, N.: Programming in Modula-2. 2. Aufl. New York: Springer 1983
- [Zuse 72] Zuse, K.: Der Plankalkül. *Berichte der Gesellschaft für Mathematik und Datenverarbeitung*, Nr. 63. Bonn 1972