TU Wien

Business Informatics Group
Institut für Softwaretechnik und Interaktive Systeme

Proseminar Grundlagen wissenschaftlichen Arbeitens
Programmiersprachen

Leitung: Wolfgang Hoschek

Christian Donhofer 0125815
Michael Jeremias 0126564
Hans Zojer 0226374

Wien, 6. Mai 2003

Inhaltsverzeichnis

1. Einleitung ... 3
2. Programmiersprachen allgemein ... 4
   - 2.1 Methodische Grundlagen: ... 5
   - 2.2 Abstrakte Datentypen (ADT, Typ) ... 5
   - 2.3 Programmierparadigmen ... 6
   - 2.4 Elemente von Programmiersprachen ... 7
     - 2.4.1 Syntax ... 7
     - 2.4.2 Semantik ... 7
   - 2.5 Lebensdauer und Bindungen ... 7
   - 2.6 Datentypen und Typbindungen ... 9
     - 2.6.1 Grundtypen ... 9
     - 2.6.2 Zusammengesetzte Typen ... 9
     - 2.6.3 Vereinigungstypen und polymorphe Typen ... 10
     - 2.6.4 Typäquivalenz ... 10
   - 2.7 Ausdrücke ... 10
   - 2.8 Sequentielle Ablaufsteuerung ... 11
   - 2.9 Zuweisungen, bedingte Anweisungen und Schleifen ... 11
     - 2.9.1 Fallunterscheidung ... 11
     - 2.9.2 Schleifen ... 12
   - 2.10 Prozeduren und Funktionen ... 12
   - 2.11 Ausnahmen ... 12
   - 2.12 Skriptsprachen ... 13
   - 2.13 Historie der Programmiersprachen ... 14
3. Prozedurorientierte Programmierung ... 19
   - 3.1 Grundbegriffe ... 19
   - 3.2 Strukturierte Programmierung ... 20
   - 3.3 Modulbildung und Schnittstellen ... 22
   - 3.4 Entwurfs- und Zerlegungstechniken ... 22
4. Objektorientierte Programmierung ... 24
   - 4.1 Beschreibung ... 24
   - 4.2 Objekte ... 24
   - 4.3 Methoden (Funktionen) ... 26

4.4 Vererbung 27
4.5 Polymorphie (Vielgestaltigkeit) 29
4.6 Zusätzliche Informationen 31
5. Funktionale Programmierung 32
5.1 Grundbegriffe 32
5.2 Beispiel: Haskell 33
Literaturverzeichnis 35

1. Einleitung

Über die Turing-Maschine, dem ersten Computer der Welt, welcher nur als Konzept auf Papier existierte, wurde geschrieben: „Der Computer ist das Fließband im Kopf“. Doch nicht die Maschine allein besitzt die Macht, jede Aufgabe zu bewältigen. Erst durch Programme, die dem Computer die Anweisungen für das Vorgehen liefern, wird dieser zum universellen Problemlöser.

Programmiersprachen gibt es heutzutage wie Sand am Meer. Hier liegen die Fragen nahe: Welche ist die beste? Wozu brauchen wir so viele verschiedene? Wieso gibt es nicht eine Programmiersprache, die alles kann? Doch jede Programmiersprache hat irgendwo ihre Vor- und Nachteile. Welche Sprache eingesetzt wird, hängt von vielen Faktoren ab, denn nicht alle Sprachen sind für jede Aufgabenstellung geeignet, gleich effizient, überall verfügbar oder überhaupt wirtschaftlich.

Die Seminararbeit soll einen Überblick über die wichtigsten Programmiersprachen und Programmiertechniken liefern. Sie soll grundsätzlich erklären, wie Programmiersprachen aufgebaut sind und wie sie definiert sind, wo ihre Unterschiede sind und wo sie Gemeinsamkeiten aufweisen. Gezeigt werden zu Beginn, wie eine Programmiersprache aufgebaut ist und ein Überblick über die historische Entwicklung der Programmiersprachen. In weiterer Folge wird auf die Programmierparadigmen der prozedurorientierten, der objektorientierten und der funktionalen Programmiersprachen näher eingegangen.

2. Programmiersprachen allgemein

Unter einer Programmiersprache versteht man eine Notation für Programme, die für den menschlichen Leser verständlich sein soll. Die einfachsten Formen sind maschinennahe Sprachen, Sprachen die sich mehr nach dem menschlichen Verständnis richten nennt man höhere Sprachen. Obwohl höhere Programmiersprachen schwieriger zu implementieren sind, sind sie maschinennaher Programmierung vorzuziehen, da die Fehleranfälligkeit geringer ist. Höhere Programmiersprachen klassifiziert man als imperativ, funktional und logisch. Imperative Sprachen beschreiben eine Berechnung als Folge von Zustandsübergängen, die durch 'Befehle' oder 'Anweisungen' erfolgen. In funktionalen Sprachen beschreibt ein Programm die Berechnung des Ergebnisses einer mathematischen Funktion. Das Programm ist eine Menge von Funktionsdefinitionen und Funktionsaufrufen. In logischen Programmiersprachen bestätigt man eine Formel der mathematischen Logik, indem man das Gegenteil zu widerlegen versucht. Die Werte der vorkommenden logischen Variablen, die zur Widerlegung führen, bilden das Ergebnis dieser Berechnung.

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 Programmierung. Folgende programmierspezifische Kriterien sollten bei der Planung von Projekten berücksichtigt werden:

- Praktische Benutzbarkeit: Die Sprache soll die Denk- und Ausdrucksweisen der Programmierer weitgehend unterstützen, ohne die Benutzung von Spezialfunktionen, die der Rechner, das Datenbanksystem oder andere Komponenten bereitstellen, unnötig zu erschweren.

- Verständlichkeit und Genauigkeit der Spezifikation: die Programmiersprache ist durch Spezifikationen vorgegeben. Internationale Standards und Normen helfen Missverständnisse beim Schreiben und beim Lesen zu vermeiden.

- Leichte Implementierbarkeit: geringer Aufwand bei der Implementierung von Programmen hilft den Zeitaufwand geringer zu halten und damit Kosten zu senken. Oft führt leichte Implementierbarkeit auch zu effizienteren Zielprogrammen.

- Portabilität der Programme: um langlebige, oder für einen großen Kundenkreis gedachte Programme zu schaffen müssen diese auf unterschiedlichen Rechnersyste-

men gleich funktionieren und von mehreren Übersetzern unterstützt werden. Besonders bei der Ein/Ausgabe, beim Aufruf von Betriebssystemfunktionen und bei der Verknüpfung von Programmteilen unterschiedlichen Ursprungs gibt es Schwierigkeiten, auf die Bereits in der Entwurfsphase Rücksicht genommen werden muss.

- Kompatibilität

2.1 Methodische Grundlagen:

Programme in einer Programmiersprache beschreiben im einfachsten Fall Algorithmen, die Eingaben in Ausgaben umwandeln. Bei einem reaktiven System wird ein Strom von Eingaben, die nur schrittweise bekannt werden, zu einem Strom von Ausgaben verarbeitet, welcher in seiner Länge, so wie die Anzahl der Ströme selbst, unbeschränkt ist. Reaktive Systeme setzen sich aus Prozeduren (einzelne Algorithmen) zusammen, der als Schnittstelle die Spezifikation der zulässigen Eingaben sowie die Ausgabe der Berechnung hat. Die Ausführung einer Prozedur wird im Gesamtprogramm durch einen Prozeduraufruf veranlasst. Unter einer Nebenwirkung einer Prozedur versteht man wenn die Prozedur von sich aus auf Daten zugreift oder sie verändert, ohne dass diese in der Schnittstelle spezifiziert wurden. 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 einzelnes Resultat liefern, nennt man Funktionen.

2.2 Abstrakte Datentypen (ADT, Typ)

Typen bilden heutzutage in allen Programmiersprachen die Grundlage der Beschreibung von Daten. 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. [Rech03]

Ein ADT besteht aus einer Menge von Operatoren, ein Operator repräsentiert eine n-stellige Funktion, die Argumente der Typen auf einen Typ abbilden. Ein nullstelliger Operator heißt Konstante, die Menge der Operatoren eines Typs zusammen mit ihrer Stelligkeit heißt Signatur. Eine Konstante vom Typ T ist ein Term, Variable sind ebenfalls Terme eines Typs T. Terme ohne Variable heißen Grundterme. Dieser Variablenbegriff darf nicht mit den Variablen imperativer Programmiersprachen verwechselt werden, die ihren Wert in der Zeit ändern koennen. Grundterme koennen intuitiv ausgewertet werden. Dazu werden in abstrakten Datentypen Gleichungen definiert, mit deren Hilfe man Terme auswerten oder

vereinfachen kann. Gleichungen werden meist zur Definition der Bedeutung der Operatoren verwendet. Konstruktoren sind die Werte, die eine Gleichung im Datentyp annehmen kann, z.B. `true` oder `false` beim Typ `BOOL`, oder `-2, -1, 0, 1, 2,..` beim Typ `INT` (Integer). In funktionalen Sprachen kann man die Konstruktoren direkt festlegen und dann Axiome definieren. Sind alle Konstruktoren eines Typs nullstellig, so heißt der Typ Grundtyp, sonst zusammengesetzter Datentyp. Ein Datentyp T' ist ein Untertyp des Typs T, wenn zwei Terme t, t' aus T' genau dann aequivalent sind, wenn sie in T aequivalent sind. 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.

Viele Kombinationen von Operationen, z.B. bei Listen, sind nicht spezifiziert. In der praktischen Anwendung müsste eine solche Kombination zu einer Ausnahme führen, die eine Fehlerbehandlung oder den Programmabbruch einleitet. Um Konsistenz und Vollständigkeit zu prüfen, hat sich in der Praxis eine tabellarische Erfassung der Definition abstrakter Datentypen bewährt.

2.3 Programmierparadigmen

Beim imperativen Programmierparadigma betrachtet man, wie bereits erwähnt, eine Berechnung als Folge von Zustandsübergängen. Diese koennen sehr komplex sein und entsprechen dann unter Umständen umfangreichen Prozeduren. Sie koennen unter Bedingungen stehen, oder mit jeweils anderen Variablenwerten in Schleifen ausgeführt werden. Theoretisch sind die Elemente Prozedur, Prozeduraufruf, bedingte Anweisung und Schleife ausreichend, um alle Algorithmen zu formulieren. Praktisch ergeben sich die gleichen Elemente, wenn man das Programm durch schrittweise Verfeinerung mit Vor und Nachbedingungen zur Verifikation nach Hoare und Dijkstra betreibt [Hoar69, Dijk76]. Man spricht vom Paradigma des strukturierten Programmierens, wenn für die Ablaufsteuerung nur diese Elemente eingesetzt werden. Sprünge (willkürliche Bestimmung der nächsten auszuführenden Anweisung) sind dabei nicht erlaubt, Ausnahmen bilden Formulierungen von Alternativen Programmausführungen, um nach einem Fehler, z.B. Division durch Null, Zugriff auf eine nicht vorhandene Datei,... , wieder einen Zustand zu erreichen, in dem die Programmausführung ordnungsgemäß fortgesetzt und zu Ende geführt werden kann.

Da beim Zusammensetzten größerer Programme Prozeduren als Strukturierungshilfe nicht mehr ausreichen, richtet man sich nach dem Paradigma des modularen Programmierens, bei dem ein Programm aus mehreren Modulen besteht, die über eine gemeinsame Schnittstelle verfügen. Nur auf die in der Schnittstelle angegebenen Groessen kann auch von au-

ßerhalb zugegriffen werden, alle Groessen innerhalb der Module unterliegen dem Geheimhaltungsprinzip und sind von außen nicht sichtbar.

2.4 Elemente von Programmiersprachen

Die Bedeutung eines Programms ergibt sich durch das Zusammensetzen der Bedeutungen der einzelnen Elemente. Die Sprachelemente sind ausführbare Operationen, die den Stand der Berechnung beeinflussen.

2.4.1 Syntax

Die von der Programmiersprache vorgegebene Struktur der Anordnung der Elemente wird als Syntax bezeichnet, die mit Hilfe einer kontextfreien Grammatik (KFG) in erweiterter Backus-Naur Form (EBNF) definiert wird. Die Syntax bezeichnet den genauen Wortlaut einer Anweisung und gibt die Positionen an, wo eventuell vorkommende Parameter anzugeben sind.

2.4.2 Semantik

Es gibt drei verschiedene Formen der Definition der Semantik von Programmiersprachen:

- Operationale Semantik beschreibt die Wirkung eines Programms als schrittweise Zustandsänderung.
- Denotationale Semantik begreift ein Programm als Abbildung seiner Eingaben in seine Ausgaben.
- Die axiomatische Semantik geht davon aus, dass Ein- und Ausgaben bestimmte Vor- und Nachbedingungen erfüllen. Funktionale Sprachen werden mit denotationaler Semantik definiert, alle imperativen Sprachen verwenden operationale Semantik.

2.5 Lebensdauer und Bindungen

Jedes Objekt (konstante Werte, Variablen, Ergebnisse von Operationen) hat eine bestimmte Lebensdauer, nämlich den Teil während einer Programmausführung, in der es benutzt werden kann. Man unterscheidet zwischen

- persistenten Objekten (Objekte die auch vor Beginn oder nach dem Ende der Programmausführung existieren, z.B. Datenbankeinträge),
- Objekten beschränkter Lebensdauer (werden bei der Einführung von Bezeichnern gebildet und koennen gelöscht werden, wenn der Bezeichner ungültig wird) und
- Objekten unbeschränkter Lebensdauer (werden durch eine explizite Operation "neue Variable" gebildet, welche die Referenz auf die Variable als Ergebnis liefert. Die Variable mindestens so lange, wie diese Referenz als Wert einer anderen Variable vorkommt).

Im Programmtext finden sich keine Objekte, sondern Bezeichner und Operatoren zur Benennung von Objekten und Operationen. Diese Zuordnung heißt Bindung, und wird in folgende Fälle unterschieden:

- Statische Bindung: sie wird durch eine Vereinbarung im Programm festgelegt. Es gibt einen Programmteil, gegeben durch eine syntaktische Struktur, in dem diese Zuordnung gültig ist (Gültigkeitsbereich). Diese Bereiche koennen auch geschachtelt sein.
- Dynamische oder polymorphe Bindung: Die Bindung von Operatoren 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. Ist `a`, `b` durch die statische Semantik bereits ein Typ zugeordnet so liegt statische Bindung vor, bei dynamischer Bindung geschieht das erst während der Ausführung der Operation.
- Dynamische Bindung bei Vereinbarung: 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: ein Bezeichner benennt während der Ausführung das jeweils jüngste unter diesem Bezeichner eingeführte Objekt. Der syntaktische Kontext der Anwendung des Bezeichners bleibt unberücksichtigt, was sich, insbesondere bei Programmänderungen als sehr fehleranfällig erwiesen hat.

- Bindung bei Programmstart: bei Programmstart werden die vom Betriebssystem verwalteten Dateien des Programms vereinbarten Dateibezeichnern zugeordnet, danach ist diese Bindung unveränderlich.

Vereinbarungen führen also zur Bildung von Objekten und sind damit ausführbare Operationen, außerdem spezifizieren sie eine Bindung zwischen Bezeichner und dem benannten Objekt.

2.6 Datentypen und Typbindungen

Der Typ einer Variablen definiert die Menge ihrer möglichen Werte. Ein Wert einer Variable ist durch eine Bitfolge codiert, der durch die Definition des Typs dann in einer bestimmten Art interpretiert wird, z.B. als boolesche Werte, ganze Zahlen, Gleitpunktzahlen, etc. Hängt die Interpretation, und damit der Typ, von der jeweils ausgeführten Operation ab, so heißt die Programmiersprache typfrei, andernfalls typisiert oder typgebunden. Typisierte Sprachen koennen in zwei weitere Richtungen 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.

2.6.1 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 die Zahlen, Gleitpunktzahlen, Zeichen und Wahrheitswerte gebräuchlich. Ausschnittstypen definieren eine Teilmenge aufeinanderfolgender ganzer Zahlen oder Einzelzeichen (characters).

2.6.2 Zusammengesetzte Typen

Ein Feld eines zusammengesetzten Typs besteht aus Elementen und Stufen, die durch Indizierung zugänglich gemacht werden. Bei einem statischen Feld gehören die Indexmengen zum Typ. Die Unter- und Obergrenzen sind im Programm durch statisch berechenbare Konstanten gegeben. Bei einem dynamischen Feld gehören die Grenzen nicht zum Typ, sie werden bei Bildung eines Feldobjekts, z.B. durch eine Vereinbarung, festgelegt und sind dann fest. Wie bei einem dynamischen Feld gehören beim Flexiblen Feld die Grenzen auch nicht zum Typ, sie koennen sich aber bei jeder Zuweisung an eine Feldvariable ändern.

2.6.3 Vereinigungstypen und polymorphe Typen

Ein Vereinigungstyp beschreibt eine disjunkte Vereinigung von Typen. Objekte dieses Typs koennen wahlweise Werte eines der Typen annehmen. Sei T ein Verbundtyp mit Feld t und x eine Variable vom Typ union(T, int), dann ist der Zugriff x.t auf die Komponente t von x nur dann erlaubt, wenn T der Typ des aktuellen Wertes von x ist. Mit anderen Worten: will man eine Vereinigung herbeiführen, muss die Variable, die den neuen Wert annehmen soll, dem Typ entsprechen, mit dem die Vereinigung stattfindet.

Ein Vereinigungstyp heißt polymorph, wenn die Interpretation der auf ein Objekt angewandten Operatoren oder Prozeduren vom Typ des aktuellen Werts abhängt. Das Polymorphe Auswahlverfahren kann ein- oder mehrstufig sein. In einigen Programmiersprachen (C++, Java,..) hängt die Auswahl der Operation nur vom ersten Operanden ab, in anderen Programmiersprachen (z.B. CLOS) von allen. 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, allerdings werden die polymorphen Typen nicht explizit benannt, sondern ihre Elementtypen aus dem Kontext bestimmt.

2.6.4 Typäquivalenz

Bei der Übergabe eines Objekts zwischen verschiedenen Modulen oder Klassen wechselt der Objekttyp im Allgemeinen nicht. Daher muss festgestellt werden koennen, ob und wann die auf beiden Seiten einer solchen Schnittstelle erklärten Typen als gleich gelten. In allen neueren Sprachen verwendet man Namensgleichheit: zwei Typen gelten als gleich, wenn sie gleich benannt sind. Bei Strukturgleichheit sind zwei Typen aequivalent, wenn ihre Algebren als abstrakte Datentypen betrachtet isomorph sind. Bei der Strukturgleichheit dynamischer und flexibler Felder wird die Anzahl der Stufen und der Elementtyp berücksichtigt, nicht jedoch die Anzahl der Elemente.

2.7 Ausdrücke

Ausdrücke berechnen (Zwischen-)Ergebnisse durch Anwendung von Operationen auf ihre Operanden. Ein Ausdruck regelt die Reihenfolge der Operandenzugriffe und der Operationen.

2.8 Sequentielle Ablaufsteuerung

Um beliebige Algorithmen auszudrücken genügen bedingte Ausdrücke bzw. Anweisungen zusammen mit Schleifen oder rekursiven Funktionen. In imperativen Sprachen ist zusätzlich die Zuweisung erforderlich. Die Ablaufsteuerung definiert, wie diese Anweisungen ausgeführt werden und damit der Gesamtablauf eines Programms gesteuert wird. In imperativen, höheren Sprachen ist die Zuweisung die wesentliche, zustandsverändernde Operation. Die Zustandsänderung ist eigentlich eine Nebenwirkung. Andere Sprachelemente koennen durch die in ihnen enthaltenen Zuweisungen ebenfalls Nebenwirkungen hervorrufen.

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

2.9 Zuweisungen, bedingte Anweisungen und Schleifen

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.

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 `S1` ausgeführt, sonst die Anweisungen `S2`. Fehlt der `else`-Zweig und ist das Resultat der Auswertung von `b` `false`, wird keine Anweisung ausgeführt.

2.9.1 Fallunterscheidung

Unter einer Fallunterscheidung kann man sich eine bedingte Anweisung mit mehreren Fällen vorstellen, z.B. `if e=l1 then S1 else if e=l2 then S2 else.. else if e=lk then Sk else S0`, wobei `e` nur einmal berechnet wird. 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.

2.9.2 Schleifen

Die Bedeutung der Schleife `while b loop S` wird auf die bedingte Anweisung `if b then S; while b loop S` zurueckgefuehrt. Ist die boolesche Bedingung `b` wahr, 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. `break` in C und C++). Bei geschachtelten Schleifen kann man die einzelnen Schleifen auch benennen und dann mit break name bestimmen, welche Schleife verlassen werden soll.

2.10 Prozeduren und Funktionen

Der Begriff Prozedur wird allgemein als Oberbegriff für echte Prozeduren, (Funktions-)Prozeduren und auch für Operationen wie die ein- oder zweistellige Subtraktion gebraucht. Bei einem Prozeduraufruf `f(a1, ..., an)` mit den Argumenten `a1, ..., an` werden die Parameterspezifikationen zu initialisierten Vereinbarungen ergänzt, und diese dem Rumpf `S` vorangestellt. Der so ergänzte Rumpf, ein Block, wird dann an Stelle des Aufrufs ausgeführt.

Eine Funktion p heißt rekursiv, wenn während ihrer Ausführung p nochmals aufgerufen wird. Auch indirekte Rekursion, bei der p eine Funktion q aufruft, die wiederum p aufruft, ist möglich. Fast alle Programmiersprachen erlauben rekursive Funktionen, verboten ist sie jedoch in COBOL und Fortran.

Prozedurvereinbarungen kann man als Vereinbarungen konstanter Objekte ansehen. In einigen Programmiersprachen koennen Prozeduren auch als Argumente von Prozeduren auftreten. Der Typ dieses Prozedurobjekts ist die Signatur der Prozedur. Die Vereinbarung bindet globale Groessen aus ihrer Umgebung, bei der Übergabe von Prozeduren muss neben der Prozedur daher auch diese Umgebung übergeben sein. Das Paar (Umgebung, Prozedur) nennt man Hülle (closure) der Prozedur. Bei Aufruf eines Prozedurparameters wird zuerst diese Umgebung wieder hergestellt und dann die Prozedur ausgeführt.

2.11 Ausnahmen

Ausnahmen signalisieren, dass eine Vorbedingung einer Operation verletzt wurde, und die Operation daher nicht ordnungsgemaess durchgefuehrt werden konnte (z.B. bei Division durch 0). Ziel in diesen Faellen ist es, einen konsistenten Zustand herzustellen, in dem die Programmausfuehrung fortgesetzt werden kann, andernfalls wird der Programmablauf ab-

gebrochen. Durch die `raise`-Anweisung steht ein Mittel zur programmgesteuerten Ausloesung von Ausnahmen und ein spezielles Sprachelement, die Ausnahmebehandlung, zum Abfangen von hard- oder softwaregesteuerten Ausnahmen zur Verfuegung. Eine Ausnahmebehandlung bezieht sich immer auf 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 laeuft so lange, bis eine Ausnahmebehandlung oder das Ende der Programmausfuehrung erreicht ist.

2.12 Skriptsprachen

Mit *Skriptsprachen* fügt man Programmkomponenten zu größeren Programmen auf eine andere Weise als beim modularen oder objektorientierten Programmieren zusammen: Die Komponenten sind selbstständige Programme P_1, P_2, . . . , die in unterschiedlichen Sprachen geschrieben sein können oder sogar nur übersetzt vorliegen [Rech03];

Beispiele für Skriptsprachen sind: Perl, Tk/Tcl, Python, UNIX-Shellskripts, MSDOS-Batchprogramme, REXX-Prozeduren, JCL (Job Control Language), JavaScript, VB-Skript.

Eine Skriptsprache kann man an folgenden Merkmalen erkennen:

- *Imperative Ausführung:* Eine Skriptsprache ist im Regelfall eine Programmiersprache der dritten Generation. Es handelt sich dabei also um eine Sprache mit befehlsorientierter Programmdurchführung.

- *Interpreter:* Gewöhnlich wird eine Skriptsprache nicht kompiliert, sondern direkt interpretiert oder höchstens rudimentär übersetzt.

- *Eingeschränkte Datentypen:* Normalerweise hat eine Skriptsprache hat nur wenige Datentypen oder die Variablen haben gar keinen bestimmten Datentyp (Typfreiheit), sondern werden nach Bedarf als Zeichenkette oder Zahl interpretiert.

- *Fremdprogramme:* Meist kann man aus einem Skript heraus ein Programm starten, das in Maschinencode vorliegt oder Betriebssystemkommandos durchführen.

Das folgende Shellskript liest beliebig viele Zahlen ein und gibt die Summe aus:

```
#!/usr/bin/bash

sum=0
read -p "enter number: " i
```

```
while test "$i" != "x"
do
     if expr "$i" + 0 > /dev/null 2> /dev/null
     then sum=`expr $sum + $i`
     else echo "Not a number"
     fi

     read -p "enter number: " i
done

echo $sum
```

In dem vorangegangenen Programm wird als erstes die Variable `$sum` angelegt und mit `0` initiiert (`sum=0`). Der nächste Befehl `read -p "enter number: " i` liest eine Eingabe von der Tastatur und speichert sie in der Variablen `$i`. Der Ausdruck `test "$i" != "x"` liefert falsch, wenn der Benutzer ein „x“ zum Programmbeenden eingegeben hat; in diesem Fall wird das Programm beendet und die Summe ausgegeben (`echo $sum`). Hat der Benutzer etwas anderes eingegeben, dann wird der Block in der `while-do-done`-Schleife durchgeführt. Dort wird geprüft, ob es sich bei der Eingabe um eine Zahl handelt (`if expr "$i" + 0 > /dev/null 2> /dev/null`). Wenn ja, wird die eingegebene Zahl `$i` zur Summe `$sum` addiert (`sum=`expr $sum + $i``). Wenn nicht, wird eine Fehlermeldung ausgegeben (`echo "Not a number"`). Danach wird wieder eine Eingabe von der Tastatur gelesen und an den Anfang der Schleife gesprungen, wo überprüft wird, ob ein „x“ eingegeben wurde. Die Schleife wird so lange wiederholt bis jemand ein „x“ eingibt.

2.13 Historie der Programmiersprachen

Die früheren Programmiersprachen hat man in Generationen eingeteilt. Bei neueren Sprachen hat man diese Einteilung aufgegeben und ist zu einer Klassifikation nach den grundlegenden Konzepten – wie Objekt-, Funktions- oder Logikorientierung - übergegangen [Hans98].

Maschinensprachen (Sprachen der ersten Generation) sind die Programmiersprachen, deren Programme ein Computer direkt ausführen kann. Die einzelnen Befehle werden in binärer Form (als Folge von 0- und L-Zeichen) eingegeben. Diese Programme sind aufwendig zu schreiben, extrem Fehleranfällig und schwer zu warten.

Beispiel – in Register BL den Wert 7 speichern und zu BL den Wert 5 addieren:

```
L0LL 00LL 0000 0LLL
L000 0000 LL00 00LL 0000 0L0L
```

Assemblersprachen (Sprachen der zweiten Generation) haben als Befehlssatz Abkürzungen, sogenannte Mnemonics, die 1:1 dem Code der Maschinensprachen entsprechen. Gegenüber von Sprachen der ersten Generation ist das natürlich eine starke Vereinfachung.

Beispiel – in BL den Wert `7` speichern (`MOVe`) und zu `BL` den Wert `5` addieren:

```
MOV BL,7
ADD BL,5
```

Prozedurale Programmiersprachen (Sprachen der dritten Generation) sind die heutzutage am weitesten verbreiteten Sprachen.

Beispiel – den Wert `7` und `5` addieren und auf die Variable `SUMME` speichern:

```
SUMME = 7 + 5
```

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 [Rech03].

Die älteste Programmiersprache für technisch-wissenschaftliche Anwendungen ist FORTRAN (FORmula TRANslator) [Hans98]. Die erste Fassung dieser Sprache entstand 1954 und die erste eigentliche Version erschien 1957 (FORTRAN I). Der aktuelle Standard ist FORTRAN 90. Aus älteren Version entwickelten sich die Programmiersprachen ALGOL (ALGOrithmic Language) und Anfang der 60er Jahre BASIC (Beginners All-Purpose Symbolic Instruction Code). BASIC war lange Zeit für seine GOTO-Anweisungen und unleserlichen „Spaghetticode“ berüchtigt; heutige BASIC-Versionen hingegen enthalten Strukturierte Sprachelemente von Pascal.

COBOL (COmmon Business Oriented Language) wurde sehr früh (1960) normiert und ist eine weit verbreitete Sprache für betriebswirtschaftliche Anwendungen. Die derzeitige Norm ist ANSI-COBOL-85. Aus COBOL, FORTRAN und ALGOL wurde die Sprache PL/I entwickelt, die sowohl für kommerzielle, als auch für technisch-wissenschaftliche Problemstellungen geeignet ist. PL/I enthält daher einen großen Sprachumfang und ist daher etwas schwieriger zu erlernen.

Pascal wurde 1971 vom Züricher Hochschulprofessor Niklaus Wirth entwickelt und ist DIE strukturierte Programmiersprache schlechthin. Sie ist nach dem französischem Mathematiker und Philosophen *Blaise Pascal* (1923 – 1662) benannt.

Modula-2 ist eine Weiterentwicklung von Pascal, wobei besonderer Wert auf die Unterstützung der modularen Programmierung gelegt worden ist [Hans98].

Somit kann ein Programm auch in mehrere Teilprogramme (Module) aufgegliedert werden. Die Pascal ähnliche Sprache Ada (erster Standard 1983) ist nach der „ersten Programmiererin der Welt“, *Augusta Ada Byron* (1815 – 1852) benannt. Sie wurde für embedded Systems entwickelt, um besonders effiziente und zuverlässige Programme zu erstellen. Leider hat diese Sprache – so wie PL/I – einen komplexen Sprachumfang.

Die Sprache C ist Anfang der 70er Jahre entstanden. Sie ist als „Hochsprachenassembler“ für hardwarenahe Programmierung geeignet. Pascal und C haben sich in ihrer Entwicklung gegenseitig beeinflusst und daher sind die strukturellen Elemente und der Aufbau dieser Sprachen sehr ähnlich. C zeichnet sich durch optimale Hardwareausnutzung und effizientes Laufzeitverhalten aus. Daher konnte in vielen Bereichen auf Assemblerprogramme verzichtet werden; das Betriebssystem UNIX wurde beispielsweise fast ausschließlich in C geschrieben. Zu den Nachfolgern von C gehören die Objektorientierten Sprachen C++, Java und C# (C-Sharp gesprochen).

Sprachen der vierten Generation sind *Makrosprachen* oder *Generatorsprachen*. Weil mit der Zeit die Nachfrage nach neuen Programmen durch die Fachabteilungen in vielen Betrieben stark angestiegen war, gab es einen Engpass in der Versorgung. Daher wurden Entwicklungsumgebungen und Sprachen entwickelt, die komplette Programme oder Programmsysteme generieren konnten. Vorteil dieser Sprachen ist, dass man in kurzer Zeit sehr umfangreiche Programme mit geringem Aufwand erstellen kann. Meist erlauben diese Sprachen die Verwendung von unterschiedlichen Dateiverwaltungs- bzw. Datenbanksystemen. Nachteil ist, dass sich nicht alle Probleme damit lösen lassen oder nur mit erhöhtem Aufwand. Außerdem sind diese Sprachen oft herstellerspezifisch und laufen meist nur in bestimmten Umgebungen.

Die bekannteste Sprache der vierten Generation ist SQL. Hierbei handelt es sich um eine Generatorsprache, mit der man Datenbankabfragen auf einfache Weise erstellen kann.

Beispiel - Gib alle Lieferfirmen aus, bei denen etwas bestellt wurde:

```
SELECT   Lieferantennummer, Firma
FROM     Lieferanten, Bestellungen
WHERE    Lieferanten.Lieferantennummer =
              Bestellung.Lieferantennummer
ORDER BY Firma
```

[Hans98]

Beispiele für *Systeme der vierten Generation* sind: ABAP/4 (SAP), AS, CSP und QMF (IBM), Drive/Windows (Siemens), Informix/4GL (Garmhausen & Partner), Mapper (Unisys) und SQL*Forms.

Zu den Sprachen der Alternativen Sprachrichtungen gehören: funktionale Programmiersprachen, logische Programmiersprachen, verteilte und parallele Programmiersprachen und objektorientierte Programmiersprachen. Mit zunehmender Steigerung der Hardwareleistung wurden Ansprüche, wie effiziente Laufzeit und Speicherplatz unbedeutender und andere Kriterien, wie zum Beispiel Entwicklungsaufwand und Wartungskosten, sowie die Wiederverwendbarkeit von Programmcode, wurden entscheidend.

Funktionale Programmiersprachen sind Sprachen die aus mathematische Formeln bestehen oder größtenteils solche enthalten. LISP (LISt-Processing langage) wurde in den 50er Jahren entwickelt und wird zum Programmieren von künstlicher Intelligenz verwendet. In den 70er Jahren entstand Scheme; der derzeitige Standard ist COMMON LISP. Andere funktionale Programmiersprachen sind beispielsweise APL, Caml, Haskell, Hope, Mathematika und SASL.

Logische Programmiersprachen beschäftigen sich mit der Logik, einem Teilgebiet der Mathematik. PROLOG (PROgramming LOGic) entstand in den frühen 70er Jahren in Frankreich. Diese Programmiersprache hat sich allerdings nur in Nischen durchsetzen können: Bei Expertensystemen und bei bestimmten Datenbanksystemen.

Verteilte und parallele Programmiersprachen sind Programmiersprachen, die spezielle Sprachelemente für die Verwendung von Mehrprozessorsystemen besitzen oder um gesamte Computernetzwerke zu Programmieren. Beispiele für solche Sprache sind OCCAM und SR und bestehende Sprachen wurden um entsprechende Sprachelemente erweitert: Concurrent C, Concurrent LISP, Distributed Smalltalk). Um bei allen Programmiersprachen eine einheitliche Erweiterung mit solchen Elementen zu schaffen, wurde LINDA entwickelt. C-LINDA, Perl-LINDA, REXX-LINDA haben somit alle die gleichen Erweiterungen.

Objektorientierte Sprachen sind Sprachen, bei denen nicht die Programmlogik sondern sogenannte Objekte im Vordergrund stehen. Die ersten Ansätze beginnen mit dem Entwurf von SIMULA 67, einer Sprache die für Simulationsaufgaben konzipiert war und eine Erweiterung zu ALGOL 60 darstellt. Als DER Urvater aller „reinen“ objektorientierten Sprachen gilt Smalltalk, die Alan Kay in seiner Doktorarbeit (1969) beschrieb. Diese Sprache wurde zum ersten Mal 1980 durch Xerox PARC implementiert. Die jüngste, „rein“ objektorientierte Sprache ist Java von Sun. Es gibt eine Reihe von prozeduralen Hybridsprachen, zu denen zählen C++, Eiffel und Ada 95. Nicht normierte Hybridsprachen sind beispielsweise Object COBOL, Object PASCAL, Perl5, Object REXX, CLOS und PROLOG++.

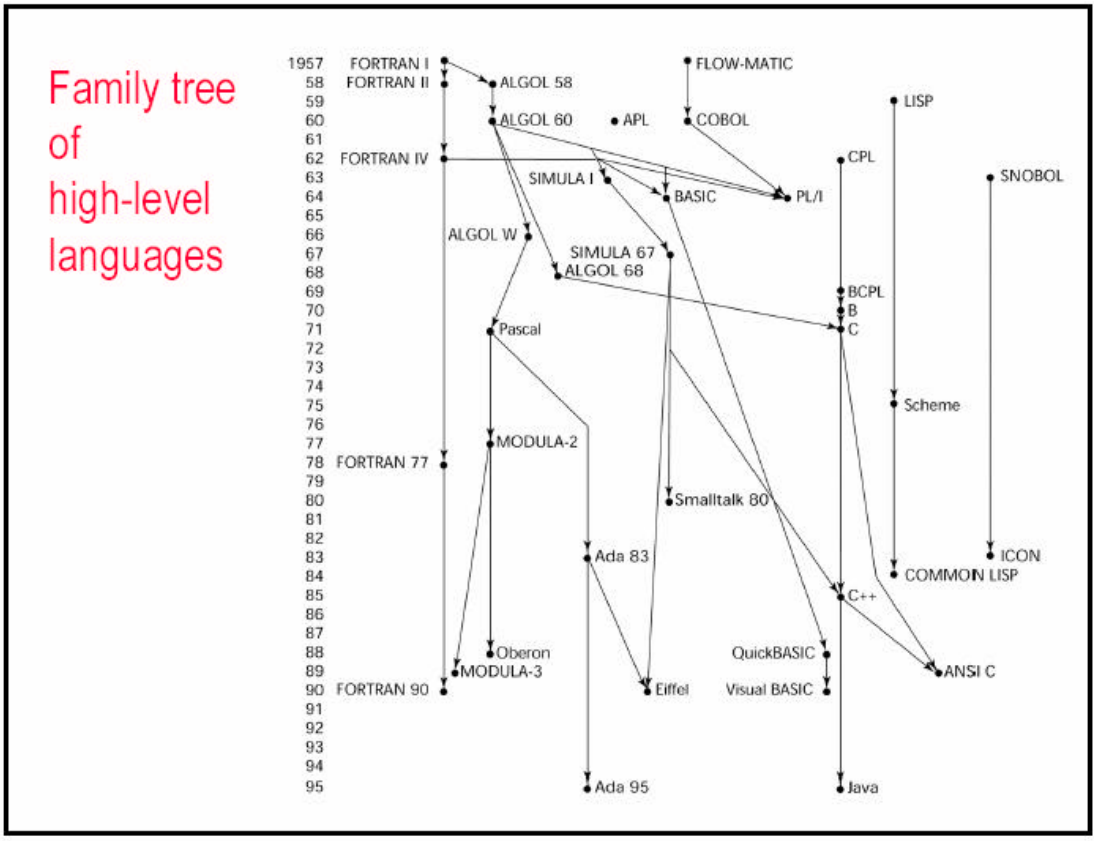

Abbildung 1: Historische Entwicklung der höheren Programmiersprachen [Dave02]

Programmgeneratoren und Softwareentwicklungsumgebungen wurden geschaffen, um mit grafischen Hilfsmitteln Programmcode zu generieren. Ursprünglich wollte man das Erzeugen von Bildschirmmasken und Menüsystemen, sowie Druckprogrammen erleichtern. Beispiele dafür sind Borland C++, Smalltalk und Visual C++.

3. Prozedurorientierte Programmierung

Das bis heute in der Praxis vorherrschende Programmierparadigma ist die imperative Programmierung, die auf dem Konzept des v.Neumann-Rechners aufbaut. Sie beruht auf dem Zustandskonzept [Rech98].

Imperative Programme werden als eine Folge von Befehlen Schritt für Schritt abgearbeitet. Variable in solchen Programmen nehmen gewisse Werte an, die Ihnen eine Anweisung zuweist. Dadurch können sich die Werte der Variablen ändern, während das Programm ausgeführt wird. Außerdem gibt es Sprungbefehle, die unbedingt (immer) oder bedingt (möglicherweise, abhängig von bestimmten Zuständen) ausgeführt werden und das Programm arbeitet die Befehle somit an einer anderen Stelle weiter ab. Vorteil dieser Programmierung ist, dass man damit alle lösbaren Probleme programmieren kann. Nachteil ist, dass es für bestimmte Aufgabenstellungen sehr aufwendig sein kann.

3.1 Grundbegriffe

Eine *Methode* (engl.: method) beschreibt eine systematische Vorgehensweise zur Lösung eines Problems. Ist diese Verfahrensvorschrift exakt und vollständig formuliert, so handelt es sich um einen *Algorithmus* (engl.: algorithm). Er gibt an, wie Inputgrößen bei einem gegebenen Zielsystem in Outputgrößen umzuwandeln sind [Hans98]. Ein Algorithmus heißt *abbrechend*, wenn er die gesuchten Größen nach endlich vielen Schritten liefert, anderen falls heißt der Algorithmus *nicht abbrechend* [Rech03].

Eine *Prozedur* ist ein abgeschlossener Teil eines Programms und besteht aus mehreren Anweisungen. Sie kann aus einem anderen Programmteil heraus aufgerufen (gestartet) werden und hat Eingabe- und Ausgabeobjekte, sogenannte Argumente oder Parameter. Hat eine Prozedur genau ein Ausgabeobjekt so heißt sie *Funktion*. Dieses Ausgabeobjekt heißt in diesem Fall auch Ergebnis, Resultat oder Rückgabewert. Funktionen haben oft eine eigene Darstellung im Programmcode, um sie von Prozeduren zu unterscheiden.

Da Problemstellungen oft sehr groß sind, ist es notwendig ein Programm von mehreren Programmierern gleichzeitig entwickeln zu lassen. Damit nicht jeder das Gleiche noch ein-

mal programmiert, wird das Programm in Abschnitte zerteilt und den Programmierern zugeteilt. Diese Abschnitte nennt man *Module*.

Bei unterschiedlichen Programmiersprachen und Darstellungen werden die Begriffe „Prozedur“, „Funktion“, „Modul“ und „Methode“ anders gebraucht. In JavaScript beispielsweise wird sowohl eine Funktion als eine Prozedur als „Funktion“ bezeichnet. Beim Nassi-Shneiderman-Diagramm (Struktogramm) spricht man oft von „Modulen“, obwohl man eigentlich Prozeduren meint. Im Folgenden werden die Begriffe so verwendet, wie wir sie oben definiert haben.

3.2 Strukturierte Programmierung

Böhm und Jacopini haben 1966 bewiesen, dass sich jedes Programm aus drei Grundelementen aufbauen lässt: Sequenz, Selektion und Iteration. Daher kann man auf unbedingte Sprungbefehle (`goto`-Anweisungen) verzichten und dadurch wird der Programmcode übersichtlicher.

Das *Nassi-Shneiderman-Diagramm* (*Struktogramm*) ist eine Darstellungsweise, die genau diese Elemente enthält und die Verwendung von `goto`-Anweisungen verhindert.

Die *Sequenz* (Anweisungsfolge) bedeutet, dass mehrere Befehle untereinander geschrieben werden und nach der Reihe ausgeführt werden.

Abbildung 2: Sequenz

Die *Selektion* (Verzweigung, Auswahl, Entscheidung, Alternative, Bedingung) bedeutet, dass hier das Programm in verschiedene Möglichkeiten hat, fortzufahren. Dies ist abhängig davon, ob die Bedingung wahr oder falsch ist (Bsp.: `a < b` "Ist Variable a kleiner als Variable b?").

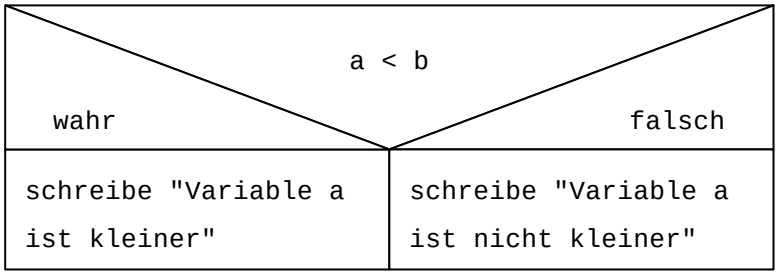

Abbildung 3: Selektion

Die *Iteration* (Schleife, Wiederholung) bedeutet, dass der Block innerhalb einer Schleife mehrfach hintereinander ausgeführt wird (wiederholt wird). Es gibt zwei grundlegende verschiedene Schleifen: die Kopf- und die Fußgesteuerten Schleifen. Bei der kopfgesteuerten Schleife wird zuerst geprüft, ob der Bedingungsteil wahr ist und dann wird der Block ausgeführt. Ist der Bedingungsteil falsch, wird nach der Schleife das Programm fortgesetzt. Bei der Fußgesteuerten Schleife wird zuerst der Block ausgeführt und dann wird geprüft. Die Schleife wird solange wiederholt, bis der Bedingungsteil wahr wird - also genau umgekehrt, wie bei der kopfgesteuerten Schleife.

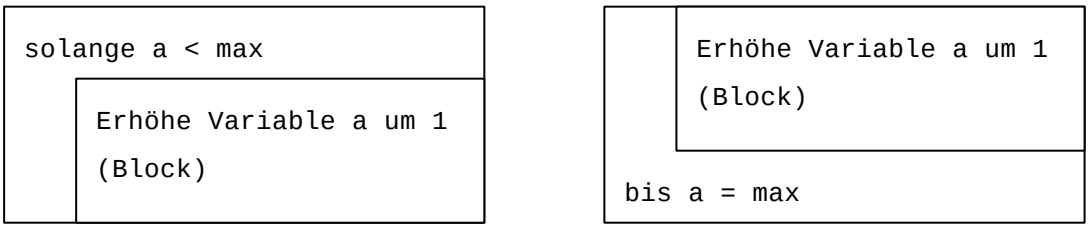

Abbildung 4: Kopfgesteuerte Schleife (Links) und fußgesteuerte Schleife (Rechts)

Der Prozeduraufruf startet eine andere Prozedur aus dem aktuellen Programmteil heraus und führt diese Prozedur aus. Sobald die Prozedur fertig abgearbeitet ist, wird zu diesem Programmteil zurückgekehrt und fortgesetzt. Eine Prozedur, die sich selbst aufruft, bezeichnet man als rekursiv.

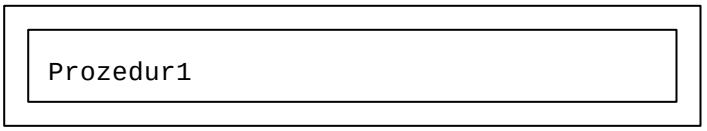

Abbildung 5: Prozeduraufruf

Es gibt noch einige andere nützliche Erweiterungen zum Nassi-Shneiderman-Diagramm auf die hier nicht näher eingegangen wird.

Gegen die Grundsätze der strukturierten Programmierung gibt es normalerweise in den Programmiersprachen eingeschränkte unbedingte Sprunganweisungen (`break`, `return`,

`throw`), die es dem Programm ermöglichen an anderer Stelle fortzufahren. Sie werden jedoch nicht als schlechter Programmierstil angesehen, im Gegensatz zur `goto`-Anweisung.

Der *unmittelbare Schleifenabbruch* (`break`) bewirkt, dass aus einer Schleife unverzüglich abgebrochen wird und hinter der Schleife fortgesetzt wird.

Die *Rückkehr aus einem Unterprogramm* (`return`) bewirkt, dass eine Prozedur oder Funktion unmittelbar beendet wird und an dem Programmteil fortgesetzt wird, wo diese aufgerufen wurde.

Die *Ausnahme* (`throw-catch`, `exception` oder `on error`) bewirkt, dass ein Programm bei einem Fehler an einer anderen Stelle fortgesetzt wird. Diese Ausnahmen können meist auch "manuell" ausgelöst werden - als Anweisung im Programm.

3.3 Modulbildung und Schnittstellen

Wie schon einleitend erwähnt, gibt es die Möglichkeit Programme auf verschiedene abgeschlossene Module aufzuteilen, die von einzelnen Programmierern getrennt entwickelt werden können. Zu diesem Zweck hat jedes Modul klar definierte Schnittstellen, über die es mit seiner Außenwelt kommuniziert. Diese Schnittstellen können Prozeduren und Objekte (Variable, Konstanten...) sein, die nach außen sichtbar sind.

Beispiel: Schnittstellen in Java

```
public class A { // nach außen sichtbare Klasse (Schnittstelle)
 public  int x; // nach außen sichtbare Variable (Schnittstelle)
 private int y; // nach außen unsichtbare Variable
 public  void f() { /* ... */ } /* nach außen sichtbare Prozedur
(Schnittstelle) */
 private void g() { /* ... */ } // nach außen unsichtbare Prozedur
}
class B { // nach außen unsichtbare Klasse
 /* ... */
}
```

3.4 Entwurfs- und Zerlegungstechniken

Es gibt zwei große Entwurfstechniken: Der zielgerichtete und der kompositionelle Entwurf.

Der zielgerichtete Entwurf ("top-down") sieht vor, dass man ein Problem in Teilproblem zerlegt. Jedes dieser Teilprobleme zerlegt man wieder in Teilprobleme und so weiter, bis man schließlich Elementare Anweisungen hat. Falls dies nicht klappt und man vor einem

Problem steht, das nicht bewältigbar, aber auch nicht mehr zerteilbar ist, so versucht man erneut alles zu zerteilen, jedoch diesmal in andere Teile.

Der kompositionelle Entwurf ("bottom-up") geht davon aus, dass man viele kleine Module programmiert, und diese dann zu immer größeren Prozeduren zusammen fasst, bis man die Aufgabenstellung erfüllt hat.

Es gibt auch verschiedene Zerlegungsarten, nach denen man unterscheiden kann und die die Entwicklung eines Programms beeinflusst:

Funktionsorientierte oder prozedurorientierte Zerlegung. Im Mittelpunkt steht die Funktionalität des Programms, nach der das Programm in Module geteilt wird.

Datenorientierte Zerlegung. Bei dieser Zerlegung steht die Struktur und die Verarbeitung der Daten im Vordergrund.

Modulorientierte Zerlegung. Hierbei wird davon ausgegangen, dass Daten und Operationen zusammengehören und gemeinsam Module bilden, die miteinander kommunizieren.

Objektorientierte Zerlegung. Ähnlich wie bei der Modulorientierten sicht, werden hier Daten und Operationen zusammengefasst. Im Mittelpunkt stehen sogenannte Objekte, die die Wirklichkeit oder eine fiktive Welt im Programm abbilden sollen.

4. Objektorientierte Programmierung

4.1 Beschreibung

Die Objektorientierte Programmierung ist eine Methode um Programme durch Nachbildung der Realität besser zu strukturieren. Das heißt, dass Gegenstände wie z.B. Autos, Schachbretter, Wörter- und Adressbücher oder auch Personen unter möglichst genauer Angabe ihrer Eigenschaften und Fähigkeiten in programmierte Beschreibungen ihrer selbst umgeformt werden: hat man also vor ein Adressbuch in einer objektorientierten Sprache zu programmieren, wird darin wahrscheinlich jeweils ein Objekt für jede Kategorie (Familie, Freunde,…), für jede eingetragene Person, etc. vorkommen. Diese Objekte hätten dann bestimmte Eigenschaften oder Daten, z.B.:

- Person: Name, Adresse, Telefonnummer, …
- Kategorie: Bezeichnung, Detaillierte Beschreibung, …

Durch diese Art der Programmstrukturierung werden aufwendige und komplexe Programme überschaubarer und vor allem weniger fehleranfällig. Weiters bietet sie bequeme Möglichkeiten der Planung: der Programmierer kann gewissermaßen *aus der realen Welt abschreiben*, die Hauptarbeit ist das Ermitteln des Verhaltens der verschiedenen Objekte untereinander. Jedes der Objekte wird separat geplant, entwickelt und getestet. Sollten also Fehler im Programm auftauchen, sind diese schnell ausfindig zu machen da bekannt ist welches Objekt welche Funktionen ausführt und welche Eigenschaften es besitzt.

4.2 Objekte

Objekte können, wie bereits erwähnt, sowohl Daten(Eigenschaften) als auch Funktionen besitzen. Die objekteigenen Daten sind meist nicht direkt von außen zugänglich, sondern nur über das so genannte *Interface*. Das bedeutet, dass Objekte keinen Zugriff auf die Daten anderer Objekte haben. Dieser ist nur durch die, als von außen zugänglich deklarierten, Funktionen eines Objekts möglich, welche das *Interface* bilden.

Die aus dieser, auch *Encapsulation* oder *Data Hiding* genannten, Methode hervorgehenden Vorteile sind zahlreich. So ist es zum Beispiel möglich, dass ein Objekt Benutzername und Passwort einer Person als private Daten gespeichert hat, diese jedoch keinem anderen Objekt direkt zugänglich macht. Der Vergleich eines Passwortes mit dem richtigen, im Objekt gespeicherten, Passwort könnte dann über eine Funktion erfolgen die ein Wort annimmt und einen bestimmten Wert bei Übereinstimmung, einen anderen bei fehlender Übereinstimmung zurückgibt.

Objekte werden in der Programmierung mit *Klassen* beschrieben. Ein Beispiel für die Notation einer Klasse für ein Auto könnte in etwa so aussehen:

```
KLASSE AUTO{ //Anfang der Klassenbeschreibung

  //Anfang: Daten in Form von lokalen Variablen:

  var Modell = „Marke“;
  var Farbe = „Blau“;
  var Baujahr = „1982“;
  var Preis = „14599“;

  //Ende: Daten in Form von lokalen Variablen

  //Anfang: Interface
  //Funktionen/Methoden zum Erhalten/Ändern der Daten

    //Ausgabefunktionen
  function getModell(){
    -- Programmcode zur Ausgabe von Daten an andere Objekte --
  }

  function getFarbe(){
    -- Programmcode zur Ausgabe von Daten an andere Objekte --
  }
  function getBaujahr(){
    -- Programmcode zur Ausgabe von Daten an andere Objekte --

  }

  function getPreis(){
    -- Programmcode zur Ausgabe von Daten an andere Objekte --

  }


    //Eingabefunktionen
  function setModell(){
    -- Programmcode zur Änderung von Daten dieses Objekts --
  }

  function setFarbe(){}
```

```
    function setBaujahr(){}
    function setPreis(){}
    //Ende: Interface
} //Ende der Klassenbeschreibung
```

Klassen sind so zu sagen ein Entwurf eines Objekts, nach dessen Vorbild ein Programm beliebig viele Exemplare, also tatsächliche und separat behandelte Objekte, dieser Klasse erstellen kann. Diese Exemplare unterscheiden sich also nur durch die verschiedenen Werte die sie enthalten sowie durch eine, bei der Erstellung des Exemplars zugeteilten, Bezeichnung (*Identifier*).

„Klassen haben in objektorientierten Sprachen mehrere Aufgaben:

- Sie definieren Datentypen, die Deklarationen verwendet werden können.
- Sie bilden Mengen, deren Elemente die Objekte sind.
- Sie sind Module mit vergleichbaren Strukturierungseinheiten.
- Sie dienen als Mechanismen zur Erzeugung von Objekten („Objektfabriken“). [Rech03]

4.3 Methoden (Funktionen)

Methoden dienen zur Ausführung der Fähigkeiten und Funktionen eines Objekts. Die Klasse `AUTO` könnte zum Beispiel um eine Methode namens `move()` zur Veränderung der aktuellen Position eine Autos erweitert werden. Zu diesem Zweck müssten außerdem noch Datenfelder für die Position zusätzlich zu den bereits vorhandenen für Modell, Farbe, etc. hinzugefügt werden, z.B. x und y in einem zweidimensionalen Koordinatensystem, x, y, und z in einem dreidimensionalen. Der Prozess der Bewegung könnte folgendermaßen aussehen:

- *Über das Exemplar der Klasse AUTO mit dem Bezeichner meinAuto, welches sich auf Position (x=0, y=0) befindet, wird dessen Funktion* `move()` *mit folgenden Parametern aufgerufen: Achse=y, Distanz=5 Pixel, Richtung=1;*

- *Wurde die Methode korrekt ausgeführt, befindet sich meinAuto nun auf Position (x=0, y=5);*

Eine Deklaration der Funktion `move()` in, für viele gängige Sprachen, typischer Form kann so aussehen:

```
function move(Achse, Distanz, Richtung){

  if(!Achse || !Distanz || !Richtung){
  //wenn einer der Parameter fehlt...
    -- Funktionsabbruch --
  }else{  //sind jedoch alle Parameter gegeben...
    --Veränderung der Position      auf der angegebenen Achse
    um die
    angegebene Distanz in der angegebenen Richtung--
  }
}
```

Den Mechanismus des Aufrufens der Funktion eines Objekts nennt man auch *Nachricht*.

4.4 Vererbung

Einige Klassen unterscheiden sich nur wenig voneinander und haben somit Eigenschaften und Funktionen gemeinsam. Um dies zu verdeutlichen und um Programmcode zu sparen, gibt es die Möglichkeit der Vererbung. So kann zum Beispiel eine Klasse namens `FAHRZEUG` erstellt werden:

```
KLASSE FAHRZEUG{ //Anfang der Klassenbeschreibung

  //Positionsdaten: x- und y-Achse
  var posX = 0;
  var posY = 0;

  var Modell = „Marke“;
  var Farbe = „Blau“;
  var Baujahr = „1982“;
  var Preis = „14599“;

    //Ausgabefunktionen
  function getModell(){}
  function getFarbe(){}
  function getBaujahr(){}
  function getPreis(){}


    //Eingabefunktionen
  function setModell(){}
  function setFarbe(){}
```

```
    function setBaujahr(){}
    function setPreis(){}


    //Ende: Interface

  } //Ende der Klassenbeschreibung
```

Diese Klasse kann nun als Basis für andere Klassen, wie zum Beispiel `AUTO` und `FAHRRAD` dienen:

```
KLASSE AUTO erweitert FAHRZEUG{
//Anfang der Klassenbeschreibung

    var Kennzeichen = „1234567“;
    var Stadt = „Wien“;
    var Land = „Österreich“;

    function getKennzeichen(){}
    function getStadt(){}
    function getLand(){}

    function setKennzeichen(){}
    function setStadt(){}
    function setLand(){}

} //Ende der Klassenbeschreibung

KLASSE FAHRRAD erweitert FAHRZEUG{
//Anfang der Klassenbeschreibung

    var Rahmengroesse = „3“;

    function getRahmengroesse(){}
    function setRahmengroesse(){}

} //Ende der Klassenbeschreibung
```

Diese neue Version der Klasse `AUTO` ist aufgrund des Bestehens der *Basisklasse (base class)* `FAHRZEUG` um vieles kürzer, vereinigt aber, da sie `FAHRZEUG` *erweitert*, sowohl die Daten und Funktionen der Klasse `FAHRZEUG` (z.B. `Farbe`, `Baujahr`, `getModell()`, …) als auch die eigenen (z.B. Kennzeichen, getKennzeichen(), …) in sich. Exemplare der Klasse `FAHRZEUG` haben jedoch keinen Zugriff auf in der Klasse `AUTO` oder anderen *abgeleiteten Klassen (derived classes)* definierte Funktionen und Daten.

Außer `AUTO` wurde noch eine weitere abgeleitete Klasse erstellt: `FAHRRAD`. Diese Klasse besitzt wie `AUTO` alle Eigenschaften und Funktionen der Klasse `FAHRZEUG`, jedoch nicht die

von AUTO. Dies wäre nur der Fall, wenn die Klasse FAHRRAD die Klasse AUTO erweitern würde und diese, wie im Beispiel, die Klasse FAHRZEUG als Basis hätte.

4.5 Polymorphie (Vielgestaltigkeit)

Polymorphie kann in der Programmierung mehreres bedeuten. Zum besseren Verständnis müssen jedoch zuerst die Begriffe *typisierte Sprache* und *untypisierte Sprache* erklärt werden:

- Typisierte Sprachen: sind Sprachen, in denen Variablen bei ihrer Erstellung immer ein Typ (eine Klasse) zugeordnet werden muss, z.B. Java oder C++. Ein Beispiel:
  - ➔ `int x = 10;`
  - ➔ `String x = "abc";`
- Untypisierte Sprachen: sind Sprachen, in denen Variablen ohne bestimmten Typ erstellt werden können, wie z.B. Javascript. Ein Beispiel:
  - ➔ `x = 10;`
  - ➔ `x = "abc";`

Polymorphie kann nun z.B. bedeuten, dass eine Variable für Objekte mehrerer Klassen stehen kann. In Typisierten Sprachen ist das nur möglich, wenn das Objekt das der Variable zugewiesen wird ein Exemplar einer Klasse ist, die von der Klasse (also dem Typen) der Variable abgeleitet ist. Dies lässt sich am Beispiel der Klassen FAHRZEUG und AUTO darstellen:

```
//Erstellung der Variable vom Typ FAHRZEUG
FAHRZEUG meinFahrzeug;

// Erstellung des AUTO Objektes und
// Zuweisung zur Variable meinFahrzeug
meinFahrzeug = new AUTO();
```

Erklärung: FAHRZEUG ist die Basisklasse, von der AUTO abgeleitet wurde, besitzt also alle Daten die ein FAHRZEUG Objekt benötigt. Somit kann jeder Variable vom Typ FAHRZEUG ein Exemplar der Klasse AUTO, oder von Klassen die wiederum von AUTO abgeleitet wurden, zugewiesen werden.

Würden die Objekte direkt in Variablen gespeichert, könnte es dadurch zu Problemen kommen: erweiterte Klassen haben meist mehr Daten und größeren Speicherplatzbedarf als ihre Basisklassen, z.B. hat die Klasse AUTO das Datenfeld Kennzeichen, welches der

Klasse `FAHRZEUG` fehlt. Wenn also einer Variable ein Objekt zugeordnet wird, dessen Bedarf an Speicherplatz den der Variable zugeordneten Bereich übersteigt, kommt es zu einem Speicherverwaltungsfehler.

Aus diesem Grund sind Variablen vielmehr Zeiger auf Objekte, beinhalten also die Speicheradresse des jeweiligen Objekts.

In untypisierten Sprachen ist somit zwar das Programmieren bequemer, die Fehleranfälligkeit jedoch wesentlich höher. Dem wird in typisierten Sprachen durch größeren Aufwand bzw. Genauigkeit Abhilfe verschafft.

Eine weitere Form der Polymorphie ist die, dass (in typisierten Sprachen) verschiedene Versionen gleichnamiger Funktionen existieren können, sofern ihre Zuordnung eindeutig definiert ist, z.B. durch Zugehörigkeit zu einer Klasse. Dies könnte z.B. so aussehen: Die Klassen `AUTO` und `FAHRRAD` enthalten beide eine Methode namens `getObjectID()`, die in Form einer Zeichenkette eine Identifikation für Objekte dieser Klassen zurückgeben soll. Diese Identifikation könnte eine Kombination aus Erstellungsdatum, Name und einer Produktkennung, bei Autos ***at*** und bei Fahrrädern ***fa***, bestehen. Folglich unterscheiden sich die beiden Versionen von `getObjectID()` dadurch, dass der zurückgegebenen Identifikation eine jeweils andere Produktkennung beigefügt wird.

Die Funktion `getObjectID()` könnte ebenso gut bereits in der Klasse `FAHRZEUG` implementiert die von sowohl `AUTO` als auch `FAHRRAD` überschrieben oder erweitert wird. Mit der Erweiterung einer Funktion ist folgendes gemeint, wieder anhand unserer Beispielklassen erklärt: angenommen die Klasse `FAHRZEUG` enthält bereits die Funktion `getObjectID()`, welche zwar schon eine Identifikation, jedoch ohne eine Produktkennung zurückgibt. In den abgeleiteten Klassen `AUTO` und `FAHRRAD` müssten nun nur mehr Versionen der Funktion `getObjectID()` erstellt werden, die die Version der Super- bzw. Basisklasse `FAHRZEUG` aufrufen und dem Ergebnis die jeweilige Produktkennung hinzufügen, sowie das Ergebnis dieser Operation zurückgeben. Hier der Beispielcode dazu:

```
//Implementierung in der Klasse FAHRZEUG
KLASSE FAHRZEUG{
  function getObjectID(){

    -- Programmcode zur Erstellung und Rückgabe der ID --

  }
```

```
}

//Implementierung in den Klassen AUTO und FAHRRAD
KLASSE AUTO/FAHRRAD{

  function getObjectID(){

//Zuweisung des Ergebnisses der Funktion
//der Superklasse an die lokale Variable x
     (Typ: ZEICHENKETTE) x = super.getObjectID()
     -- Programmcode zum Hinzufügen der Produktkennung und
     Rückgabe d. ID(in der (Buffer-)Variable x gespeichert) --
  }
}
```

Erklärung: mit `super.Funktionsname` wird hier eine Funktion der Super- oder Basisklasse aufgerufen. `(Typ: ZEICHENKETTE)` wird hier der Typ der nachfolgenden Variable `x` definiert.

4.6 Zusätzliche Informationen

Setzt man die Einführung der Begriffe *Objekt* und *Klasse* mit dem Beginn der Objektorientiertheit in den Programmiersprachen gleich, so liegt dieser in den sechziger Jahren des 20. Jahrhunderts und zwar durch das Erschaffen der Sprache Simula. Die untenstehende Tabelle listet einige objektorientierte Programmiersprachen auf.

Sprache	Jahr	Eigenschaften
Smalltalk	1980	einfache Syntax, rein objektorientiert, auch elementare Daten werden als Objekte aufgefaßt, interaktive Programmentwicklung mit Browser und umfangreicher Bibliothek.
Object Pascal	1986	kleine Erweiterung von Pascal um die wichtigsten objektorientierten Sprachelemente.
C++	1986	Weiterentwicklung von C, immer wieder erweitert, statische und dynamische Objekte, überladene Operatoren, Generizität, mehrfache Vererbung.
Oberon	1987	Nachfolger von Modula-2, Typerweiterung als Mechanismus zur Implementierung von Vererbung.
Self	1987	rein objektorientiert, Prototypen statt Klassen, Delegation statt Vererbung.
Eiffel	1988	rein objektorientiert, mehrfache Vererbung, Zusicherungen, Vor- und Nachbedingungen, Ausnahmebehandlung.
Oberon-2	1991	Erweiterung von Oberon um typgebundene Prozeduren.
Java	1995	Vereinfachung von C++, für verteilte Programme (z.B. im Internet) geeignet, Sicherheitsmechanismen.

Abbildung 6: Objektorientierte Programmiersprachen mit Erscheinungsjahr [Rech03]

5. Funktionale Programmierung

Funktionales Programmieren ist das Programmieren allein mit Ausdrücken und Funktionen, d.h. ohne Zuweisungen und damit ohne Nebenwirkungen und ohne Alias-Effekt [Rech03].

5.1 Grundbegriffe

Beim funktionalen Programmieren wird mit mathematischen Formeln, Funktionen und Regeln (und an die Mathematik angelehnte Formeln, Funktionen und Regeln) programmiert.

Jede Variable (genau genommen jede Instanz einer Variablen) bekommt daher während der Programmausführung nur einmal einen Wert zugewiesen und behält diesen Wert bis zum Ende des Programms. So wie in der Mathematik, hat jede Variable einen Wert, der zwar beliebig ist, sich aber nicht ändert. Man spricht auch nicht vom Zuweisen eines Wertes an eine Variable, sondern vom Berechnen einer Variablen.

Beispiel einer mathematischen Funktion (kein Zwischenergebnis!):

```
x = f(g(a), h(b))
```

Beispiel einer Zwischenergebnisberechnung (Pascal). Variable `x` ändert ihren Wert:

```
x := x + 1;
```

Funktionale Programme haben keine *Nebenwirkungen*, also keine Zwischenergebnisse, und damit lassen sich die Funktionen sehr einfach parallelisieren. Sind die einzelnen Parameter einer Funktion errechnet (Errechnen parallel möglich!), kann auch die eigentliche Funktion ausgeführt werden. Dieses Vorgehen nennt man *strikte Auswertung*. *Bedarfsauswertung* (*lazy evaluation*) bedeutet, dass man zu erst die Funktion aufruft und nur, wenn es notwendig ist, die Argumente berechnet. Das hat den Vorteil, dass auch mit unendlichen Daten und Datenstrukturen vernünftig gerechnet werden kann, sofern das Ergebnis endlich ist. In der Praxis haben Programme ohne Nebenwirkungen auch Nachteile, weil dadurch keine Ausnahmen, beispielsweise I/O-Fehler, abgefangen werden können. Deshalb gibt es sogenannte *Monaden*, spezielle Sprachkonstrukte, die Nebenwirkungen ermöglichen.

Die meisten funktionalen Sprachen haben sehr strikte Typisierungen. Man kann hier jedoch zwischen *statischer Typisierung* – vollständige Typenprüfung beim Übersetzen – und *dynamischer Typisierung* – Typisierung zur Laufzeit – unterscheiden.

Man unterscheidet auch zwischen funktionalen Sprachen *erster Ordnung* und *höherer Ordnung*. Erster Ordnung bedeutet, dass Werte als Argumente an Funktionen weitergegeben werden und Werte als Resultatobjekt wieder zurückgegeben werden können. Höherer Ordnung bedeutet, dass auch Funktionen als Argumente übergeben werden können und Funktionen auch als Resultatobjekte wieder zurückgegeben werden können.

Beispiel einfacher Ordnung – Funktion `min` gibt die kleinere Zahl zurück:

```
min( 5 , 7 )
```

Ergebnis:

```
5
```

Beispiel mehrfacher Ordnung – Funktion `min` gibt die kleinere Funktion zurück:

```
min( x*x , x*x+1 )
```

Ergebnis:

```
x*x
```

		Strikte Auswertung	Bedarfsauswertung
Dynamisch typisiert	Erster Ordnung	Mathematic Erlang	
	Höherer Ordnung	Lisp, Scheme APL, FP	SASL
Statisch typisiert	Erster Ordnung	SISAL	Id
	Höherer Ordnung	ML, SML, Caml Hope	Miranda Haskell

Tabelle 1: Kategorisierung funktionaler Sprachen

5.2 Beispiel: Haskell

Nun folgen einige Beispiele einer Haskell orientierten Sprache, siehe [Rech03].

Algebraische Typen können folgendermaßen definiert werden. `Int` ist hierbei der vordefinierte Datentyp für Ganzzahlen Beispiel:

```
data Color    = Red | Green | Blue
data IntTree  = Empty | Branch Int IntTree IntTree
```

Hier kann eine Variable vom Typ `Color` die Werte `Red`, `Green` oder `Blue` annehmen. Eine Variable vom Typ `IntTree` kann also den Wert `Empty` oder `(Branch x l r)` ha-

ben. `Branch` ist in diesem Fall ein Konstruktor, der mit den Werten `x`, `l` und `r` befüllt wird (`x` vom Typ `Int`, `l` und `r` vom Typ `IntTree`). Dadurch erhält man eine rekursive Datenstruktur. Wie man sieht, kann man nicht benötigte Klammern weglassen; in Java würde der Konstruktor so aussehen: `Branch(Int x, IntTree l, IntTree r)`.

Vor definierte Typen könnten so definiert worden sein:

```
data Bool = True | False
data Char = chr(0) | ... | chr(255)
data Int  = MININT | ... | MAXINT
```

Um beispielsweise die Summenformel des Baumes zu definieren schreibt man:

```
sum Empty = 0
sum (Branch n l r) = n + sum l + sum r
```

Die vorherige Formel könnte man auch mit einer Gleichung und einer Fallunterscheidung schreiben:

```
sum tree = case tree of
     Empty          => 0
|    Branch n l r   => n + sum l + sum r
```

Literaturverzeichnis

[Dave02] Davereed, http://www.creighton.edu/~davereed/csc533.F02/Lectures/overview.pdf, http://www.creighton.edu/~davereed/csc533, Creighton, 2002

[Dijk76] Disjkstra, E.W.: A discipline of programming. Englewood Cliffs, N.J.: Prentice Hall 1976

[Hans98] H. R. Hansen (Hrsg.), Wirtschaftsinformatik 1, Lucius & Lucius, Stuttgart, 1998

[Hoar69] Hoare, C.A.R.: An axiomatic basis for computer programming. Communications of the ACM 12, 1969

[Rech03] P. Rechenberg, G. Pomberger (Hrsg.), Informatik-Handbuch, Carl Hanser Verlag, München, Wien, 2003

[Schi98] Herbert Schildt, C++: The Complete Reference, Third Edition, Osborne McGraw-Hill, Berkeley (Kalifornien), 1998

[Well01] Luke Welling and Laura Thomson, PHP and MySQL Web Development, Sams Publishing, Indianapolis (Indiana), 2001