Programmiersprachen

Inhaltsverzeichnis

Kapitel 1

Allgemeines

1.1 Einleitung

1.1.1 Die Vielfalt der Programmiersprachen

Es gibt tausende von Programmiersprachen, von denen die meisten allerdings nur von ihren Erfindern benutzt werden. Einerseits gibt es ältere Sprachen, die heute nicht mehr benutzt werden, wie z.B. Algol, andererseits werden uralte Sprachen wie Fortran oder Cobol heute noch stark genutzt.

Viele Programmiersprachen sind sich allerdings relativ ähnlich.

Einteilung in vier Hauptklassen („Paradigmen")

imperative Sprachen Zentrale Konzepte sind Variablen und Wertzuweisungen. Variablen bezeichnen Speicherplätze, deren Wert verändert werden kann.

funktionale Sprachen Zentrale Konzepte sind Werte und insbesondere Funktionswerte. Variablennamen bezeichnen Werte, es gibt keine Wertzuweisung.

objektorientierte Sprachen (OO–Sprachen) stehen den imperativen recht nahe. Zentrale Konzepte: Objekte und Methoden. Objekt = Stück Speicher + Methoden = Prozeduren, die auf dem Speicher arbeiten.

Bsp.: Zählerobjekt ist ein integer-Speicherplatz und 3 Methoden: Wert abfragen, Wert um 1 erhöhen und Wert auf 0 setzen.

Vgl.: eine Variable in einer imperativen Sprache hat 2 Methoden: lesen und schreiben.

Manche OO-Sprachen sind aus imperativen Sprachen entwickelt worden (z.B. C++ aus C), andere wurden neu entwickelt (z.B. Smalltalk).

logische Sprachen zentrales Konzept: Prädikate ($p(X_1, \ldots, X_n)$ n-stelliges Prädikat) werden mit Hilfe logischer Formeln definiert. Anfrage: Für welche Werte ist das Prädikat erfüllt?

Bsp.: Prädikat auf Listen: $app([a_1, \ldots, a_n], [b_1, \ldots, b_m], [c_1, \ldots, c_k])$ ist genau dann erfüllt, wenn gilt: $[a_1, \ldots, a_n, b_1, \ldots, b_m] = [c_1, \ldots, c_k]$

Mögliche Anfragen:

$$\begin{array}{lcl} app([1],[2,3],Z)? & \longrightarrow & Z = [1,2,3] \\ app(X,Y,[1,2])? & \longrightarrow & X = [], Y = [1,2] \text{ oder} \\ & & X = [1], Y = [2] \text{ oder } X = [1,2], Y = [] \end{array}$$

Andere Formen der Einteilung

- nach Anwendung, z.B. für numerische Berechnungen, für Text-/Listen- oder Mengenverarbeitung

  Ursprünglich war diese Einteilung viel wichtiger, heute haben sich diese speziellen Eigenschaften der Sprachen vermischt. Es gibt aber auch heute noch Sprachen für spezielle Anwendungen wie z.B. LaTeX oder PostScript.

- Einteilung in interaktiv und Batch-Betrieb. Beim Batch-Betrieb wird zuerst das gesamte Programm fertiggestellt und läuft erst nach Übersetzung. Bei interaktivem Betrieb wird das Programm Zeile für Zeile eingegeben und kann sofort getestet werden (BASIC, die meisten funktionalen Sprachen)

- Unterscheidung nach anderen Gesichtspunkten, wie z.B. getypt–ungetypt.

1.1.2 Was ist eine Programmiersprache?

Eine Programmiersprache ist ein System zur Spezifikation von Berechnungen, das von Maschinen und Menschen gut lesbar sein soll, um

- Programme in Teamarbeit zu erstellen
- Programme zu warten (Behebung von Fehlern, Anpassung an neue Anforderungen)
- Information lokal zu halten, d.h. Schutz vor unberechtigtem Zugriff
- Kompliziertes aus einfachen Bestandteilen aufzubauen

Ziel: Programme sollten verläßlich sein, verständlich, leicht zu planen, zu schreiben, zu lesen, zu testen und zu verifizieren.

1.1.3 Kriterien zum Sprachentwurf

Ausdruckskraft alles Wichtige soll ausdrückbar sein, zusammensetzbar aus einfachen Dingen, oder aber direkt eingebaut.

Einfachheit leicht zu lernen, zu lesen, zu benutzen. Verzicht auf schwer verständliche Konzepte.

Aber: wenn es zuwenig Konzepte gibt, braucht man komplizierte Konstruktionen, um den Rest aufzubauen.

Orthogonalität Grundkonzepte sind voneinander unabhängig und beliebig zusammensetzbar.

Aber: es gibt einfache Konzepte (z.B. if), die nicht unabhängig von den anderen sind und trotzdem bereitgestellt werden sollten.

Bibliotheken Die Sprache kann sehr klein sein, wenn es große Bibliotheken vordefinierter Funktionen gibt (C, Lisp).

Gefahren: Bibliotheken können unübersichtlich bzw. systemabhängig werden!

Implementierbarkeit Ziele: leicht zu implementieren, der Compiler soll schnell und klein sein und effizienten Code erzeugen. Dies steht in Konflikt mit der Ausdruckskraft.

Fehlervermeidung Es gibt Konstruktionen, die dazu einladen, Fehler zu machen.

Bsp.: in C ist `==` der Vergleichsoperator und `=` der Zuweisungsoperator. `if (x = y)` ist korrektes C!

Fehlermeldungen, Korrektur

klare, eindeutige und vollständige Definition der Sprache z.B. ist bei $E_1 + E_2$ die Auswertungsreihenfolge nicht definiert. Viele verschiedene Versionen derselben Sprache können ebenfalls zu Problemen führen.

1.1.4 Sprachdefinition und Beschreibung

Die Sprachdefinition ist eingeteilt in einen statischen Teil (Syntax), der festlegt, was korrekte (d.h. übersetzbare, ausführbare) Programme sind, und einen dynamischen Teil (Semantik), der sagt, was diese Programme tun, wenn sie ausgeführt werden.

1. Statischer Teil

   lexikalische Struktur z.B. wie bauen sich Namen oder Konstanten auf? Welche Rolle spielen Leerzeichen und Zeilenenden? Was sind Kommentare?

   Dafür gibt es formale Beschreibungen, aus denen ein Teil des Übersetzers (Scanner) automatisch generiert werden kann. (z.B. lex/flex)

   syntaktische Struktur welche Konstrukte gibt es? Wie sind sie auseinander aufgebaut?

   Beschreibbar durch kontextfreie Grammatik/BNF (Backus–Naur–Form). Mit entsprechenden Werkzeugen (yacc/bison) kann man automatisch Parser erzeugen, die die syntaktische Struktur erkennen.

   Kontextbedingungen z.B. jeder Typ jeder Variablen ist vorher festgelegt, hinter „if“ steht ein Ausdruck vom Typ „bool“.

2. Dynamische Semantik

   Leider gibt es noch keine einheitliche, lesbare formale Beschreibung.

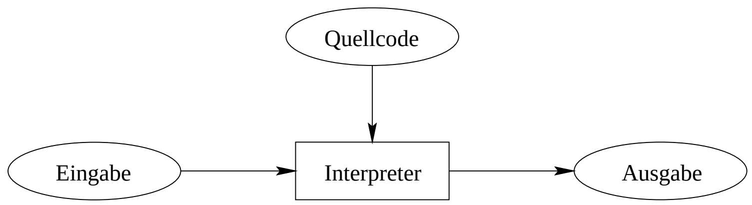
Ideal ist eine dreistufige Beschreibung der Sprache:

1. formale Definition für die Spezialisten
2. Reference Manual (natürlich sprachlich)
3. Tutorial, d.h. erläuterte Beispielprogramme

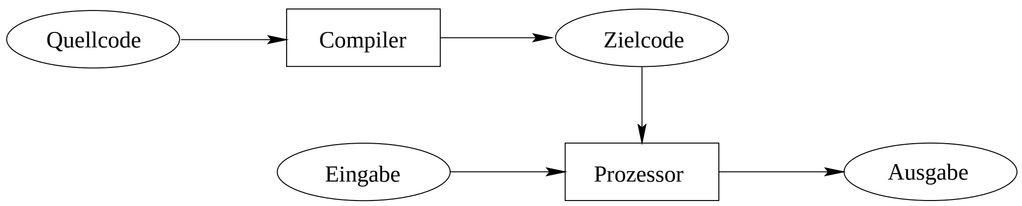
1.1.5 Implementierung von Programmier–Sprachen

2 Methoden: Programm in Quellcode kann interpretiert oder übersetzt werden.

Ein **Interpreter** liest das Quellprogramm und die Eingabedaten und führt das Programm aus



Ein **Compiler** übersetzt das Quellprogramm in einen Zielcode, der direkt vom Prozessor ausgeführt wird



Zwischenstufe: Compiler übersetzt nach Zwischencode, der nicht direkt ausführbar ist. Der Zwischencode kann interpretiert oder weiter übersetzt werden.

Vorteil des Zwischencode–Modells: Der (1.) Compiler ist nicht maschinenabhängig.
(z.Bsp.: Pascal $\rightarrow$ P–CODE oder JAVA $\rightarrow$ BYTE–CODE)

Vorteile der Compilierung:

- Compiler kann gewisse Fehler erkennen
- Compiler kann effizienteren Code erzeugen

Vorteile des Interpreters:

- Programme sind unmittelbar lauffähig

1.2 Geschichte der Programmier–Sprachen

1.2.1 Die 40'er Jahre: Vorläufer

Seit Ende der 40'er Jahre gibt es erste Digitalcomputer mit gespeicherten Programmen

- zuerst: Programme in Maschinensprache : Bestehen aus Zahlenkombinationen, die Instruktionen und Adressen im Speicher und Zahlenkonstanten angeben.
- *Assembler :*
  - Namen für Instruktionen und für Speicherplätze (Ursprung der Variablen)
  - Keine globale Struktur, deshalb schwer zu lesen
  - Maschinenabhängig, aber schnell

1.2.2 Die 50'er Jahre: Urväter

FORTRAN (1954-57 von IBM-Team unter John Backus entwickelt)
"Formula Translator"

Ziel: Compilierung mit Zielcode, der nur wenig langsamer als der beste handgeschriebene ist.

Motivation:

- Leichtes Programmieren
- Bessere Fehlerkorrektur

⇒ Erhöhung der Produktivität der Programmierer

Die Sprache hatte riesigen Erfolg und ist bis heute noch lebendig (mit gewissen Anpassungen in FORTRAN 77 und FORTRAN 90)

Eigenschaften:

- Ausdrücke wie `a*a-2*b`
- Schleifen (`WHILE`- und `FOR`-artig)
- `IF`, Sprünge (`goto`), Unterprogramme
- Felder
- Ein- / Ausgabeformate
- Variablen können deklariert werden, müssen aber nicht (Nicht deklarierte Variablen sind `INTEGER`, wenn 1. Buchstabe = I, J, K, L, M, N, ansonsten `REAL`)

Beispiel:

`C`	`Berechnung des Mittelwertes`	Kommentare beginnen mit `C`
	`DIMENSION A(99)`	Deklaration eines Feldes mit 99 Elementen vom Typ `REAL`
	`REAL MEAN`	
	`READ(1,5) N`	Es wird 1 Zahl eingelesen, die in `N` (`INTEGER`) abgespeichert wird
`5`	`FORMAT(I2)`	Die eingegebene Zahl hat 2 Stellen
	`READ(1,10)(A(I),I=1,N)`	Liest Zahlen `A(1)`,...,`A(N)` ein
`10`	`FORMAT(6F10.5)`	
	`SUM = 0.0`	
	`DO 15 I = 1,N`	15 ist die letzte Zeile der Schleife
`15`	`MEAN = SUM / FLOAT(N)`	
	`WRITE(1,25) MEAN`	
`25`	`FORMAT(8H,'MEAN =', F10.5)`	Erst String "MEAN =" ausgeben, dann die Zahl
	`STOP`	
	`END`	

Bemerkungen:

- alles in Großbuchstaben
- die Nummern sind symbolische Zeilennummern
- C in der 1., Zeilennummern ab der 2. und Text ab der 7. Spalte

COBOL (1959-60 von US-Navy-Team unter Grace Hopper)
"Common Business Oriented Language"

Motivation: nicht-numerische Datenverarbeitung in Betrieben und Verwaltung.

Neuerungen:

- Records
- symbolische Daten
- Strings
- genaue Beschreibung von Daten-Formaten
- einige sehr hohe Operationen wie Sortieren.

Aber: keine echten Unterprogramme → nur ein Namensraum

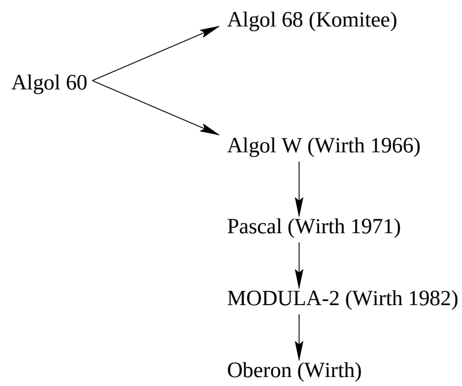
Ziel: Programme sollten auch für nicht-Cobol-Kenner lesbar sein. Daher ist COBOL sehr „geschwätzig". Das Ziel wurde trotzdem verfehlt, denn es gibt etwa 300 Schlüsselwörter!

COBOL wird bis heute stark benutzt, obwohl es sich kaum weiterentwickelt.

Algol 60 (1958-60 von einem akademischen Komitee entwickelt)
Erst AL genannt: „Algorithmic Language"

Ziel: allgemeine, ausdrucksstarke Sprache, in der man Algorithmen elegant zur Ausführung oder Publikation hinschreiben kann.

Algol hat sich praktisch nie durchgesetzt, aber theoretisch: Algol hat die späteren Sprachen sehr beeinflußt.



Eigenschaften von Algol 60:

- Formatfreiheit (keine speziellen Spalten mehr)
- strukturierte Anweisungen
- Blöcke mit lokalen Variablen
- Felder haben dynamische, aber dann feste Größe
- Rekursion (die zunächst nicht recht verstanden wurde)
- Ein- / Ausgabe leider undefiniert

Algol W: bereinigt und führt neue Konzepte ein: `CASE`, Trennung von `FOR` und `WHILE`, Records, Pointer, Strings (mit denen man aber nicht viel machen kann)
wenig benutzt, aber beliebt

Algol 68: formal und sauber, aber unverständlich definiert.
Vorteile:

- konsistente Struktur
- klar kombinierbare Konzepte
- nur wenig Konzepte

LISP (um 1968 von McCarthy)

Einheitliche Datenstruktur und S-Ausdrücke ($\approx$ Listen) für Daten *und* Programme.
Nachteil: Programme schwer lesbar
Vorteil: Programme können von Programmen leicht bearbeitet werden

Eigenschaften:

- Einführung der automatischen Speicherverwaltung (*Garbage Collection*)
- fast reine funktionale Sprache
- Nachkommen: FranzLisp, CommonLisp,…, Scheme

1.2.3 Die 60'er Jahre: Explosion

Hunderte von Sprachen, kaum eine hat sich durchgesetzt. (vgl. Algol 68, Algol W)

PL/I (1963/64 von IBM-Team entwickelt)
„Programming Language I"
Ziel: allgemeine, universelle Sprache, die alle anderen enthält: „Sprache, die das Ende aller Sprachen ist"

- von FORTRAN: Unterprogramme, Parametermechanismen, Ein- / Ausgabeformate, COMMON, ...
- von Algol: Block- / Kontrollstrukturen
- von COBOL: Records, Datenlayout, Record–Ein- und Ausgabe
- von Lisp: Listen, Speicherverwaltung
- dazu (neu): Ausnahmen (*Exceptions*)

⇒ riesige Sprache, die von kaum jemandem beherrscht wird. Der Compiler ist groß und langsam. Wurde von IBM gefördert, aber trotzdem wenig erfolgreich.

SNOBOL (Snobol 4: 1967)
Zur Stringverarbeitung, leistungsfähige Mustererkennung in Texten
(⇒ Einfluß auf Texteditoren)

SIMULA 67 (Nygoard, Dahl)
Ursprünglich für Simulationen (≈ Algol 60 + Klassen ⇒ erste OO-Sprache)

BASIC (Kemeny, Kurtz 1964)
Idee: äußerst einfach, leicht erlernbar, interaktiv, enthält Programmierumgebung
Sehr erfolgreich

1.2.4 Die 70'er Jahre: Trend zur Einfachheit

Pascal (N. Wirth 1971)
kleine und effiziente Sprache, sehr beliebt an Universitäten (als Einsteigersprache)
Vorteile:

- gutes Typsystem (da sicher und streng definiert)
- `set`- und `file`-Typ

Nachteile:

- keine getrennte Übersetzung
- kaum Stringverarbeitung
- nur Felder fester Länge
- wenig flexible Ein- und Ausgabe
- Syntax uneinheitlich

C (Ritchie 1972 bei Bell Labs)
Wurde entwickelt, um Unix implentieren zu können, bietet einen guten Zugriff auf die Maschine, ist schnell und einfach.
Die Programme könen sehr kurz gehalten werden :

`x++`	für	`x=x+1`
`x+=1`	für	`x=x+1`

Nachteil: Das Typsystem ist mager und unsicher (`POINTER` $\cong$ `INTEGER` $\cong$ `BOOLEAN`)

Euclid und **Mesa** haben Modulkonzepte.

1.2.5 Die 80'er Jahre: Imperative, Funktionale, OO und Logische Sprachen

Imperative Sprachen

ADA Auf Initiative des Verteidigungsministeriums 1975 - 80 von Komitee unter Leitung von Ichbiah entwickelt. Große und umfangreiche Sprache. („Das PL/I der 80'er")

Ideen:

- alles so allgemein wie möglich
- Datentypen (packages)
- Nebenläufigkeit (Prozesse, „tasks")
- Ausnahmebehandlung
- alles genau vorgeschrieben (ADA–Compiler müssen eine Reihe von Tests bestehen $\Rightarrow$ es hat lange gedauert, bis Compiler verfügbar waren! Was aber auch unter anderem daran lag, daß das Zusammenwirken der Konzepte zunächst unklar war.)

MODULA-2 (N. Wirth 1982 / 85 / 88)
Pascal–Nachfolger
Zusätze:

- Modul - Konzept
- klarere Syntax
- Prozedurvariablen
- Coroutinen

Relativ kleine Sprache, jedoch gerade deshalb in der Lehre sehr beliebt.

Funktionale Sprachen

ML („Meta Language" von Milner ab 1978)

- neue Syntax, die im Gegensatz zu Lisp lesbarer ist ($\Rightarrow$ Daten $\neq$ Programme)
- Typsystem mit polymorphen Typen
- es gibt noch Variablen mit Wertzuweisung, sind aber nicht mehr wichtig
- strikte Ausführung: alle Ausdrücke in Funktionsargumenten werden ausgewertet
- recht verbreitet

- Versionen: CAML (categoric abstract ML) und SML (standard ML)

Miranda (Turner 1986)

- rein funktional
- verzögerte Auswertung
  (z.Bsp.: `f(x)=5`: dieses Argument wird nie ausgewertet, da Ergebnis immer 5)
- potentiell unendliche Datenstrukturen
- Miranda ist klein und wird kaum benutzt
- Turner hat Copyright ⇒ niemand anders darf die Sprache implementieren

Haskell Neuer Sprachvorschlag für eine rein funktionale Sprache mit verzögerter Auswertung.
Neues Konzept: *Typklasse*

Gofer (M.P. Jones)
Frühe Implementierung einer Teilmenge von Haskell mit gewissen experimentellen Konzepten (z.Bsp.: Konstruktorklassen)

OO - Sprachen

Smalltalk (1972-80 Ingalls, Kay)
Neuentwurf einer völlig objektorientierten Sprache
Umfangreiche Laufzeit- und Entwicklungsumgebung

Eiffel (Mitte der 80'er, Meyer)
eher Pascal - ähnlich

C++ (1980 Stroustroup)
OO - Erweiterung von C
relativ verbreitet

Java (1990 Gosling, Joy von SUN)

- geschrumpftes C++, bereinigt und konsistenter gemacht
- Syntax noch sehr C-artig
- als Sprache des WWW propagiert: Programmstücke sollen über die ganze Welt verteilbar, kombinierbar und ausführbar sein
- soll auch robust und sicher sein
- große Zukunft vorhergesagt

Logische Sprachen

PROLOG (ab 1972 von Colmerauer in Marseille)

- seither in verschiedenen Versionen weiterentwickelt
- Uridee: Programming in Logic
- Endergebnis aber nicht sehr „logisch“
- Vorteile:
  - sehr wenig Konzepte

    - (Programme = Daten)
  - Nachteile:
    - ungetypt
    - nur auf kleinster Ebene strukturiert

Neue Vorschläge mit Typen, Modulen, Kombination mit funktionalen Sprachen

1.3 Lexikalisches

1.3.1 Der Zeichenvorrat

Welche Einzelzeichen kann man zum Aufbau des Programms benutzen?

Manche Sprachen, wie Algol 60 und APL ($\sim$ 1960) verwenden einen riesigen Zeichenvorrat (aber eher selten! APL: normale + griechische Buchstaben + mathematische Symbole).

Nachteil: Tastaturen und Bildschirme meist ungeeignet.

Ausweg: Ersatzvorstellungen, Spezialtastaturen

Häufigste Lösung: Verwende den gerade üblichen Zeichenvorrat

z.Bsp. Fortran: Die 64 Zeichen, die auf den Lochkarten vorkommen können: A-Z, 0-9, und ein paar andere. Kleinbuchstaben und [ ], { }, `< >` kommen nicht vor.

Später meistens: ASCII - Zeichensatz (95 druckbare Zeichen $\approx$ amerikanische Tastatur)

Ältere Variante: Kleinbuchstaben können benutzt werden, sind aber mit Großbuchstaben identisch (Pascal, Ada)

Jüngere Variante: Kleinbuchstaben $\neq$ Großbuchstaben (Algol 60, C, MODULA, Prolog, funktionale Sprachen)

Java: Unicode Zeichensatz $\supseteq$ ASCII - Satz (7-Bit)

Zeichenfolgen ersetzen nicht vorkommende Symbole:

Pascal	C	Ada	Modula	Fortran		
`<>`	`!=`	`/=`	`#`	`.NE.`	statt	$\neq$
`<=`	`<=`	`<=`	`<=`	`.LE.`	statt	$\leq$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

1.3.2 Bezeichner

Bezeichner dienen oft als Namen für Variablen, Unterprogramme, Typen, Konstanten, usw.

Oft: Folge von Buchstaben und Ziffern mit evtl. anderen Zeichen (), die mit einem Buchstaben anfängt. (Manchmal ist es wichtig, ob der 1. Buchstabe groß oder klein ist: Prolog: groß = Variable, klein = Unterprogramm oder Konstante)
Länge:

- älter: Maximallänge (Fortran: 6, Fortran '90: 31)
- jünger: „beliebig“ lang (meistens)

- Zwischenstufe: beliebig lang, aber nur die ersten paar Zeichen sind entscheidend (z.Bsp. bei bestehenden Pascal-Implementierungen 8)

Meist gibt es bestimmte Wörter, die wie Bezeichner aussehen, aber eine Sonderrolle spielen:

- *Vordefinierte Bezeichener* haben vorgegebene Bedeutung, können aber umdefiniert werden

  z.Bsp.: `true` und `false` in Pascal: `var true: boolean;`
  `true := false` ist erlaubt !!

- *Schlüsselwörter* haben in gewissen Kontexten eine spezielle Bedeutung, sind aber sonst normale Bezeichner (z.Bsp.: `WHILE` in Fotran: `WHILE = 3.7` ist eine normale Werzuweisung, während `WHILE x>y DO` eine Schleife einleitet)

- *reservierte Wörter* sind nicht als Variablennamen erlaubt (z.Bsp.: `if` in Pascal: `var if: boolean` ist verboten!!)

1.3.3 Leerzeichen

Moderne Regel: Leerzeichen trennen (z.Bsp.: `x 12` ist Bezeichner + Zahl, `x12` ist nur ein Bezeichner). Ein Leerzeichen ist so gut wie 1000.

Alte Regel (Fortran, Algol 60): Leerzeichen werden völlig ignoriert: `x 12 = x12`

Beispiel (berühmter Fehler in Fortran):
`DO 99 I=1,10` ist Anfang einer Schleife
`DO 99 I=1.10` ist eine Wertzuweisung ($\approx$ `DO99I=1.10` wird akzeptiert !)

1.3.4 Format

Modern: meist „freies Format“, d.h. beliebig viele Leerzeichen und Zeilenenden (zählen auch als Leerzeichen).

Alt: festes Format (Fortran, Cobol). Zeilenende beendet Anweisungen, bestimmte Sachen in bestimmten Spalten.

Moderne Idee: *„Abseitsregel“*
Einrückung definiert die Struktur, also ist die Zahl der Leerzeichen am Zeilenanfang nicht beliebig (Miranda, Haskell, Gofer)

1.3.5 Kommentare

Kommentare werden im allgemeinen vom Compiler ignoriert.

Typische Formen:

1. *Zeilenkommentar* beginnt mit spezieller Symbolfolge und endet am Ende der Zeile :

Gofer, Haskell, Ada	:	`--...`
Miranda	:	`||...`
Java, C++	:	`//...`
TeX, LaTeX, Prolog	:	`%...`

2. *geklammerter Kommentar* wird von 2 speziellen Zeichenfolgen eingeschlossen :

Pascal, Modula, ML	:	`(*...*)`
C, Java	:	`/*...*/`

Frage: Können Kommentare geschachtelt werden? $\left(\underbrace{\overbrace{(*\ldots\ldots(*\ldots\ldots*)}^{flach}\ldots\ldots*)}_{geschachtelt}\right)$

Frage: Trennen Kommentare ? Gilt `A(*...*)B` $\approx$ `A B` oder `AB`?
In TEX gilt
`A%...`
`B` $\approx$ `AB`

Kapitel 2

Imperative Programmiersprachen

2.1 Typen, Werte, Variablen

2.1.1 (Einfache) Typen

Überblick

Typ = Menge von möglichen Werten, auf denen es bestimmte Operationen gibt

Einteilung in einfache und zusammengesetzte Typen
Übliche einfache Typen:

- numerische (`integer, real`)
- Typ der Wahrheitswerte
- Typ der Zeichen

Dazu manchmal Unterbereichstypen (z.B. `0..9`) und Aufzählungstypen (z.B. {`rot, grün, blau`})
Zeigertypen sind in gewissem Sinne auch einfach (doch dazu später)

Unterteilungen:

fest eingebaut	-	benutzerdefiniert
diskrete Typen	-	nicht diskrete Typen

diskrete Typen (alle außer `real`):	alle Elemente (außer evtl. dem ersten und letzten) haben einen Nachfolger und einen Vorgänger
nicht diskrete Typen (`real`):	es gibt keine Vorgänger bzw. Nachfolger

Umfang eines Typs

Manche Typen stehen für unendliche Mengen, z.B. `integer` für $\mathbb{Z}$.
Dann wird in der Programmiersprache nur eine endliche Teilmenge davon realisiert (implementationsabhängig). Es gibt also einen kleinsten und einen größten `int`-Wert.

In manchen Sprachen gibt es mehrere verschieden große Typen,
z.B. in **C**: `short` $\subseteq$ `int` $\subseteq$ `long` (dürfen in Implementationen auch übereinstimmen)
Manchmal kann man die Grenze abfragen:

Pascal:	`maxint`
Modula-2:	`MAX(INTEGER)`
C:	`sizeof(Typ)` = Größe des Speicherplatzes für diesen Typ

Bei `real` ist die Lage noch schlimmer: Es gibt einen größten und einen kleinsten Wert, und die Präzision ist beschränkt.

Tabelle

Fortran	Pascal	Modula-2	Ada	C	Java	SML (funkt. Sprache)
`INTEGER`	`integer`	`INTEGER`	`integer` `short_integer` `long_integer`	`int` `short` `long`	`int` `short` `long`	`int`
unbekannt	*unbekannt*	`CARDINAL`	*unbekannt*	`unsigned`	–	–
`REAL` `DOUBLE PRECISION`	`real` –	`REAL` –	`float` `long_float`	`float` `double`	`float` `double`	`real` –
`LOGICAL`	`boolean`	`BOOLEAN`	`boolean`	–	`boolean`	`bool`
`CHARACTER`	`char`	`CHAR`	`character`	`char`	`char`	`char` (neuere Versionen)
–	–	–	–	(`char`)	`byte`	(`word8`,`word31`, `word32`)
`COMPLEX`	–	–	–	–	–	–

Wahrheitswerte: true, false

Oft vordefinierte Bezeichner, manchmal reservierte Wörter

Fortran:	`.FALSE.` und `.TRUE.`
C:	`0` $\cong$ false, alles andere (meist `1`) $\cong$ true

Operationen: $\wedge, \vee, \neg$ (and, or, not)
2 Auswertungsarten (vgl. 2. Übungsblatt)

Zeichentyp

meist implementationsabhängig, betrifft Umfang und Anordnung

ASCII 128 Zeichen, 'a'..'z' zusammenhängender Bereich

EBCDIC 256 Zeichen, zwischen 'a' und 'z' an 2 Stellen Nicht-Buchstaben eingestreut,
Bsp.: `('a'` $\leq$ `ch) and (ch` $\leq$ `'z')`

Ada und Java schreiben den Code vor:

Ada ASCII (7 Bit)

Java Unicode (16 Bit)

Zahlentypen

Operatoren +, -, $*$ normalerweise überladen:

int	+	int	→	int
real	+	real	→	real

Division:

Fortran, Ada, C nur "/" mit:	int `/` int → int	7 `/` 2 → 3
	real `/` real → real	7.0 `/` 2.0 → 3.5
Algol 60, Pascal:	int `div` int → int	7 `div` 2 → 3
	int `/` int → real	7 `/` 2 → 3.5
	real `/` real → real	7.0 `/` 2.0 → 3.5
Modula-2:	int `div` int → int	7 `div` 2 → 3
	real `/` real → real	7.0 `/` 2.0 → 3.5

real: meist Gleitkommazahlen, z.B. bei 32 Bit: $\pm 0, \ldots * 10^{\cdots}$
1 Bit Vorzeichen, 7 Bit Exponent, 24 Bit Nachkommazahlen ($\approx$ 7 Dezimalstellen)

In manchen Sprachen gibt es zusätzlich Festkommazahlen,
z.B. in PL/I: `declare x fixed decimal (6,3)`
Dezimalzahl mit 6 Vorkomma- und 3 Nachkommastellen

Weiterhin werden in manchen Sprachen `int`'s bei Bedarf automatisch in `real`'s umgewandelt,
z.B. 1 + 3.5 → 1.0 + 3.5 → 4.5,
in anderen Sprachen braucht man eine explizite Umwandlungsfunktion.

Unterbereichstypen

existieren in Pascal, Modula-2, Ada; **nicht** jedoch in C, Java

Bsp.: Pascal: `var a: 1..10`
Modula-2: `var a:` [`1..10`] *oder* `var a: INTEGER`[`1..10`]
Ada: **subtype** `T1` **is** `integer` **range** `1..10`
type `T2` **is new** `integer` **range** `1..10`
Unterschied:
`T1`-Werte können wie `integer`'s benutzt werden (`T1` + int → int)
`T2`-Werte sehen wie `integer`'s aus, sind aber keine (`T2` + int ist verboten)
(Es gibt automatische Konvertierungsfkt. `integer: T2` → `integer`)

Aufzählungstypen

Bsp.: Pascal ($\approx$ Modula-2, $\approx$ C):
type `Tag=(Mo,Di,Mi,Do,Fr,Sa,So)`
⇒ `succ, pred`: Tag → Tag automatisch definiert
Vergleiche: $<, =$ usw.

Pascal, Modula-2, C: Elemente sind Bezeichner, dürfen nur in einem Typ vorkommen

Ada:

1. Aufzählungstypen dürfen überlappen
   Compiler bestimmen den Typ
   Konstrukt: Tag'Do
2. Elemente müssen nicht notwendigerweise Namen sein
   z.B. `hex = ('0',...,'9','A',...,'F')`

2.1.2 Bezeichner und Deklarationen

Deklarationen

Bezeichner (Namen) bezeichnen Konstanten, Variablen, Unterprogramme, Typen, ...
Die *Bindung* von einem Bezeichner an das, was er bezeichnet, wird durch eine *Deklaration* festgelegt.
In modernen imperativen und OO-Sprachen müssen alle Bezeichner deklariert werden. Das kann auch in einer Bibliothek erfolgen (in vielen funktionalen Sprachen sind Deklarationen unnötig).
Typisches Aussehen einer Variablendeklaration:

Pascal, Modula-2 (keine Initialisierung): **var** `Name:` Typ
Ada (Initialisierung möglich):

C, C++, Java (Initialisierung möglich): Typ Name [= Ausdruck]

Blockstruktur

Die meisten Programmiersprachen gliedern ihre Programme in Blöcke. Jeder Block besteht aus Deklarationen und Anweisungen, die zusammengehören.

Pascal usw.: Block = Unterprogramm; diese können geschachtelt sein

C usw.: Unterprogramme können nicht geschachtelt werden. Aber jedes Gebiet {...} bildet einen Block; diese können geschachtelt sein (auch ganze Datei und Unterprogramme bilden Blöcke).

⇒ **Pascal & C:** Blöcke können beliebig geschachtelt sein.

Meist (Pascal, Modula-2, C) stehen vorne im Block die Deklarationen, hinten die Anweisungen.
C++, Java: Deklarationen und Anweisungen können innerhalb desselben Blocks beliebig gemischt sein.
Pascal (≈ Modula-2):

```
program Beispiel(input,output);
var x, y: real;
procedure P;
  var y, z: integer;
  begin y := 1; z := 2; x := 1.7;
  end;
begin P; x := 2 * x; y := x;
end.
```

C mit Unterprogramm:

```
float x, y;
P()
  { int y, z;
    y = 1; z = 2;
    x = 1.7;
  }
main()
  { P(); x = 2 * x;
         y = x;
  }
```

C ohne Unterprogramm:

```
main()
  { float x, y;
    { int y, z;
      y = 1; z = 2;
      x = 1.7;
    }
    x = 2 * x;
    y = x;
  }
```

Unterscheide zwischen deklarierendem (in Deklarationen) und benutzendem (in Zuweisungen; egal ob links oder rechts) Vorkommen eines Namens.

lokale Vorkommen: im selben Block deklariert

nicht-lokale (=globale) Vorkommen: in einem umfassenden Block deklariert

Gültigkeits(GB)- und Sichtbarkeitsbereiche(SB)

Bsp.: Betrachte obiges „C-Programm ohne Unterprogramm":

z: `{ ... { int z ... z=2 ... } ... }`

GB = SB von "`int z`"

x: `{ float x ... { ... x=1.7 } ... x=2*x ... }`

GB = SB von "`float x`"

y: `{ float y ... { int y ... y=1 ...} ... y=x ... }`

GB von "`float y`"

GB = SB von "`int y`"

SB von "`float y`"

hier ist "`float y`" nicht sichtbar, sondern verdeckt von "`int y`"
⇒ **Sichtbarkeitslücke**

SB von "`float y`"

Bemerkungen:

- genau genommen geht es um Vorkommen von Deklarationen;
  wenn im Beispiel zweimal "`float y`" stünde, wäre die Situation auch nicht anders
- GB der Deklaration ≈ Block, in dem sie vorkommt, plus die umfaßten Blöcke
  - genauer: Wo fängt er an? Bei der Deklaration oder beim Blockanfang?
    Bsp.:

    geeignete Ausgabefunktion

    ```
    { int x=1; { int y=x; int x=2; write(y)}}
    ```

    GB = SB

    ⇒ **C:** GB beginnt mit Deklaration (zumindest beim *gcc*; implementierungsabhängig)
    Modula-2: GB beginnt am Blockanfang
    Bsp.:

    ```
    { int x=1; { int x=x+1; write(x); }}
    ```

    GB

    ⇒ Ergebnis ist eine unbestimmte Zahl (nicht initialisiert)
  - Wo hört der GB auf? Bei einer Neudeklaration im *selben* Block oder am Ende des Blocks (und Neudeklarationen sind verboten)?
    Bsp.: `{ int x; float x; ...}`
    ⇒ **C:** Neudeklarationen sind verboten
- Unterprogramme / Parameter:
  C: Unterprogramm erzeugt 2 Blöcke:

  ```
  float f(int n) { float n = 1.0; return n; }
  ```

  (gcc Trier: Warnung, kein Fehler!)
  Ada (ohne Gewähr für Syntax):

  ```
  procedure P; var x: integer;
    procedure Q; var x: real;
    begin ... x ... P.x ... end
  begin ... x ... end
  ```

Statische $\leftrightarrow$ ***Dynamische Bindung***

Begriffe: Blockanfang / -ende, umfaßter Block usw.
Können auf 2 Arten interpretiert werden:

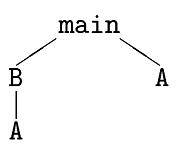
- im Raum (d.h. im Programmtext) $\Rightarrow$ *statische Bindung*
- in der Zeit (d.h. während des Programmablaufs) $\Rightarrow$ *dynamische Bindung*

(bisher wurde statische Bindung angenommen!)

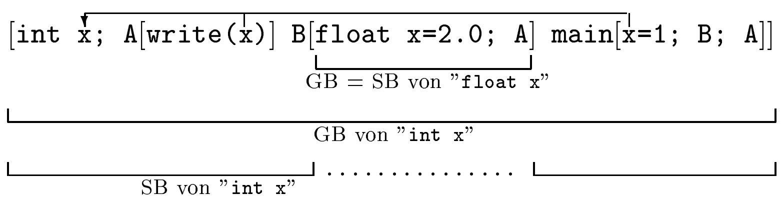
Bsp. C ($\approx$ Pascal, Modula-2, Ada):

```
int x;
void A()
 { write(x);}
void B()
 { float x=2.0; A();}
main()
 { x=1; B(); A();}
```

Ablauf:

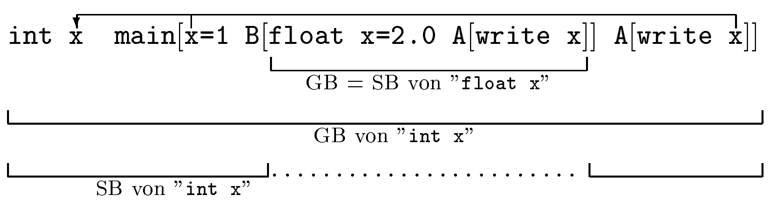


statische Blockstruktur (Anfang/Ende; [,]):



$\Rightarrow$ Ausgabe: 1 1

dynamische Blockstruktur :



$\Rightarrow$ Ausgabe: 2.0 1

Bemerkung: Oft falsche Erklärung: x bezieht sich auf zeitlich letzte vorhergehende Deklaration (stimmt nicht, da Blockstruktur nicht beachtet; vgl. Bsp.)

Bindungsarten:

- **C, Pascal, Modula-2, Ada:** statische Bindung
- frühes **LISP:** dynamische Bindung
  aber
- spätes **LISP, Scheme:** statische Bindung
- **Java:** Variablen werden statisch gebunden, Methoden (Unterprogramme) dynamisch
- **TEX/LATEX:** dynamische Bindung
  Bsp. in TEX:
  ```
  \def\x{1}   \def\A{{␣\x}}
  \def\B{{    \def\x{2.0}\A}}
  {\B\A}
  ```
  TEXanwenden ⇒ dvi-File ⇒ Ausgaben: 2.0 1

  zur Syntax: Bezeichner werden mit \ eingeleitet
  `\def`*Name*{*Rumpf*} ≅ Name = Rumpf *ohne* Blockbildung
  `\def`*name*{{*Rumpf*}} ≅ Name = {Rumpf} *mit* Blockbildung

  (mit nur 1 Klammer ⇒ Ausgabe: 2.0 2.0)

Vorteil statischer Bindung:

- Jedes Vorkommen eines Namens im Programm bezieht sich auf eine eindeutige Deklaration, dem ein eindeutiger Typ zugeordnet werden kann.

Dynamische Bindung:

- z.B. `x` in `write(x)` kann sich auf beliebige Typen (bspw. `int, float`) beziehen
- dynamische Bindung widerspricht statischem Typsystem
  {`T x; A();`}: bei dynamischer Bindung hat `x` in `write(x)` Typ `T`

Vorteile dynamischer Bindung:

- höhere Flexibilität
- Software besser wiederverwendbar

Beispiel: *Programmbibliothek*

`{...Q{...}...P{...Q(...)...}}`

P ruft Hilfsfunktion Q auf.

Angenommen, einem Benutzer gefällt nicht, was P tut; die Unzufriedenheit beschränkt sich auf die von Q erledigte Teilaufgabe:

`P` aus Bibliotek;
ruft jetzt dieses neue `Q` auf

dynamische Bindung: `mein_P{Q{` $\underbrace{\texttt{........}}_{\texttt{neue Def. von Q}}$ `}P}`

statische Bindung: `mein_P{Q{...}` $\underbrace{\texttt{..............}}_{\texttt{Kopie des Rumpfes von P aus Bibliothek}}$ `}`

Nachteil: Ein Benutzer von P, der P's Implementierung nicht kennt und nichts von Q weiß, kann versehentlich schreiben: {`Q`{...}`P`}

neue Def. von `Q`

bei dynamischer Bindung tut `P` etwas ganz anderes als erwartet

Statisches ↔ Dynamisches Typsystem:

statisches Typsystem: Compiler kennt Typen aller vorkommenden Namen und kann entsprechend Speicherplatz anlegen sowie effizienten Code erzeugen,
z.B. `x := y`: Typfehler werden vom Compiler erkannt
→ unvereinbar mit dynamischer Bindung

Scheme hat zwar *statische Bindung*, aber ein **dynamisches Typsystem**; zur Laufzeit haben die im Programm vorkommenden Werte einen Typ.
Es gibt auch Typfehler, die jedoch erst während der Ausführung erkannt werden (z.B. Liste + Zahl).
Ferner gibt es Typtestfunktionen: `number?`: Wert → bool (testet, ob Argument eine Zahl ist)

Manche Sprachen haben zwar ein statisches Typsystem, aber Schlupflöcher darin, z.B. *variante Records* in **Pascal**.
Funktionale Sprachen wie **SML, Miranda, Haskell** haben zwar ein statisches Typsystem, kommen aber (fast) ohne Deklarationen aus:

Beispiel SML: **val** `n = 10` ⇒ `n: int`

fun `double(x) = 2 * x` ⇒ `x: int; double: int` → `int`

`int`

fun `double'(x) = x + x`

Typ?

Ausweg ⇓

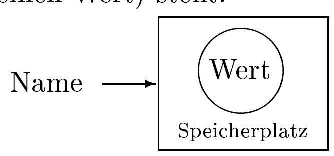
`double'(x: int)`

`int` × `int` → `int`
`real` × `real` → `real` ⇒ Abbruch

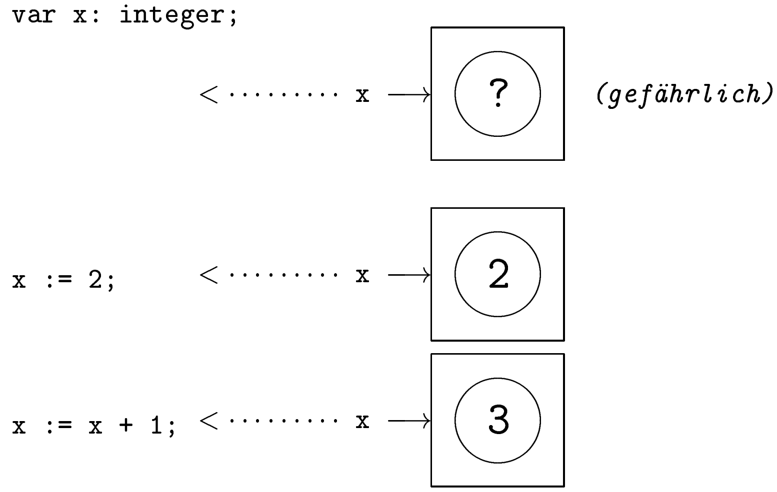
2.1.3 Variablen, Konstanten, Zeiger

Variablen

Ein Variablenname bezeichnet einen Speicherplatz, in dem ein Wert (bzw. eine Referenz auf einen Wert) steht.



Typische Situation:
z.B. in Pascal: `var x: integer;`



Speicherplatz wird vergeben mit der Deklaration, aufgegeben am Blockende;
diese Begriffe sind dynamisch zu verstehen (trotz *statischer Bindung*).

Die Zeit, während der der Speicherplatz vergeben ist, heißt **Lebensdauer** der Variable.
In Pascal, Modula, Ada, C, Java ist die Zuordnung Variablennamen → Deklaration statisch (statische Bindung). Bindung von Variablennamen an Speicherplatz ist dagegen dynamisch, wird immer am Blockanfang oder bei der Deklaration vorgenommen und beim Verlassen des Blocks aufgegeben.

Beispiel:
In der rekursiven Prozedur

```
p(){ int n; n... p()... }
```

erhält n mit jedem rekursiven Aufruf einen neuen Speicherplatz, das heißt „Inkarnation"; mit jedem rekursiven Aufruf wird einen neue Inkarnation der Variablen angelegt. Werden die Aufrufe hingegen hintereinander ausgeführt

```
p(){ int x; ... } main{ p();... p();... }
```

ergibt sich folgender Ablauf:

`main[p[x` erhält Speicher; Ende: Speicher von `x` wird freigegeben`]`
`p[x` erhält Speicher; am Ende wieder freigegeben`]]`

Unter glücklichen Umständen erhält x zweimal denselben Speicher und in der Zwischenzeit hat ‚keiner' darauf geschrieben. Dafür gibt es aber keine Garantie.

Ausweg in manchen Sprachen:

- in Algol60: *own-Variablen*
- in PL/I und C: *static–Variablen*, sie haben denselben GB und SB wie normale Variablen, aber die Lebensdauer ist anders: während des gesamten Programmablaufs ist fest ein Speicherplatz zugeordnet. Sie werden auch nur einmal initialisiert.

Beispiel in C:

```
p(){ static int n=0; ① n=n+1; ② printf("%d",n);}
main(){ ⓪ p(); ③ p();}
```

Ablauf **ohne** static: ⓪ `p[` ①, ② `1]` ③ `p[` ①, ② `1]`

n→ $\quad$ n→$\boxed{⓪}$ $\quad$ n→$\boxed{①}$ $\quad$ n→ $\quad$ n→$\boxed{⓪}$ $\quad$ n→$\boxed{①}$

Ablauf bf mit static: ⓪ `p[` ①, ② `1]` ③ `p[` ①, ② `2]`

n→$\boxed{⓪}$ $\quad$ n→$\boxed{⓪}$ $\quad$ n→$\boxed{①}$ $\quad$ n→$\boxed{①}$ $\quad$ n→$\boxed{①}$ $\quad$ n→$\boxed{②}$

Problem: Wann genau und in welcher Reihenfolge werden static–Variablen initialisiert?

Zuweisungen

Gewöhnlich wird die linke Seite einer Zuweisung zu Speicherplatz ausgewertet und die rechte zu einem Wert; der Wert wird in Platz geschrieben.

`x:=y` vorher: x→$\boxed{①}$ nachher: x→$\boxed{②}$

y→$\boxed{②}$ y→$\boxed{②}$

Von x interessiert: $\boxed{}$ (Speicherplatz, Referenz, L–Wert)
Von y interessiert: ② (gespeicherter bzw. referenzierter Wert, R–Wert)

Der Übergang von L–Wert zum R–Wert heißt *dereferenzieren*. In den meisten Sprachen wird automatisch (das heißt bei Bedarf) dereferenziert.
Ausnahme: SML (funktionale Sprache mit Wertzuweisung)

```
x := !y; x := !x + 1
```

explizites Dereferenzieren

ansonsten passiert dasselbe wie oben.

Andere Sicht: (Snobol, einige Lispdialekte)

`x:=y` vorher: x→$\boxed{①}$ nachher: x $\boxed{①}$

y→$\boxed{②}$ y→$\boxed{②}$

Konstanten

Der Konstantenname wird direkt an Wert gebunden:

Pascal: `Const n=10` n→⑩

Konstanten können nicht auf linker Seite einer Zuweisung stehen.

- Pascal hat *manifeste* Konstanten: In der Konstantendefinition dürfen keine Ausdrücke stehen, sondern nur eine einzelne Konstante. Compiler kann Konstanten bei der Codererzeugung direkt durch ihren Wert ersetzen.
- Modula hat *statische* Konstanten: Ausdrücke aus Konstanten und vordefinierten Operatoren sind erlaubt, z.B. `n-1`, wenn n Konstante. Können von Compiler ausgewertet werden.

- Ada hat *dynamische* Konstanten, d.h. beliebiger Ausdruck ist erlaubt.

  ```
  procedure P( x: in out integer) is
     c: constant integer := x
  ```

  Compiler weiß den Wert von c nicht. Daher muß der Wert ‚irgendwo' abgespeichert sein, aber dieser Speicherplatz spielt im Programm keine Rolle. Deshalb:

  c → Wert

  c ist direkt an Wert gebunden, `c:= ...` ist verboten!

Situation in SML

Allgemein in funktionalen Sprachen: die sogenannten „Variablen" sind eigentlich dynamische Konstanten.
SML: die üblichen „Variablen" sind dynamische Konstanten. Echte Variablen werden mit 'ref' eingeführt und sofort initialisiert.

Beispiel:

```
val n=1; val x=ref 2; val y=ref 3; val z=y;
```

es ergibt sich: n → 1   x → $\boxed{2}$ y → $\boxed{3}$ ← z

Verboten sind `n:=5; x:=y;`, dagegen erlaubt: `x:=3; x:=n; x:=!y`.

Nach `x:=!y` ergibt sich: x → $\boxed{3}$ y → $\boxed{3}$ ← z

Vergleiche:

`x=y` → `false`
`!x=!y` → `3=3` → `true`
`y=z` → `true` (derselbe Speicherplatz)

Nach `y:=4` ergibt sich: x → $\boxed{3}$ y → $\boxed{4}$ ← z   (auch !z ist 4)

Übliche Sprachen: haben keinen Typunterschied zwischen Konstanten und Variablen.
SML:

`n: int; z,y,x: ref int`
`!: ref T` → `T` ∀ Typen T ⟹ `y:ref int; !y:int`

`L := R`
Typ `ref T`   Typ `T`

Das heißt, `n:=5` (int → int) und `x:=y` (ref int → ref int) sind nicht erlaubt, `x:=n` (ref int → int) und `x:=!y` (ref int → int) sind erlaubt.

Es gibt auch ref(ref int): `val p=ref (ref 3)`   p → $\boxed{\circ}$ → $\boxed{3}$

Zeiger (Pointer)

Gab es noch nicht in Fortran, Cobol oder Algol60; meist in den jüngeren Sprachen, jedoch nicht in Java. Zeiger haben Werte, die Speicherplätze sind bzw. Referenzen.

p → $\boxed{\circ}$ → $\boxed{\circ}$ p: Pointer auf Integer

PLI:	Typ Pointer (auf was, wird nicht gesagt)
Pascal:	`^integer`
C:	`int *`
Modula:	POINTER TO INTEGER
Ada:	`access integer`

Vorsicht bei Deklarationen in C: '*' gehört nicht zum Typ, sondern zum deklarierten Namen:
`int *p,i;` $\widehat{=}$ `int *p; int i;`

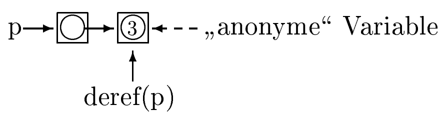
Nach Deklaration: p→$\boxed{?}$ sehr gefährlich

Außer Speicherplätzen gibt es noch einen speziellen Zeigerwert, der kein Speicherplatz ist:

Pascal:	`nil`
C:	`NULL`
Ada:	`null`
Modula:	NIL

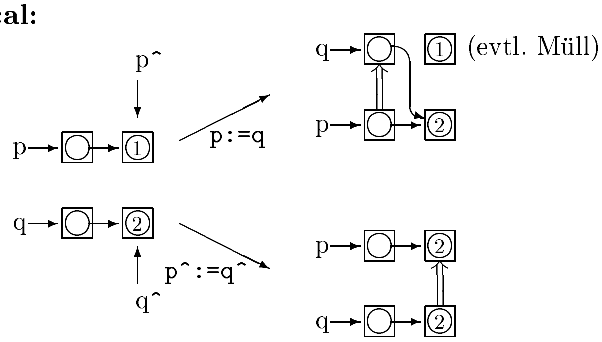
(In Ada wird mit null initialisiert. In SML gibt es kein analoges Konzept!)

Typische Operationen:
in Pascal, Modula, C, Ada: =, ≠ (untereinander bzw. mit nil oder null), dereferenzieren



Pascal:	`p^`
C:	`*p`
Ada:	`p.all`
Modula:	`p^`

Pascal:



Ursprung der anonymen Variablen:

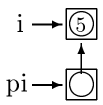
Pascal:	`new(p)` vorher: p→$\boxed{\circ}$ nachher: p→$\boxed{\circ}$→$\underbrace{\boxed{?}}_{\text{neu angelegt}}$
	die benötigte Größe wird aus Typ von gefolgert
Ada:	`p:=new integer`
Modula:	ein spezielles Modul stellt 'allocate' zur Verfügung:
	Allocate(p, SIZE(INTEGER)), umständlich und gefährlich
C:	`p=(int *) malloc(sizeof(int))`, ebenfalls umständlich und

gefährlich
Ausweg:
```
#define new(typ) ((typ *) malloc(typ)))
p=new(int)
```

In C gibt es mehr Möglichkeiten als in Pascal, Modula usw.

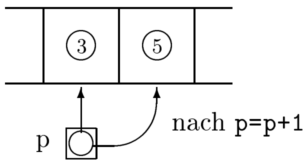
Beispiel:

```
int i=5; int *pi; pi=&i;
```
L–Wert von i (Speicherplatz)



danach sind `i` und `*pi` identisch; `*pi=7` ⇒ auch `i=7`

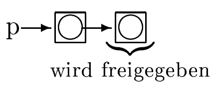
Pointer sind in C ähnlich Integers: '<', '+' und '-' sind definiert.



Speicherfreigabe: p sei Zeiger auf int

Pascal:	`dispose(p)`, Größe wird automatisch ermittelt
C:	`free(p)`, Größe wird automatisch ermittelt
Modula:	Deallocate(p,SIZE(INTEGER))

`free(p)`:



Ist sehr fehleranfällig, daher haben funktionale Sprachen und Java automatische Speicherfreigabe

2.2 Ausdrücke und Anweisungen

Die meisten imperativen Sprachen unterscheiden:

Ausdruck:	soll Wert ergeben (kann Speicher ändern als Seiteneffekt)
Anweisung:	soll Speicher ändern

Manche Sprachen haben nur Ausdrücke, keine Anweisungen: funktionale Sprachen (SML); Algol68. C, C++ und Java haben dagegen ungewöhnlich viele Ausdrücke (z.B. auch Zuweisungen), aber es gibt auch Anweisungen (die keine Ausdrücke sind).

2.2.1 Arithmetische und logische Ausdrücke

Sie bestehen aus Variablen, Konstanten, +, -, * ..., and, or ... meist als *Infix*: $E_1 + E_2$. Ausnahme Lisp: (+ 2 3) steht für 2+3 und (+ 2 3 4) steht für 2+3+4.

a) Priorität:

```
        3 + 4 * 5
       /         \
(3+4)*5    3+(4*5)
```

Pascal, Modula: hat drei Prioritätsstufen

```
* / DIV MOD AND
+ - OR
= # < > <= >=
```

Nachteil: `(a<b) AND (c<0)` muß geklammert werden

Ada: hat weitere Prioritätsstufen

```
* /
+ -
= < > ...
and
or
```

dann ist `a<b and c<0` möglich

b) Assoziativität:

```
        1 - 2 - 3
       /         \
(1-2)-3    1-(2-3)
```

Die meisten Operatoren sind linksassoziativ, in manchen Sprachen gibt es ein paar rechtsassoziative. *Nicht–assoziativ:* meist ist $a < b < c$ verboten!

c) unäre Operatoren: -x, not b; meist '-' unär und binär.
Ausnahme SML: ~x, aber x-y

d) benutzerdefinierte Operatoren: gibt es meist nicht; manche funktionale Sprachen erlauben sie (SML: <<<)

e) Typfragen: *Überladung*: int + int → int; real + real → real
automatische Umwandlung int → real in Pascal und C, in Modula nicht

f) bei logischen Ausdrücken:

false and b:	b braucht nicht ausgewertet zu werden (sogenannter *„short–cut“*)
true or b:	analog

Ada: 'and' und 'or' werten beide Argumente aus, 'and then' und 'or else' eventuell das zweite Argument nicht.

g) Auswertungsreihenfolge: $E_1 + E_2, f(E_1, E_2)$ was wird zuerst ausgewertet?

2.2.2 Zuweisungen

a) Allgemeines: Form: L:=R, aber in Fortran, C und Java: L=R. L wird zu L–Wert (Speicherplatz) und R zu R–Wert (Speicherinhalt) ausgewertet. Oft ist L eingeschränkt: nur Variable, Feld–, Pointer–, Record–Zugriffe!
Ausnahme SML: auf linker Seite darf beliebiger Ausdruck stehen (der auch Funktionsaufrufe beinhalten darf). Das Ergebnis muß nur ref–Wert sein.
Frage der Typanpassung: real:=int erlaubt? (vgl. Ausdrücke)

b) Wert einer Zuweisung (wenn sie ein Ausdruck ist):

Algol68:	Wert von L:=R ist der L–Wert von L.
C:	Wert von L=R ist der R–Wert von R.
	⇒ Mehrfachzuweisungen sind erlaubt:
	Algol68: x:=y:=z ≙ x:=(y:=z) ≙ y:=z; x:=y
	C: x=y=z ≙ x=(y=z) ≙ y=z; x=z
SML:	Wert von L:=R ist (), das ist das einzige Element des Typs *unit*
	⇒ Mehrfachzuweisung ist nicht sinnvoll.

c) Arithmetische Zuweisungen:

Modula:	INC(x,E) erhöht x um E
C:	x+=E steht für x=x+E;
	in derselben Weise werden '-=', '*=' usf. interpretiert
	x++ erhöht x um 1, liefert den alten x–Wert (ebenso x–)
	++x erhöht x um 1, liefert aber neuen x–Wert (–x)

2.2.3 Anweisungen ↔ Anweisungslisten

Sei S (statement) einzelne Anweisung und SL (statement list) Anweisungsfolge.

a) Bildung einer SL: oberflächlich betrachtet: $S_1; S_2; ...; S_n$

Pascal, Modula:	$S_1; ...; S_n$
	da es „leere Anweisung“ gibt
	⇒ $S_1; ...; S_n$; $(S_(n+1)$ leer)
C, Java:	S_1 $S_2 ... S_n$ **ohne** ';' dazwischen, diese gehören zu den Anweisungen, denn $L = R$ ist Ausdruck, $L = R$; aber eine Anweisung
	(ebenso $f(x)$ ist Ausdruck, $f(x)$; ist Anweisung)

b) Verbundanweisungen: SL → S

Pascal:	**begin** SL **end**	(kein Block)
C/Java:	{ SL}	(Block)

Man braucht diese Verbundanweisungen, wenn es Konstrukte gibt, die hinten offen sind, z.B. in Pascal: if B then S. Wenn alle Konstrukte abgeschlossen sind, unnötig: if B then SL end

c) Ausdruckslisten: In C ist E_1, E_2 ein Ausdruck (E_1 und E_2 sind Ausdrücke), E_1 wird ausgewertet, Wert weggeworfen, dann E_2 ausgewertet, sein Wert ist der des ganzen: `y=(x=1,3)`. In SML hat $E_1; E_2$ die Bedeutung wie oben (Beispiel: `(x:=1;y:=2;x+y)`

2.2.4 Bedingte Anweisungen

a) offen:

	Algol60, Pascal	C, Java
einseitig	**if** B **then** S	**if** (B) S
zweiseitig	**if** B **then** S_1 **else** S_2	**if** (B) S_1 **else** S_2

Problem: **if** B **then** **if** B' **then** S_1 **else** S_2: zu welchem if gehört das else?

Algol60: Situation verboten!
Pascal,C: else gehört zum letzten if davor, das noch kein else hat.

b) abgeschlossen:

	Modula
einseitig	**if** B **then** SL **end**
zweiseitig	bf if B **then** SL_1 **else** SL_2 **end**

(in Algol68: **fi**, in Ada: **end if**, in Fortran77: **endif**)
Nachteil: Bei **if** B_1 **then** S_1 **else if** B_2 **then** S_2 **else** ... **end** stehen sehr viele (so viele wie if's) end's am Ende der Anweisungen.
Ausweg: Ziehe **else** und **if** zusammen wie in Modula oder Ada bzw. **elif** in Algol68, das kein eigenes **end** hat.

c) bedingte Ausdrücke:
funktionale Sprachen: **if** B **then** E_1 **else** E_2 müssen zweiseitig sein; E_1 und E_2 müssen denselben Typ haben.
C: $B?E_1 : E_2$ rechtassoziativ, also $B_1?E_1 : B_2?E_2 : E_3$

d) arithmetisches if in Fortran:
if (E) L_1, L_2, L_3 bedeutet:
  if $E < 0$ then goto L_1
  else if $E = 0$ then goto L_2
  else if i$E > 0$ then goto L_3

2.2.5 Fallunterscheidung (erstmals AlgolW)

a) Pascal: **case** E **of** CL_1:S_1;...;CL_n:S_n **end**, wobei CL_i eine Liste von Konstanten $c_1, ..., c_k$ ist. E wird ausgewertet; wenn der Wert in Liste CL_i aufgeführt ist, wird S_i ausgeführt; dann hinter **end** weiter. CL_i müssen disjunkt sein. Ist der Wert in keiner Liste folgt ein Fehler.

b) Modula: etwas allgemeiner:
  CASE E OF CL_1:SL_1|...|CL_n:SL_n END
Vor END kann ELSE SL_{n+1} stehen. In den CL_i sind außerdem Aufzählungen erlaubt: z.B. 1,2..7,9

c) C/Java: `switch (`E`) {`$\widetilde{SL}$`}`. Vor den Anweisungen in $\widetilde{SL}$ kann `case` c: oder auch wiederholt `case` c_1`:` `case` c_2: stehen. Man läuft von der Einsprungstelle durch alle Fälle hindurch bis zum Ende oder bis zu `break`.
Modula:
CASE s OF 0: i:=0;j:=1 — 1: j:=1 — 2,3: k:=3 END
C:
```
switch(x){case 0:i=0;case 1: j=1;break;case 2:case 3: k=3;}
```

d) case–Ausdrücke in funktionalen Sprachen sind viel allgemeiner als die imperativen Konstrukte: Später!

2.2.6 Schleifen

a) *while–Schleifen*:

Pascal:	`while` B `do` S
Modula:	WHILE B DO SL END
Algol68:	`od`
C,Java:	`while` (B) S

Schema: $\longrightarrow B? \longrightarrow S \longrightarrow$ S wird ≥ 0–mal ausgeführt.

b) *repeat–Schleife*:

Pascal, Modula:	`repeat` SL `until` B
C:	`do` S `while` (B)
	analog, aber Bedingung anders herum!

Schema: $\longrightarrow SL \longrightarrow R? \longrightarrow$ SL wird ≥ 1–mal ausgeführt.

c) Aussprung aus Schleifen
In C/Java können alle Schleifen vorzeitig verlassen werden: `break;`
Modula/Ada: while und repeat können nicht vorzeitig verlassen werden, dafür gibt es eigene Schleifenart: `loop` S_l `end`
Modula: exit
`break/exit` können auch in if's vorkommen.
Ada: `exit when` B (`if` B `then exit end`)
Pascal: existiert nicht.

d) Gezählte Schleifen: Pascal-artig
Pascal: $for\ V :=\ E_1\ to\ E_2\ do\ S$
$for\ U :=\ E_1\ downto\ E_2\ do\ S$
Bedeutung: to: V nimmt nacheinander die Werte $E_1, E_1+1, \ldots, E_2$ an und für jeden Wert wird S ausgeführt. Bei $E_2 < E_1 :$ S wird nicht ausgeführt.
downto: $E_1, E_1 - 1, \ldots, E_2$.
Fortran: ursprünglich: S wird mindestens einmal ausgführt
Pascal: E_1, E_2 nur einmal ausgewertet (ganz am Anfang).
(Algol60: immer wieder ausgewertet)
Ada: 2 ähnliche Konstrukte.
Modula: $for\ U := E_1\ to\ E_2\ do\ S_l\ end$ ($\approx$ Pascal)
$for\ U := E_1\ to\ E_2\ by\ E_3\ do\ S_l\ end$
Bedeutung: U nimmt Werte $E_1, E_1+E_3, E_1+2E_3, \ldots$ an solange Wert$\leq E_2$ bei $E_3 > 0$ bzw. $\geq E_2$ bei $E_3 < 0$.
Viel diskutiert: welchen Wert hat V nach der Schleife ?

Pascal,Modula: undefiniert. Ada: for ist Block:
$\underbrace{for\ V\ in\ E_1\ \dots\ E_2\ loop\ S_l\ end\ loop}_{GB\ von\ Dekl.\ V:E_1 ... E_2}$

e) C-artig (C,Java)
$for\ (E_1; E_2; E_3)\ S \hat{=} E_1;\ while(E_2)\{S; E_3;\}$
z.B. $for(i = 1; i \leq n; i++)\ S \hat{=} for\ i = 1\ to\ n\ do\ S$
$for(;;)$ ist möglich $\hat{=}$ while(true)
Java: for ist Block, E_1 darf Deklaration sein
$\underbrace{for(int\ i = 1; i \leq n; i++)}_{GB\ von\ int\ i}\ S$

2.2.7 Sprünge

a) goto L ... L: ...
L kann Zahl oder Name sein (kann, in seltenen Fällen, sogar Variable sein)
Fortran sehr wichtig, heute meist eingeschränkt (z.B. Sprung in Block, else-Teil oder ähnliches verboten).
Modula, Java, funktionale Sprachen: kein goto mehr.

b) Aussprunganweisung: break,exit zum Verlassen von Schleifen.
C: break verläßt innerstes switch/while/do/for
Java: Konstrukte können mit Marke versehen werden, break L möglich
$l_1 : while(B_1)\{\dots while(B_2)\{\dots break \dots break l_1\} \dots\} \dots$
continue in c/Java: nächster Schleifendurchlauf beginnt sofort.
$while(B)\{\dots continue \dots break\}$
Java: auch continue l.
vgl. auch return/return E zum Verlassen eines Unterprogramms.

2.3 Unterprogramme und Parameter

meist 2 Formen: Prozedur $\hat{=}$ Anweisung
Funktion $\hat{=}$ Ausdruck (jedenfalls sein Aufruf)
Meist ist der Funktionsrumpf trotzdem eine Anweisung.
Prinzipielle Struktur:
Prozedurdefinition: Name + Liste der formalen Parameter(mit Typen) + Rumpf
Funktionsdefinition: genauso + Typ des Ergebnisses
Bsp:

Pascal: procedure p(...) Rumpf
funktion f(...): Typ Rumpf

Modula: genauso, aber immer 'procedure'

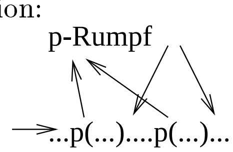
C: Funktion: Typ f(...) Rumpf
Prozedur void f(...) Rumpf

Aufruf: Name + Liste der Argumente (akt. Parameter) p(...)
Interaktion mit Umgebung über Parameter und nichtlokale Variablen
(Es gibt Sprachen, die das Aufzählen dieser Variablen an der Definition erfordern)

2.3.1 Kontrollfluß

Definition:

Aufruf:



Prozeduren:

Pascal: Verlassen nur am Ende des Rumpfes.

C/Modula: Am Ende des Rumpfes oder return.

Funktionen:

Pascal: Def $f(\ldots)begin \ldots f := E \ldots end$
Festlegen des Ergebnisses durch Zuweisung an Funktionsname; bewirkt kein Verlassen!

C/Modula: return E gibt Ergebnis und verläßt

fkt. Sprachen: Nur Funktionen, Rumpf ist Ausdruck

Makros: Definition sieht ähnlich aus wie bei Prozedur, aber Rumpf wird an Stelle der Aufrufe einkopiert.
$\ldots m(\ldots) \ldots m(\ldots) \ldots$
$\Rightarrow \ldots Rumpfvonm \ldots Rumpfvonm \ldots$

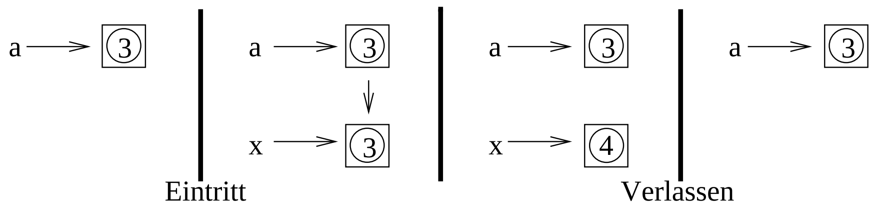
TeX: nur Makros

C: als Zusatzkommandos, die nicht zur Sprache gehören
$\#define \underbrace{Name(Par_1, Par_2, \ldots, Par_n)}_{keineLeerstelle}$ Rumpf
manchmal syntaktische Schwierigkeiten:
Funktion:
`int fac(int n)`
`int f=1; while(n>=2) f*(n--); return f;`
`m=fac(3);`
`#define fac(3)`$\{\ldots\}$
`m=fac(3);`$\Rightarrow m = \{\ldots\} Syntaxfehler$
Vorsicht: Makros geben dynamische Bindung.

2.3.2 Parameterübergabe

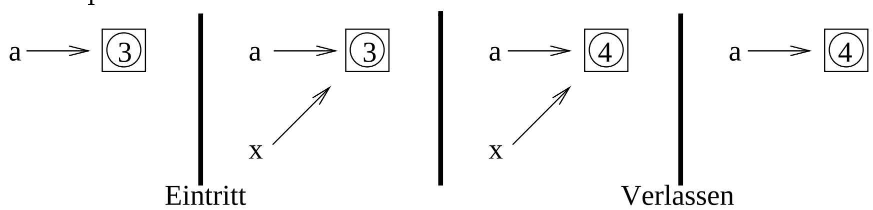
Im folgenden Parameter immer x

a) call by value (Pascal,Modula: Standard, C einzige Methode)
Argument: Ausdruck E
Eintritt: x erhält neuen Speicherplatz, der mit Wert von E initialisiert wird
Verlassen: Speicher wird freigegeben.
proc inc (x:int) begin x:=x+1 end … a:=3; inc(a)…

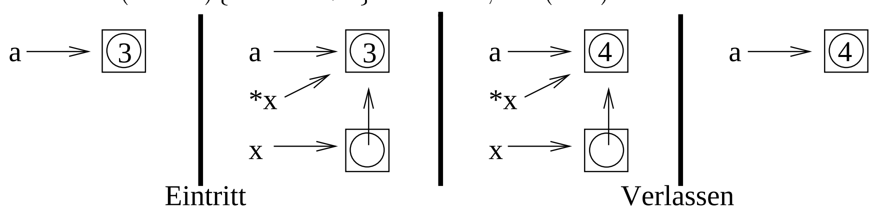


Vorteil: Zuweisung an den Parameter können außen nichts zerstören.(vgl. fac-Beispiel)

b) call by reference (Pascal,Modula: var ein Parameter)
Argument: ein spezieller Ausdruck, wie links von Zuweisungen, der zum L-Wert (Referenz, Speicherplatz) ausgewertet wird.
Eintritt: Parameter x wird an diesen Platz gebunden.
Verlassen: Bindung wird gelöst.
inc Bsp.:

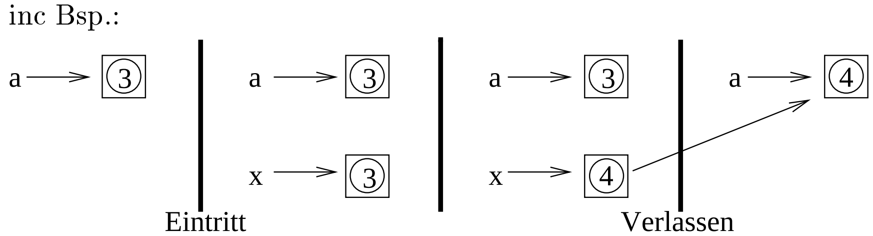


Was kann man in C tun ? Es gibt kein 'call by reference', aber man kann es simulieren.:
void inc (int *x){ *x=*x+1} ... a:=3; inc(& a)

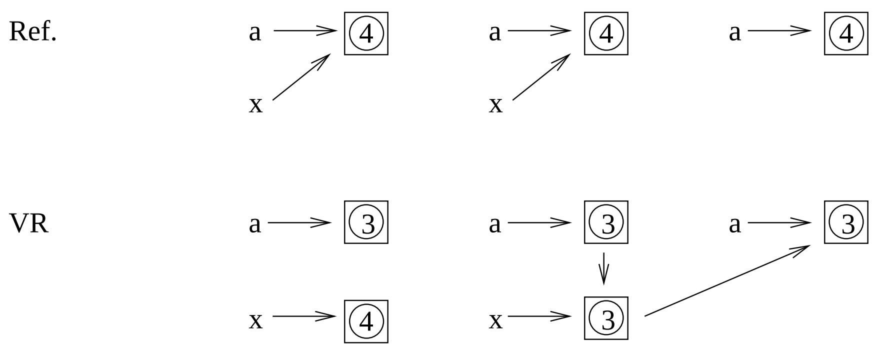


umständlich, * und & nie vergessen!

c) call by value-result (Fortran, Algol W, Ada: "in-out" Parameter))
Argument: spezieller Ausdruck mit L-Wert (Speicherplatz S)
Eintritt: Parameter x erhält neuen Speicherplatz, der mit Wert in S initialisiert wird
Verlassen: Wert von x wird nach S kopiert.
inc Bsp.:



Unterscheidung von call by reference:
- Argument im Rumpf benutzt:
p(x:int){ x:=x+1; x:=a;} a:=3 ... p(a)



- wenn 2 Parameter dasselbe Argument haben:
p(x,y:int) x:=x+1;y:=y+2 a=3 ... p(a,a)
Problem bei VR: Reihenfolge des Rückkopierens: anderes P(a[i])
Ada: Am Anfang a[i] zu Speicherplatz

f) call by text
Parameter wird an den Text des Argumentes gebunden. Immer, wenn x im Rumpf vorkommt, wird es durch diesen Text ersetzt. Typischerweise in Verbindung mit Makros (TeX, C-defines). Bei beiden geschieht das Ersetzen von textuell, unabhängig von syntaktischen Einheiten.
Bsp 1:\`# define sqr(x)  x*x`\\sqr(a+b) -¿ a+(b*a)+b
1/sqr(y) -¿ (1/y)*y
richtig: # define sqr(x) ((x)*(x))
dann: ((a+b)*(a+b))
1/((y)*(y))
Bsp 2:
p(x) {x:=3; i:=i+1; y:=x; }
p(a[i]) -> a[i]:=3; i:=i+1; y:=a[i] i.a. nicht mehr 3

g) call by name (Algol 60)
Ähnlich wie call by text, aber - Argument bildet syntaktische Einheit

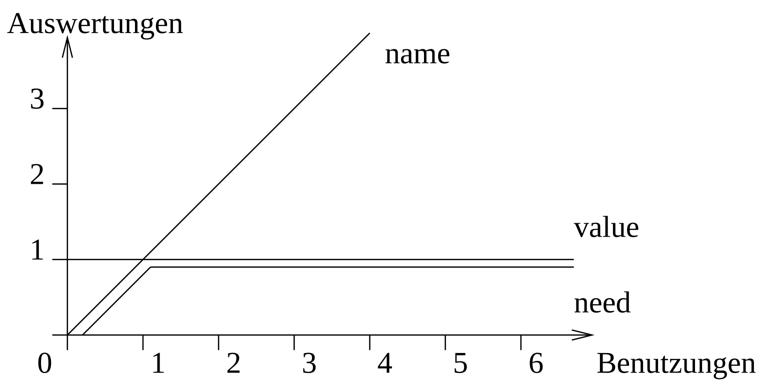
```
sqr(x){ return (x*x); }
sqr(a+b) -> (a+b)*(a+b)
```

- das ganze ist richtiges Unterprogramm, kein Makro
Wesentlicher Unterschied: bei 'call by name' werden noch an der Aufrufstelle die Namen im Argument fest ihren Deklarationen zugeordnet. Diese Zuordnung bleibt im Rumpf erhalten.

```
Bsp:     var i:int ; proc p(x:int) { var i:int ... x ...}
p(i+1) name         i+1
p(i+1) text         i+1
```

'name' erhält statische Bindung; 'text' gibt dynamische Bindung.
'call by name': x wird an Abschluß (closur) des Arguments (text) gebunden:
Text+Zuordnung $\rightarrow$ Deklaration

h) call by need (Miranda, Haskell, Gofer)
Das sind rein funktionale Sprachen, d.h. ein Ausdruck hat immer denselben Wert, wenn er mehrfach ausgewertet wird (refferentielle Transparenz) Begründung: keine Wertzuweisung. 'call by value' und 'call by name' habein unnötige Argumentauswertungen.



Argument wird höchstens einmal ausgewertet (Minimum der 2 Kurven).
Methode: Beim Aufruf wird Parameter x an Abschluß des Argument gebunden. Wenn im Rumpf das Argument zum ersten Mal gebraucht wird, wird der Abschluß ausgewertet und durch den Wert ersetzt. Wenn es zum zweiten Mal gebraucht wird, ist der Wert gleich da.(SML hat 'call by value' und Wertzuweisungen) .
Bsp:
i=s ... f(x,y)= ... ^{1} if x > 100 ^{2} then ^{3} x-100 else y
fac(n) = ...
f(fac(i),fac(i+1))
1.) x→ (fac(i), i=s,fac → ...)
y→ (fac(i+1), i=s,fac → ...,+ → ...)
2.) x→ 120
y→ wie oben
3.) x→ 120-100
y→ wie oben
f(0,fac(5))
1.) x→ 0
y→ (fac(s),fac→ ...)
4.) dieser Abschluß wird zurückgegeben! (nicht der Wert)

2.3.3 Weitere Besonderheiten

a) Überladung: In Ada, C++, Java können verschiedene Prozeduren den selben Namen haben und gleichzeitig sichtbar sein. Diese müssen verschiedene Anzahl oder Type von Parametern haben.
Bsp:
p(x:int) p(v:real) p(x:int,y:real)
p(3.4) p(4) p(3,3.7)

b) Benannte Parameterzuordung
üblich: Zuordnung Argumente → Parameter über Position
Def: $p(x_1, x_2, x_3)$

Aufruf:$p(a_1, a_2, a_3)$
Ada (und PL I): Zuordnung über Name des Parameters
Bsp: `proc inc(v:inout int; by: in int) is`
`begin v:=v+by end inc`
Mögliche Aufrufe: `inc(n,2), inc(v=>n,by=>2), inc(by=2,v=>n), inc(n,by=>2)`
(die ohne Namen stehen vorne, die mit hinten; die mit Namen können beliebig vertauscht

werden).
Nachteil für Prozedurschreiber: intelligente Namen müssen her. Die Namen sind außerhalb ihres GB sichtbar.
SML: es gibt ähnliches, aber nicht spezielle Syntax, sondern durch Benutzung von records.
Bsp: `fun inc (v:int ref, by:int)=(v:=!v+by)`
Aufruf: `inc(n,2) || inc: int ref*int->unit`
Andere Definition:
`fun inc {v:intref, by :int}=(v:=!v+by)` record
Typ: `{v:int ref, by:int}->unit` record mit 2 Elementen
`inc { v=n,by=2} oder inc { by=2,v=n }`

c) Defaultwerte
Ada: in-Parameterkönnen im Prozedurkopf initialisiert werden
`int(v:inout int; by: in int:=1) is ...` Aufruf: `inc(n,2)` by wird 2 ⇒ n:=n+2
`inc(n)` by wird 1 ⇒ n:=n+1

d) Mutuelle Rekursion
`p(char c){... q(c)...} q(char c){... p(c)...}`
p ruft q auf und umgekehrt. Algol, Fortran, Modula: kein Problem.
SML: `fun p(...)... ; fun q( ...)...` geht nicht;
`fun p(...)... and fun q( ...)...` geht;
Pascal: `procedure q(c:char); forward;`
`procedure q(c:char)...;`
`procedure q;...` (Parameter wird nicht wiederholt)
Ada: ähnlich, aber Parameter wird wiederholt.
C: ohne Vorwärtsverweis wird zwar akzeptiert, aber es wird `q(int)` beim Aufruf angenommen ⇒ Typprobleme.

```
void q(char); /* ohne Parametername */ void p(char c){...}
void q(char c){...}
```

2.3.4 Unterprogramme als Werte

a) Untergrogramm als Argument/Parameter (Fortran,Algol,Pascal,C,Modula,fkt. Sprachen)
Bsp: Pascal

```
function steigung( f(x:real):real;x1,x2:real):real
begin if x1=x2 then steigung:=0.0
else steigung:=(f(x2)-f(x1)/(x2-x1))
end
```

'steigung' hat drei Parameter, der erste ist eine Funktion (f)
Aufruf: 1.Argument muß ein Name einer vorher dekl. Funktion sein.
Bsp: `function gerade(x:real):real begin gerade:=2.0*x+1.0 end`
`steigung(gerade,1.0,3.0)`
SML hat auch funktionswertige Ausdrücke:
`steigung(fn x=> 2.0*x+1.0,1.0,3.0)`
auch `steigung(fn x=>a*x+b,1.0,3.0)` a und b können Variablen sein: `fun g(a,b)`
Pascal usw: Zahl und Art der Funktionen statisch.
fkt. Sprachen: dynamisch neue erzeugbar.

b) Trick in Ada: Unterprogramme können keine Parameter sein.

```
generic with function f(x:real) return real;
function steigung(x1,x2:real) return real is ...
```

bedeutet: steigung hängt von f ab. steigung nicht direkt benutzbar; wie Muster, von dem man Instanzen erzeugen kann.
`function geradensteigung is new steigung(f=>gerade)`
`function tansteigung is new steigung(f=>tan)`
`geradensteigung(1.0,2.0), tansteigung(0.0,0.2)`
(etwas umständlich; 'generic' hat noch andere Anwendungen (später))

c) als Variablenwerte (Modula), Prozedur/Funktion-wertige Variablen deklarierbar. Diesen können Prozeduren/Funktionen zugewiesen werden.

2.4 Module

2.4.1 Motivation

Eine grosse Programmieraufgabe soll in kleine Teile zerlegt werden, die unabhängig voneinander realisiert werden können. Die Teile sollen unabhängig compilierbar sein, die Typisierung soll prüfbar sein.
Lösung:
Zerlegung in Module; jedes Modul besteht aus 2 Teilen:

- Einer *Schnittstelle* nach aussen: gibt an, welche Typen vom Modul definiert werden und welche Operationen darauf mit den Typen
- Einem *Implementierungsteil*, der die in der Schnittstelle angekündigten Typen und Operationen implementiert und eventuell noch Hilfsoperationen dazu.

Im Idealfall kann man den Implementierungsteil durch einen anderen ersetzen, ohne daß der Rest des Programms geändert oder neu übersetzt werden müßte.

Sprache	Modula	Ada	SML	C
Schnittstelle	DEFINITION MODULE	package	signature	.h-files
Implementierung	IMPLEMENATION MODULE	package body	structure	.c-files

Beispiele mit sichtbaren Typen

Ziel: Typ `day` mit Operationen `next` und `prev`

Ada

Schnittstelle:

```
package days is
  type day is (Mo,Di,Mi,Do,Fr,Sa,So);
  function next (today:day) return day;
  function prev (today:day) return day;
end days;
```

Implementierung:

```
package body is
  function next ...
  function prev ...
end days;
```

Bemerkung:

- Typ `day` braucht hier nicht noch einmal definiert zu werden
- Die Zuordnung Implementierung $\longrightarrow$ Schnittstelle erfolgt über den Namen, der gleich ist. $\Longrightarrow$ Jede Schnittstelle kann nur eine Implementierung haben.

Benutzung:

- `day, next, prev` sind ohne besondere Vorkehrungen nicht außerhalb des Moduls benutzbar.
- `with days` $\Rightarrow$ als `days.day, days.next, days.prev` benutzbar (das hat gewissen Sinn, denn es dokumentiert den Ursprung der Operationen).
  ```
  var d1,d2 : days.day;
  d1:=days.Mo;
  d2:=days.next(d1);
  ```
- `with days; use days` $\Rightarrow$ `day, next, prev` direkt benutzbar (Compiler kann dieses `next` von evt. anderen am Argumenttyp unterscheiden).

Modula

Schnittstelle:

```
DEFINITION MODULE days;
EXPORT QUALIFIED day,next,prev;
TYPE day=(Mo,Di,Mi,Do,Fr,Sa,So);
PROCEDURE next(today:day) : day;
PROCEDURE prev(today:day) : day;
END days.
```

Implementierung:

```
IMPLEMENTATION MODULE days;
PROCEDURE next(today:day) : day; ...
PROCEDURE prev(today:day) : day; ...
END days.
```

Es gibt auch die Möglichkeit, beide Teile zu kombinieren:

```
MODULE days;
EXPORT QUALIFIED day,next,prev;
TYPE day=(Mo,Di,Mi,Do,Fr,Sa,So);
PROCEDURE next(today:day) : day; ...
PROCEDURE prev(today:day) : day; ...
END days.
```

Dies ist ähnlich wie Ada (Zuordung über Namen, der Typ muss nicht wiederholt werden). Unterschied: es wird nur nach aussen verfügbar, was in der Export-Liste steht.

Benutzung:

- `IMPORT days` $\Longrightarrow$ `days.day, days.next, days.prev`
- `FROM day IMPORT day, next` $\Longrightarrow$ `day, next` direkt benutzbar, `prev` nicht.

SML

Schnittstelle:

```
signature Days=sig
  datatype day=Mo|Di|Mi|Do|Fr|Sa|So
  val next: day->day
  val prev: day->day
end
```

Implementierung:

```
structure days:Days=struct
  datatype day=Mo|Di|Mi|Do|Fr|Sa|So
  fun prev ...
  fun next ...
end
```

Hier haben Schnittstelle und Implementierung verschiedene Namen, eine Zuordung erfolgt über die Typangabe. Man kann auf die Schnittstelle auch verzichten:
`structure days=struct ... end`
Dann werden alle in days definierten Namen exportiert. Wenn man eine Signatur angibt, dann werden nur die darin angegebenen Namen exportiert.

- Dieselbe Schnittstelle kann mehrere verschiedene Implementierungen haben.
- Die Typdefinition muß wiederholt werden.
- Namen sind sofort benutzbar als `days.day` (Name der Struktur) usw.
- `open days` $\Longrightarrow$ `day, prev, next` direkt benutzbar.

Beispiele mit versteckten Typen

Motivation

Angenommen, man braucht in einem Teil des Programmes einen Stack (Keller) von Zeichen mit Operationen `push, pop, empty`. Dafür soll ein Modell geschrieben werden. Es gibt zwei Möglichkeiten der Implementierung:

1. als Feld
2. als verkettete Liste

Man will die gewählte Implementierung nicht nach aussen bekanntgeben, weil sonst die Benutzer den Keller verändern könnten, ohne `push` und `pop` zu verwenden.

Modula

```
DEFINITION MODULE stacks;
EXPORT QUALIFIED charstack,push,pop,empty,init;
TYPE charstack;
PROCEDURE push ...
PROCEDURE pop(VAR s:charstack) : CHARACTER;
PROCEDURE empty ...
PROCEDURE init ...
END stacks.
```

Im 1. Fall könnte charstack ein Record aus Feld+integer sein.
Problem: geht in Modula nicht, charstack muß ein Pointertyp sein (ein Pointer auf so ein Record).

Nutzung:

```
VAR s1,s2 : charstack;
init(s1); init(s2);
```

`s1=s2` $\longleftarrow$ Pointervergleich
`s1:=s2` $\longleftarrow$ Unklar, ob durch pop(s1) auch s2 verändert wird oder nicht.

Abhilfe: Operationen assign und equal zur Verfügung stellen, aber die Benutzer können doch ‚=‘,‚:=‘ benutzen.

Ada

`type charstack is private` $\Longrightarrow$ versteckt,‚=‘,‚:=‘ benutzbar
`type charstack is limited private` $\Longrightarrow$ ‚=‘,‚:=‘ nicht benutzbar, ‚=‘ kann vom Modul selbst definiert werden, aber nicht ‚:=‘

Schnittstelle:

```
package stacks is
  type charstack is limited private;
  ...
  private type charstack is ...
end
```

Nachteil: Typdefinition steht in Schnittstelle, ist also nicht geheim. Aber trotzdem ist die Typdefinition nicht benutzbar.

Vorteil: Kann beliebiger Typ sein, nicht notwendig Pointertyp, z.B.

```
record inh:array ...
       anz:natural:=0 end
var s1,s2 : charstack;
```

$\Longrightarrow$ `anz` mit 0 initialisiert, kein `init(s1)` nötig.

SML

Schnittstelle:

```
signature Stacks=sig type charstack
  val empty:charstack->boolean
  val push :charstack*char->charstack (Erg. neuer Keller)
```

oder

```
  val push :charstack*char->unit} (Arg. ändert sich)
  val pop  :charstack->charstack*char
```

oder

```
  val pop  :charstack->char
  val emptystack : charstack
end
```

Mögliche Implementierungen:

- `charstack=char list`
- `charstack=(char list) ref`

Hier kann versteckter Typ beliebig sein.

```
datatype charstack=S of char list
val S=emptystack;
```

$\Longrightarrow$ nicht wirklich versteckt. Listenfunktionen sind auf den Stack anwendbar. S ist Konstruktor. (Das wird später im Teil über funktionale Sprachen erklärt.) Da S nicht in der Signatur steht, wird es nicht exportiert und es kann kein Wert vom Typ `charstack` konstruiert werden außer durch `emptystack, push` und `pop`.

Generische und polymorphe Module

Ein integer-Keller ist einem char-Keller sehr ähnlich. Kann man beide zusammen definieren? In Modula geht das zwar nicht, wohl aber in SML oder Ada:

SML

Sehr einfach, weil es in SML polymorphe Funktionen gibt, die auf beliebigen Typen arbeiten. Beispiel:

```
signature Stacks=sig
  type 'a Stack
  val emptystack: 'a stack
  val empty     : 'a stack->bool
```

```
  val push      : 'a stack*'a stack->'a stack
  val pop       : 'a stack->'a stack*'a
end

structure stacks:Stacks=struct
  datatype 'a stack:s of 'a list
end

val s=emptystack        (* s:'a stack   *)
val s1=push(s,'a')      (* s1:charstack *)
val s2=push(s,5)        (* s2=int stack *)
val s3=push(s1,7)       (* Typfehler!   *)
```

Ada

```
generic type elem is private;
package stacks is
  type stack is limited private;
  procedure push(s:in out stack;e:in elem);
  ...
  private type feld is array(1..100) of elem;
  type stack is record
                inh:feld;
                anz:natural:=0;
               end record
end
```

Vor der Benutzung müssen Instanzen gebildet werden:

```
package charstack is new stacks(item=>character);
package charstack is new stacks(integer);

with charstack, intstack;
use charstack, intstack;
var a:charstack.stack;
var s:instack.stack;
push(a,'x');
push(s,5);}
```

Der Compiler erkennt am Argumenttyp, welches `push` gemeint ist.
Nachteil : Jede Instanz hat seinen eigenen Code.

Geordnete Liste von Elementen eines beliebigen Typs

Ada:

```
generic type elem is private;
with function "<"(x,y : elem) return boolean;
package OList is ... end OList;
```

```
package OList1 is new OList(integer,"<");
package OList2 is new OList(real,">");
```

SML: Konzept der Funktoren

```
signature ORDERED=sig
  type elem
  val less:elem*elem->bool
end
signature OLIST=sig
  type Olist
  type elem
  val insert:elem*olist->olist
end
functor OList(Ordered:ORDERED):OLIST
  =struct type elem=Ordered.elem
  datatype olist=O type of elem list
  fun insert...
end
structure Ord1:ORDERED=struct
  elem=int
  val less=Int.<
end
structure OList1=OList(Ord1)
structure OList2=OList(struct type elem=real;val less=Real.> end)
```

`OList1.insert(1,OList1.insert(2,OList1.empty))` $\Longrightarrow$ O[1,2]
`OList2.insert(1.0,OList2.insert(2.0,OList2.empty))` $\Longrightarrow$ O[2.0,1.0]

2.4.2 Versteckte Variable

Angenommen, es soll im ganzen Programm nur einen einzigen Keller von Zeichen geben. Dann kann man den in der Implementation verstecken und den Typ der Operation vereinfachen.

Modula

```
DEFINITION MODULE Stack;
EXPORT QUALIFIED push,pop,empty;
PROCEDURE empty():BOOLEAN;
PROCEDURE push(c:CHARACTER);
PROCEDURE pop():CHARACTER;
END Stack.

IMPLEMENTATION MODULE Stack;
VAR inh:ARRAY[1..100] OF CHARACTER;
    anz:CARDINAL;
PROCEDURE empty();BEGIN RETURN(anz=0) END empty;
...
BEGIN
```

```
    anz:=0;
  END Stack.
```

Der Rumpf des Moduls wird am Anfang einmal ausgeführt und initialisiert.
Benutzung: `Stack.push(3)` bzw. `push(3)`.
Angenommen, man braucht doch einen zweiten Keller, dann hat man Pech gehabt.

Ada

```
generic type elem is private;
package Stack is ... end Stack;
package body Stack is ... end Stack;
```

Benutzung:

```
package s1 is new Stack(integer);
package s2 is new Stack(integer);
s1.push(3);
s2.push(5);
```

Hier kann man `s1` und `s2` nicht weglassen, da beides integer-Keller sind. Eine Auswahl kann nicht aufgrund eines Types erfolgen.

Kapitel 3

Von Modulen zu Klassen

3.1 Definition von Klassen

Ada ist objekt-basiert, zu objekt-orientiert fehlt allerdings die Vererbung. Diese ist Ziel des Ada-9X-Projektes.

b) In C++ entspricht `class` dem `generic package` in Ada. Instanzen entsprechen den Objekten bzw. den Objekt-Variablen. Beispiel:

```
class charstack {
  char inh[100];

  public:
  charstack() {anz=0;}
  boolean empty();
  void push(char);
  char pop();
}

boolean charstack::empty() {
  return (anz==0)
}
...
```

Kritik: Teile der Implementierung und die verborgenen Variablen sind mit der Schnittstelle gemischt.

Benutzung:

```
charstack s1,s2;
s1.push('x');
s2.push('y');
ch=s1.pop();
```

Klassen sind Muster für Objekte. Durch Deklaration oder `new` werden Objekte erzeugt.

b) Zeichenkeller mit Größe parametrisiert (C++):

```
class Charstack
{
  char ink[];    // ink ist Feld unbestimmter Groesse
  int  anz;
  public Charstack(int size)     // Konstruktor
  { ink = new char[size]; anz = 0; }
  ~Charstack()                   // Destruktor
  { delete ink; }
  ...            // wie letztes mal
}
```

Deklaration zum Beispiel: `Charstack a(10), b(20);` Dies ruft implizit die Konstruktoren `a.Charstack(10)` und `b.Charstack(20)` auf. Beim Verlassen des Blocks, in dem die Variablen deklariert wurden, werden implizit die Destruktoren `a.~Charstack()` und `b.~Charstack()` aufgerufen, die den Platz für `ink` wieder freigeben. Beim Beispiel mit fester Feldgröße brauchte man kein `new` und `delete`

c) Eine Klasse kann mehrere Konstruktoren haben (C++, Java). In Java gibt es zum Beispiel die vordefinierten Klassen String (unveränderliche Zeichenkette), StringBuffer (veränderliche Zeichenkette). String hat sieben Konstruktoren, Stringbuffer (mindestens) drei. Beispiel:

```
StringBuffer()
StringBuffer(int length)
StringBuffer(String s)

String()
String(String s)
String(char[] feld)
String(StringBuffer s)

String s1 = new String("Hallo");
String s2 = "hallo";
```

In Java und C++ gibt es folgende Regel: Wenn in einer Klasse C kein Konstruktor definiert ist, so gibt es implizit immer den Kostruktor `C()`. Java verschwindet das implizite `C()`, sobald mindestens ein Konstruktor explizit definiert ist. Konstruktoren und Destruktoren haben keinen Rückgabewert.

d) Das Beispiel Zeichenkeller in Java:

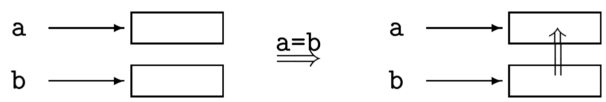
```
class Charstack
{
  char ink[];     // char ink[100] ginge nicht
  int anz;
  public Charstack(int size)
  { ink = new char[size]; anz = 0; }   // wie C++
  public Charstack()     // Ruft Charstack(100) auf, aber
  { this(100); }         // das darf man nicht hinschreiben
```

```
  public boolean empty()
  { return anz==0; }
    // Rumpf obligatorisch
  public push( ...
  public pop( ...
}
```

e) Vergleich: C++ – Java

- In Java gibt es keine expliziten Destruktoren. Der Speicherplatz wird automatisch freigegeben
- In C++ entspricht die Klassendefinition der Schnittstellendefinition, die Implementierung kann außerhalb erfolgen. In Java sind Klassendefinition und Implementierung eine Einhait. Es gibt aber spezielle Schnittstellendefinitionen (interfaces, siehe S. 59)
- In beiden Sprachen entsprechen die Objekte den üblichen Records. In C++ entspricht eine Objektvariable einer Recordvariable wie in Pascal, Modula-2 oder C. In Java entspricht eine Objektvariable jedoch einem Pointer auf ein Record. Beispiel (C++):

```
Charstack a,b; // legt Speicherplatz für zwei Objekte an
a = b; // Kopiert Objekt b nach a
```

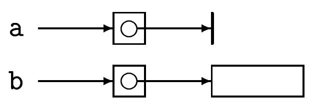


In C++ auch möglich:

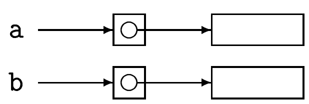
```
Charstack *c; // legt keinen Speicherplatz an, uninitial. Pointer
c = &a; // c enthält Pointer auf Objekt a
```

In Java:

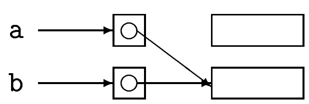
```
Charstack a,b;  // Legt keine Objekte an
a = null; b = new Charstack(17);  // moeglich
```




```
a = new Charstack(18);
```




```
a=b;
```



Deshalb gibt es in Java kein '*' und '&' mehr. Ein weiteres Beispiel:

```
Charstack a = new Charstack();
Charstack b = a;
a.push('X');
b.empty()   // ergibt false
```

Die C++ Entsprechung sieht folgendermaßen aus:

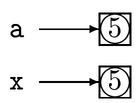
```
Charstack *a = new Charstack();
Charstack *b = a;
a->push('X');
b->empty()   // ergibt false
```

Umgekehrt halt folgendes C++-Fragment keine Entsprechung in Java:

```
Charstack a;
Charstack b = a;
a.push('X');
b.empty()   // ergibt true
```

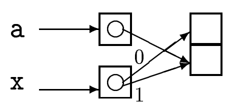
f) Werte und Typen in Java
Es existieren zwei Arten von Werten in Java: einfache (boolean, int, . . .) sowie Referenzwerte (Felder und Objekte, beachte: Strings sind Objekte). Die Parameterübergabe erfolgt eigentlcih immer "by value", aber ohne Folder und Objekte zu kopieren.

`p(int x)`
Aufruf: `p(a)`



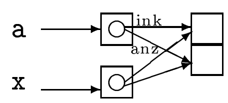
`x=0` veraendert Wert von `a` nicht

`p(int x[2])`
`p(a)`



`x[0]=5` verändert `a[0]`, `x=new int[2]` verändert `a` nicht

`p(Charstack x)`
`p(a)`



`x.push('3')` verändert `a`, `x=null` verändert `a` nicht

g) Verkette Listen in Java
Ziel des folgenden Beispiels ist es, einen Keller als verkettere Liste zu realisieren. In Modula-2 benötigt man dazu einen Record:

```
TYPE Node = RECORD
  ink : CHAR;
  next: POINTER TO Node;
END;
```

In Java gibt es keine Records, aber man kann Node als Klasse definieren unter Ausnutzung der Tatsache, das es sich bei Objektvariablen in Java stets nur um Pointer auf das jeweilige Objekt handelt:

```
class Node
{
  public char ink;
  public Node next;
}

class Charstack
```

```
    {
      Node head = null; // Initialisierung wird immer durchgefuehrt
                 // wenn ein neues Objekt dieser Klasse erzeugt wird
      public boolean empty()
      {
        return head == null;
      }

      public void push(char c)
      {
        Node aux = new Node();
        aux.ink  = c;
        aux.next = head;
        head     = aux;
}

      public char pop()
      {
        char c = head.ink;     // Laufzeitfehler bei leerer Liste
        head = head.next;
        return c;
      }
    }
```

Die push-Operation läßt sich auch elegeanter realisieren, wenn man in der Klasse Node einen Konstruktor definiert:

```
    public Node(char _ink, Node _next)
    {
      ink = _ink;
      next= _next;
    }
```

Oder:

```
    public Node(char _ink, Node _next)
    {
      this.ink  = _ink;
      this.next = _next;
    }
```

Dann:

```
push() {head = new Node(c,head);}
```

3.2 Vererbung

3.2.1 Beispiele

Als Beispiel für die Vererbung sollen geometrische Objekte wie Punkt oder Kreis als Klasse implementiert werden.

a) Punkt und Kreis in der Ebene, konvetionell

```
class Point
{
  float x;
  float y;
  public Point(float _x, float _y)
  { x = _x; y = _y; }
  public float X()
  { return x; }
  public float Y()
  { return y; }
  public void Move(float dx, float dy)
  { x += dx; y += dy; }
  public Print()
  { System.out.printf(x); System.out.printf(Y()); }
}
```

Um einen Kreis darzustellen, müssen Mittelpunkt und Radius gespeichert werden:

```
class Circle
{
  Point center;
  float r;
  public Circle(float _x, float _y, float _r)
  { center = new Point(_x, _y); r = _r; }
  public float R()
  { return r; }
  // analog zu Point
  public float X()
  { return x; }
  public float Y()
  { return y; }
  public void Move(float dx, float dy)
  { x += dx; y += dy; }
  public Print()
  { System.out.printf(x); System.out.printf(y);
    System.out.printf(r); }
}
```

b) Mit Vererbung läßt sich die Klasse Circle einfacher realiseren, da sie die Eigenschaften x und y sowie die Methoden X(), Y() und Move() unverändert von Point übernehmen kann. Sprechweise: Circle ist Unterklasse von Point und Point ist Oberklasse von Circle.

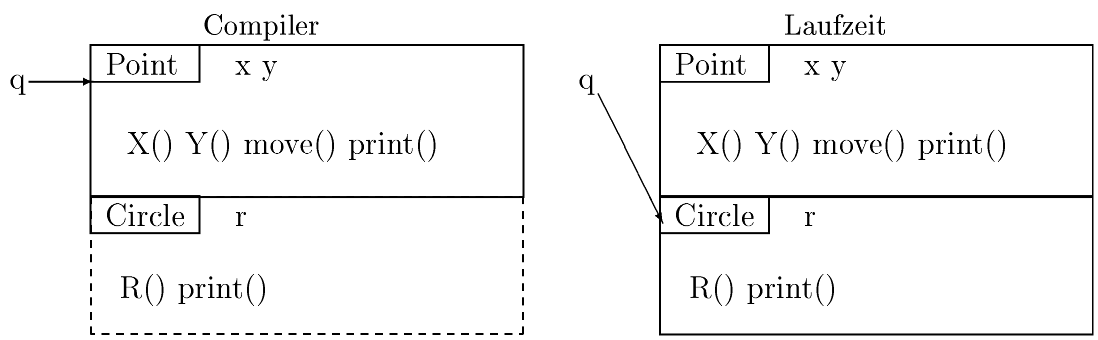
```
class Circle extends Point
{
  float r;   // x und y sind schon da
  public float R()
  { return r; }
  public void Print()
  { super.Print(); Systemout.printf(r); }
  public Circle(float x, float y, float r)
  { super(x,y); this.r := r;}
}
```

Das Schlüsselwort `super` bezeichnet in der Vererbungshierarchie die Oberklasse der jeweils aktuellen Klasse. `super.print` ruft in diesem Beispiel als die Methode `Print()` der übergeordneten Klasse `Point` von `Circle` auf, die x und y ausgibt. `super(x,y)` im Konstruktor Circle ruft analog den Konstruktor von Point auf.

Die Anwendung der beiden Klassen könnte wie folgt aussehen:

```
Point  p = new Point(1,2)    // Versteht Move, Print, X, Y
Circle c = new Circle(1,2,3) // Versteht Move, Print, X, Y, R
p = c; Point q = new Circle(1,2,3);  // erlaubt
c = p; Circle d = new Point(1,2);    // verboten
```

Für die Zuweisung `a = b` von Objekten gilt folgende Typregel: Die Klasse `a` muß Oberklasse der Klasse von `b` sein. Für Aufrufe von Funktionen gilt analog folgende Typregel: Die Klasse des Parameters muß Oberklasse von der des Arguments sein. Die Zulässigkeit eines Methodenaufrufs wird vom Compiler statisch anhand des Variablentyps geprüft. Ist die Variable `q` vom Typ `Point`, so sind die Methodenaufrufe `q.X(); q.Y(); q.Move(...); q.Print()` zulässig, nicht jedoch `q.R()`. Die Methodenauswahl (unter gleichen Methoden) erfolgt dynamisch zur Laufzeit durch das Objekt. So ergibt beispielsweise `q.Print()` die Ausgabe `1 2 3`, denn das von q bezeichnete Objekt ist ein Kreis und wählt das `Print()` von `Circle`.



Bei einem Methodenaufruf wird zur Laufzeit die bennante Methode zuerst der eigenen Klasse, dann in ihrer Oberklasse, dann in deren Oberklasse usw. gesucht..

c) Feinheiten zur Methodenauswahl. Beispiel:

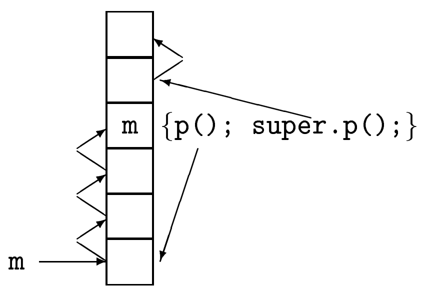
```
class A
{
  public void P() {print("a");}
}

class B extends A
{
  public void P() {print("B");}
  public void test() {this.P(); super.P();}
}

class C extends B
{
  public void P() {print("C");}    // erbt Methode test()
}

C c = new C();
c.P();             // Ausgabe: C
c.test();          // Ausgabe: C A
```

In der letzten Zeile hätte man eigentlich dies Ausgabe `B A` erwarten können. Die Ausgabe `C A` hat folgenden Grund: Wird aufgrund von Vererbung die Methode einer Oberklasse der aktuellen Klasse aufgerufen, und in dieser Methode kommt es zu einem weiteren Methodenaufruf, so wird mit der Suche nach dieser Methode wieder ganz unten in der Vererbunghierarchie bei der aktuellen Klasse begonnen. Im Beispiel wird also die Methode `test()` in der Klasse `B` gefunden, dann wird mit der Suche nach der Methode `P()` wieder in C begonnen, wo sie auch gefunden wird (Ausgabe: `C`). Danach wird durch `super.P()` explizit die Methode `P()` der Oberklasse von `B` (also `A.P()`) aufgerufen (Ausgabe: `A`).



Allgemeiner: `this` (oder "nichts") ist ganz dynamisch, `super` ist halb statisch. In C++ werden nur solche Methoden wie in Java dynamisch gebunden, die explizit mit dem Schlüsselwort `virtual` versehen werden.

3.2.2 Vererbung und Typsystem

Java hat per Definition ein statisches Typsystem. Aber wie paßt das zum Umdefinieren bzw. dynamischen Suchen von Methoden? Die Regeln:

Regel 1: Wenn in einer Unterklasse eine Methode m umdefiniert wird mit denselben Argumenttypen wie in der Oberklasse, dann muß auch der Ergebnistyp übereinstimmen (absolut gleich, nicht Unter- oder Oberklasse voneinander).

Regel 2: Methoden mit unterschiedlichen Argumenttypen gelten als verschiedene. Zwischen ihnen wählt der Compiler statisch aus (aufgrund von Variablen/Argumenttypen). Nur bei Methoden mit demselben Namen und denselben Argumenttypen gibt es dynamische Auswahl zur Laufzeit durch das Objekt.

Beispiel:

```
class A
{
  public void p(A x) {print("A");)}
}

class B extends A
{
  public void p(B x) {print("B");)}
}

class C extends B
{
  public void p(A x) {print("C");)}
}
```

Hier gibt es zwei Methoden `p(A)` und `p(B)`, zwischen denen der Compiler statisch wählt. Wenn der Compiler `p(A)` wählt, wird zur Laufzeit dynamisch eines der beiden `p(A)` ausgewählt.

```
A aa = new A(); B bb = new B(); C cc = new C();
A ab = bb; A ac = cc; B bc = cc;
```

	p(aa)	p(ab)	p(ac)	p(bb)	p(bc)	p(cc)	
aa.	A	A	A	A	A	A	Compiler: Adressat hat Typ A
ab.	A	A	A	A	A	A	⇒ existiert nur Methode p(A)
ac.	C	C	C	C	C	C	
bb.	A	A	A	B	B	B	Compiler: Adressat hat Typ B oder C
bc.	C	C	C	B	B	B	⇒ kennt Methode p(A) und p(B)
cc.	C	C	C	–	–	–	

In den drei linken Spalten hat das Argument immer den Typ `A` (paßt also nicht auf `p(B)`), deshalb wird vom Compiler `p(B)` ausgewählt. In den drei rechten Spalten ist sowohl `p(A)` als auch `p(B)` möglich. "–" bedeutet Compilerfehler, da keine Entscheidung möglich ist.

3.2.3 Typprobleme bei Vererbung

Als Beispiele sollen hier wieder die Klassen `Point` und `Circle` aus dem vorherigen Kapitel dienen. Die Klasse `Point` habe folgende zusätzliche Methode, die das Objekt identisch zurückgibt:

```
public Point id() {return this;}

Point q = new Circle(8,9,6);
q.Print();               // Ausgabe: 8 9 6
Circle cid = c.id();    // Compilerfehler, da c.id() vom Typ Point ist
```

Der Versuch, in der Klasee `Circle` die Methode `id()` umzudefinieren in `public Circle id() {return this;}` scheitert ebenfalls an einem Compilerfehler, da Regel 1 verletzt wird (gleiche Argumenttypen, aber unterschiedlicher Ergebnistyp). Vielleicht braucht man ja kein id() ...

Ähnliches Beispiel: Eine Methode `copy` soll ein Objekt kopieren, damit zwei unabhängige Objekte gleichen Typs entstehen:

```
class Point
{
  ...
  public Point Copy() {return new Point(x,y);}
}

Point q1 = new Circle(8,9,6);
q1.Print();                      // Ausgabe: 8 9 6
Point q2 = q.Copy();
q2.Print();                      // Ausgabe: 8 9
```

Für Kreise ist diese Methode aber ungeeignet, und die Redefinition der Methode in der Form `public Circle Copy()` ist verboten (vgl. Regel 1). Ein Ausweg wären verschiedene Methodennamen `public Point PointCopy()` und `public Circle CircleCopy()`.

3.2.4 Klassenmethoden

Bisher haben wir Methoden stets als Objektmethoden verwendet, d.h. sie werden an ein Objekt geschickt: `object.method()`. Eine Klassenmethode dagegen wird an eine Klasse geschickt: Point.method(). Da diese Methoden nicht mehr an ein instanziiertes Objekt geschickt werdne, haben diese natürlich keinen Zugriff auf Instanzvariablen. Beispiel:

```
class Point
{
  ...
  public static Point Copy(Poinz p)
  { return new Point(p.X(), p.Y()); }
}

class Circle extends Point
{
  ...
  public static Circle Copy(Circle c)
  { return new Circle(c.X(), c.Y(), c.R()); }   // kein Compilerfehler
}

Circle c = new Circle(8,9,6);
c.Print();                       // Ausgabe: 8 9 6
Point cc = Circle.copy(c);
cc.Print();                      // Ausgabe: 8 9 6
Point q = c;
q.Print();                       // Ausgabe: 8 9 6
```

```
Point q1 = Point.copy(q);
q1.Print();                     // Ausgabe: 8 9
Point q2 = Circle.copy(q);
q2.Print();                     // Ausgabe: 8 9 !
```

Erklärung: Es gibt zwei verschiedene Methoden `copy(Point)` und `copy(Circle)`. Deshalb entscheidet der Compiler statisch, welche der beiden verwendet wird. In der Klasse `Circle` sind durch Vererbung beide Methoden enthalten. Da `q` vom Typ `Point` ist, wird bei `q2` copy(Point) gewählt.

3.2.5 Binäroperationen: Vergleich

`q1 == q2` ergibt nur `true`, wenn `q1` und `q2` auf dasselbe Objekt zeigen:

```
Point q1 = new Point(1,1);
Point q2 = new Point(1,1);
q1 == q2;  // false
```

1. Ansatz: In der Klasse Point folgende Methode definieren:

   ```
   public boolean eq(Point p)
   { return (x==p.X()) && (y==p.Y()); }
   ```

   In der Klasse Circle kann man analog eine Methode `eq(Circle)` definieren:

   ```
   public boolean eq(Circle c)
   { return super.eq(c) && (r==p.R()); }

   Circle c1 = new Circle(0,0,1);
   Circle c2 = new Circle(0,0,2);
   Point p0 = new Point(0,0);
   Point p1 = c1;
   Point p2 = c2;
   ```

   Der paarweise Vergleich der Variablen `c1, c2, p0, p1, p2` ergibt folgende Ergebnisse:

	eq(p0)	eq(p1)	eq(p2)	eq(c1)	eq(c2)
p0.	true	true	true	true	true
p1.	true	true	true	true	true
p2.	true	true	true	true	true
c1.	true	true	true	true	false
c2.	true	true	true	false	true

   In den drei linken Spalten gibt es `eq(Point)` und `eq(Circle)`, der Compiler wählt statisch `eq(Point)`, da das Argument immer vom Typ `Point` ist. `eq(Circle)` wird nur bei den vier direkten Vergleichen von `c1` und `c2` verwendet (unten rechts).

3.2.6 Binäroperatoren

Das Ziel von eq ist, Punkte und Kreise zu vergleichen. Trickreiches Programmieren:
Damit die dynamische Auswahl möglich ist, muß `eq` in `Point` und in `Circle` denselben Argumenttyp haben (Point). Damit das Argument ebenfalls dynamisch auswählen kann, schickt man ihm eine Hilfsbotschaft `eqh`.

- In Point:

  ```
  public boolean eq (Point q)
    {
     return q.eqh (this,false,0);
    }
  ```

  Dabei bedeutet `this` 'das bin ich', `false` 'ich bin kein Kreis' und 0 'irgendetwas'.

- In Circle:

  ```
  public boolean eq (Point q)
    {
     return q.eqh (this,true,r);
    }
  ```

  Dabei bedeutet `true` 'ich bin ein Kreis', r=Radius des Kreises.

- In Point:

  ```
  boolean eqh (Point p, boolean pCircle, float pRad)
    {
     return !pCircle && p.X()==x && p.Y()==y;
    }
  ```

  Dabei bedeutet `!pCircle`, daß es ein Punkt ist, also gleich p, wenn es kein Kreis ist.

- In Circle:

  ```
  boolean eqh (Point q, boolean pCircle, float pRad)
    {
     return pCircle && p.X()==x && p.Y()==y && pRad==r;
    }
  ```

  Dabei bedeutet `pCircle`, daß es ein Kreis ist, also nur gleich p, wenn das auch ein Kreis ist. `p.Rad()` wäre (statisch) nicht zulässig.

- Beispiel:

```
Point p0=new Point (0,0);
Circle c1=new Circle (0,0,1);
Point p1=c1;
Circle c2=new Circle (0,0,2);
Point p2=c2;
```

Objekt	eq(p0)	eq(p1)	p2	c1	c2
p0.	+	-	-	-	-
p1.	-	+	-	+	-
p2.	-	-	+	-	+
c1.	-	+	-	+	-
c2.	-	-	+	-	+

Jetzt werden unabhängig vom deklarierten Typ die Objekte verglichen.

Es ist einfacher mit zusätzlichen Sprachkonstruktoren:

- In Point:

```
public boolean eq (Point q)
  {
   return !(q instanceof Circle) && x==q.X() && y==q.Y();
  }
```

- In Circle:

```
public boolean eq (Point q)
  {
   return (q instanceof Circle) && x==q.X() && y==q.Y()
   && r==((Circle)q).R();
  }
```

- Erklärung:
  Durch `(Circle)q` wird `q` der Typ `Circle` zugesprochen und damit ist der Methodenaufruf `R()` zulässig.
  !Achtung! Gefährliche Konstruktion.
  Ausdruck `a instanceof A` testet zur Laufzeit, ob `a` ein Objekt der Klasse `A` ist.

3.2.7 Schnittstellen in Java

In C++ ist das Konstrukt `class A` {..} die Schnittstelle von A, die Implementierung erfolgt ausserhalb.
In Java ist `class A` {..} die Implemtierung und die Schnittstelle ist etwas eigenes.

Beispiel

```
interface STACK {boolean empty(); char pop(); void push(char)};
class Stack implements STACK {..empty..};
```

Implementierung und Schnittstelle haben verschiedene Namen (wie in SML). Es kann also mehrere verschiedene Namen geben.

Beispiel

```
class Stack implement STACK {...;Stack copy();..}
interface STACK {.... STACK copy()..}
```

Dieses Beispiel ergibt einen Compilerfehler `(stack#STACK)`. Man muß beide Male `Stack` oder `STACK` benutzen.

Beispiel

Point und Circle haben beide Methode `void print()`. Auch ganz andere Klassen, z.B. `Name` können diese Methode `void print()` haben.

```
interface Printable {void print()}
class Point implements Printable {..}
class Name implements Printable{..}
```

Interfacenamen kann man als Typ benutzen:

```
Printable p=new Circle(1,2,3);
p.print() ist statisch zul\"assig.
Printable pa[]=new Printable[3];
Printable[] pa=new Printable[3];
pa[0]=new Point(0,1);
pa[1]=new Name(..);
pa[2]=new Circle(..);
(eleganter: Printable pa[]={newPoint(0,0), new Name(..),
 new Circle(..)\})
void printall (Printable pf[])
  {
   for (int i=0; i<pf.lenght; i++)
     {
      pf[i].print();
     }
   }
```

Durch `printall(pa)` erhält man Punkt=(0,0), Name=(..),Kreis=(..).
In Java hat jede Klasse höchstens eine Oberklasse (wenn man keine angibt, dann `Object`.

Aber eine Klasse kann mehrere Schnittstellen 'implementieren'.
Beispiel: `class Point implements POINT, Printable, Moveable..`
Vererbung gibt es auch bei Schnittstellen:
Beispiel: `interface I extends I1,..In{..}`
`I` ist Vereinigung von `I1,..In` und {..}. Warum geht das bei Interfaces (Schnittstellen) und nicht bei Klassen? Wenn man auf zwei verschiedene Methoden `void f()` zugreift, hat man bei Schnittstellen keine Probleme, da die Rümpfe nicht mit angegeben werden und somit die Methoden gleich sind. Bei Klassen hat man zwei verschiedene Rümpfe und man weiß nicht welcher gemeint ist.
(In C++ sind trotzdem mehrere Oberklassen möglich, siehe später.)

3.2.8 Abstrakte Klassen/Methoden

Beispiel:

Point/Circle erweitert.

```
    Point
    x,y,move
       |
    Figure
     /    \
Circle  Square
area(r) area(a)
```

`Figure` ist ein geometrisches Gebilde in der Ebene mit ausgezeichnetem Punkt. a ist die Seitenlänge des Quadrates, r der Radius des Kreises. Wenn man so wie jetzt Figure anlegen würde, könnten die Komponenten Kreise oder Quadrate sein, aber man könnte `area` nicht auf die Komponenten anwenden (statisch unzulässig).

Abhilfe:

Man gibt auch `Figure` eine Methode `area`. Aber was soll die berechnen?
Hier einfach nichts.

```
abstract class Figure extends Point
  {
   public abstract float area();
  }
```

Anmerkung: `Figure` könnte zusätzlich auch konkrete Methoden beinhalten.
Es gibt zwar `Figure`-Variablen, aber keine `Figure`-Objekte, `new Figure()` ist statisch unzulässig.
Eine abstrakte Klasse macht nur Sinn, wenn sie konkrete Unterklassen hat.
Gegenteil: `final class` {..} darf keine Unterklassen haben. Finale Methoden dürfen nicht in Unterklassen umdefiniert werden. Finale Variablen erlauben keine Zuweisung (außer in der Deklaration):
`final int n=100` ist erlaubt, da es eine Konstante ist. `n=200` ist verboten.

3.2.9 Sichtbarkeit in Java

Java-Programme bestehen aus `packages`, die jeweils mehrere Klassen und Schnittstellen beinhalten.
Für Variablen und Methoden gibt es 4 Stufen der Sichtbarkeit:

- `public`:
  Überall sichtbar.
- `default`:
  (d.h. es steht nichts da). Sichtbar in dem Paket, in dem die Klasse liegt.
- `protected`:
  Nur in der Klasse selbst und in ihren Unterklassen sichtbar.
- `private`:
  Nur in der Klasse selbst sichtbar. !Achtung! Das bezieht sich auf die Klasse, nicht auf einzelne Objekte.
  Z.B.:

  ```
  class A {private int geheim;
    .. void spion (A a) {geheim=a.geheim};
    .. }
  A a1=new A(..); A a2=new A(..);
  a1.spion(a2);
  ```

  Dies ist alles erlaubt.

Klassen selbst könne 2 Sichtbarkeitsstufen haben:

- `default`:
  Nur im eigenen package sichtbar.
- `public`:
  Überall sichtbar.

Einfaches Javaprogramm:
Datei X.java besitzt genau eine Methode `public class X..`

```
class A ..
class B ..
  .
  .
public class X {...
  .
  .
  public static void main (String[] argv)
    {
     Hauptprogramm
    }
}
```

Folgendes gibt es sonst noch in Java:

- Ausnahmen (`exceptions`);
- Threads ('paralle Prozesse'), diese gibt es nicht in C++.
- Es gibt keine generischen Klassen, z.B.`Stack <Element>`, jedoch gibt es diese in C++.

3.3 Andere 00-Sprachen

3.3.1 C++

Hier gibt es keine Schnittstellen, dafür aber Mehrfachvererbung, Freunde und generische Klasen.

Sichtbarkeit

- `public`:
  Überall sichtbar für Methoden und Variablen.
- `protected`:
  In der Klasse selbst, all ihren Unterklassen und für Freunde sichtbar.
- `private`:
  Entspricht `default` und ist in der Klasse selbst und für Freunde sichtbar. (Freunde genießen mehr Vertrauen als Erben).

Im Gegenteil zu Java treten diese Stufen auch in der Vererbung auf:

- `class B:public A` {`..`alle von A geerbten Methoden haben dieselbe Sichtbarkeit wie in A..}
- `class B protected A` {`..`alle von A geerbten Methoden (können nur `public` oder `protected` gewesen sein) sind jetzt `protected`, d.h. sie werden weitervererbt, können aber von den Kunden von B nicht benutzt werden..}
- `class B:private A` {`..`B erbt zwar von A, aber diese Methoden sind weder weitervererbbar noch nach aussen sichtbar, es sei denn B enthält:
  `public:..A::m() bzw protected:..A::m()..`}

Dabei bedeutet ':' dasgleiche wie `extends` in Java.

Freunde

Im Rumpf der Klasse `A` kann eine Funktion `f(..)` oder `B::m()` oder eine Klasse `C` (d.h. alle ihre Methoden) zu `friend` deklariert werden, somit sehen `f|m|C` alle Variablen von `A`. Freundschaft ist aber nicht erblich!

Mehrfachvererbung

- Problem: `class C:public` B_1`, public` B_2 {`..`}
  In B_1 und B_2 kommt `void m()` mit verschiedener Implementierung vor. Man muß immer mit angeben welche Methode gemeint ist:
  `C.`B_1`::m() bzw C.`B_2`::m()`.
  Wenn `C.m` als `C::m()` {B_1`::m();`} neu imlpementiert wird, dann ist auch `C.m()` möglich, diese ruft dann B_1`::m` auf.

- Anderes Problem:
  A mit Variable x

$$\begin{tikzcd}[column sep=small] & A & \\ B_1 \arrow[ur] & & B_2 \arrow[ul] \\ & C \arrow[ul] \arrow[ur] & \end{tikzcd}$$

  Haben C-Objekte 2 Variablen B_1::x und B_2::x, oder nur eine? Standartmäßig haben sie 2 Variablen, aber es gibt Möglichkeiten dieses zu verhindern.

- Noch ein Problem: Konstruktor C ruft implizit die Methode der Oberklasse auf. In welcher Reihenfolge? A nur einmal? Zweimal?

Generische Klassen:

- Ziel z.B. Stack mit beliebigem, aber in jedem Keller einheitlichem Typ.

```
template <class E> class Stack{..E pop()..void push E()..}
```

- Implementierung: Zur Erinnerung: char Charstack::pop(){..}.
- Hier: template <class E> E Stack <E>::pop(){..}.
- Benutzung: Stack <char> cs; Stack <Point> ps;
- Probleme: Generische Klassen können keine Unterklassen/Oberklassen sein, sondern nur die konkreten Klassen wie Stack <char>.

Allgemeines:

In C++ kann man Operatoren wie +,==,+=,= als Methoden in Klassen definieren. L=R steht für L.operator=R.operator. Dieses ist in Java nicht möglich.

3.3.2 Eiffel:

Eiffel ist direkt als 00-Sprache entworfen worden. Sie besitzt statische Typen, Objektvariablen $\simeq$ Zeiger (wie in Java).

- Deklaration:
  - p:Point (Mit NIL initialisiert)
  - p.Create(0,1) (Create ist Konstruktor (in jeder Klasse); Initialisierung mit Objekt.)
  - class Circle inherit Point redefine print (Umdefinition muß angekündigt werden. Durch inherit gibt man die Oberklasse an.)
    * rename Create as CreatePoint end (Dabei ist Create von Point geerbt und CreatePoint ist der neue Name)
    * feature{NONE} r:REAL; feature(ANY) R:REAL is do Result::r end (Dabei bedeutet NONE, daß dieses für niemanden sichtbar ist, also private ist.)
    * print is do ..end;

    * `Create (_x,_y,_r:REAL) is do CreatePoint(_x,_y);r::_r end;`
      (Dieses entspricht `super(_x,_y)` in Java. In Eiffel gibt es aber kein `super`, deshalb ist ein `rename` nötig.

- Mehrfachvererbung:
  `class C inherit` B_1 `rename..redefine..end`
  `inherit` B_2 `rename..redefine..end`
  Mit `rename` werden Konflikte vermieden.
  A mit Variable x
  $\nearrow \nwarrow$
  B_1 B_2
  $\nwarrow \nearrow$
  C
  C-Objekte haben standartgemäß nur ein x, es sei denn, x wurde unterwegs umbenannt.

- Es gibt abstrakte Klassen(`deferred`) und Methoden.

- Es gibt generische Klassen:
  `class Stack [E] feature ..` S_1`:Stack[REAL];` s_1`:Stack[REAL];`

- Es gibt explizite(`p:Point`) und implizite(`like..`) Typen:
  Z.B.:
  - `eq(x:like Current)`, d.h. das Argument `x` hat denselben Typ wie das Objekt, das die Methode ausführt.
  - `copy():like Current`

Smalltalk

- Dies ist eine eigenständige Entwicklung, die im Graphiksystem eingebettet ist. Vieles ist menüorientiert und hat daher keinen richtigen Syntax. Es gibt kein statisches Typsystem, sondern nur dynamischye Typen, (d.h. Klassen) für Objekte. Deshalb entfällt der Unterschied zwischen Variablentyp und Objekttyp. Es gibt keine statische Methoden- bzw Variablenauswahl und auch keine generischen Klassen (dies wäre auch völlig unnötig).

- Sichtbarkeit:
  Variablen sind immer `private`, Methoden immer `public`.

- Vererbung:
  Es gibt nur einfache Vererbung.

Java	Smalltalk
`o.m()`	`o m`
`o.m(x)`	`o m:x`
`o.m(x,y)`	`o` m_1`:x` m_2`:y` → zweigeteilter Methodenname
[1.+(2)]	1+2

  Alle arithmetischen Ausdrücke sind linksgeklammert, d.h. 1+2*3=(1+2)*3.

- Beispiel:
  Klasse `Node`, Oberklasse `Object`, Variablen `info,link`.
  `content` {↑`info` } `fill:x`{`info`←`x`}
  `next`{↑`link`} `linkTO:x`{`link`←`x`} Dabei sind `content` und `fill` Methodennamen, ↑ entspricht dem `return` und ← ist die Zuweisung.

- Konstruktion: `Node new`
  Dabei bedeutet `Node`, daß die Klasse als Objekt betrachtet wird und `new`, daß es eine Methode dieses Objektes ist.

- Definition eines eigenen Konstruktors:
  `(Node) new:x{↑self new fill:x}`
  Dabei bedeutet `self` der eigene Knoten und `new` ist ein vordefinierter Konstruktor für Knoten mit undefiniertem Inhalt.

- Klasse `Stack`, Variablen `top`:
  ```
  empty {↑top==nil}
  push:x{top←(Node new:x) linkTO:top}
  pop{self empty ifTrue:[↑nil]
  ifFalse:[|cont|cont←topcontent.top←top.next. ↑cont]}
  ```
  Dabei ist `|cont|` die Deklaration einer lokalen Variablen, `'.'` bedeutet `';'` z.B. in Modula und `[..]` ist Objekt der Klasse `Block`, die Ausführung des Inhalts erfolgt noch nicht und versteht Methode `value`(, d.h. führe dich aus). Die Bedingung `self empty` (bzw. `top==nil`) ergibt Objekt der Klasse `True` oder `False`. Die Klasse `True` hat die Methode:
  `ifTrue:thenBlock ifFalse:elseBlock{↑thenBlock value}`
  Die Klasse `False` hat eine analoge Methode.

Kapitel 4

Funktionale Sprachen

4.1 Einführung

4.1.1 Was ist ein funktionales Programm?

Ein funktionales Programm ist eine Folge von Typ- und Funktionsdefinitionen (allgemeiner: Wertdefinitionen). Die definierten Funktionen können auf Werte angewendet werden. Im Idealfall sind funktionale Programme seiteneffektfrei, d.h. es gibt keine Wertzuweisungen. Eine Funktion, die mehrmals auf dasselbe Argument angewandt wird, liefert immer das selbe Ergebnis.

Kennzeichen funktionaler Sprachen:

- Funktionen sind "Bürger 1. Klasse "wie alle anderen Werte. Funktionen können als Argument übergeben werden, in Paketstrukturen eingehängt werden und als Ergebnis geliefert werden.
- Funktionsdefinitionen über Fälle mit Mustern (patterns) (später)
- flexibles Typsystem (später)
- automatische Speicherverwaltung: Programmierer brauchen sich weder um das Anlegen, noch um das Freigeben von Speicher zu kümmern

Weitere Kennzeichen mancher funktionaler Sprachen:

- Verzögerte Auswertung (lazy evaluation, call by need): Zwischenergebnisse werden nur soweit ausgewertet, wie es zur Berechnung des Endergebnisses nötig ist.
- Zwischenergebnisse können (theoretisch) unendlich groß sein, wenn für das Endergebnis nur ein endlicher Teil gebraucht wird (z.B. Liste aller natürlichen Zahlen $\rightarrow$ Liste aller Primzahlen $\rightarrow$ die ersten 100 Primzahlen) Gegenteil: strikte Auswertung, d.h. alle Zwischenergebnisse werden vollständig ausgewertet
- keine Wertzuweisungen/Seiteneffekte (manche funktionale Sprachen haben doch welche, aber es wird abgeraten, sie zu benutzen)

Zusammengefaßt: Funktionale Sprachen sind abstrakter und "höher"als die imperativen Programmiersprachen. Sie erlauben kürzere, einfachere, korrektere Programme als in imperativen Sprachen, zumindest für geeignete Probleme.

Beispiele funktionaler Sprachen:

- ML (d.h. meta-language) bzw. SML (standard ML): hat strikte Auswertung und Wertzuweisungen
- CAML (categorical abstract ML): ähnlich SML, aber andere Syntax
- Hope: verzögerte Auswertung, seiteneffektfrei, kleiner Sprachumfang
- Haskell: verzögert, seiteneffektfrei, großer Sprachumfang
- Gofer: verkleinerte Version von Haskell

4.1.2 Einführendes Beispiel: Zusammenhängen von Listen

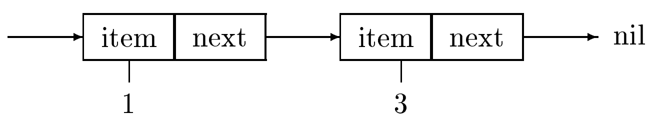
Beispiel: [1,2,3] + [4,5] $\rightarrow$ [1,2,3,4,5]

Lösung in Modula-2 (funktionaler Stil)

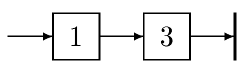
Zuerst benötigen wir einen Datentyp für Listen:

```
TYPE list = POINTER TO node;
     node = RECORD first : INTEGER;
                   rest  : list
            END;
```

Die Liste [1,3] kann man graphisch golgendermaßen darstellen:

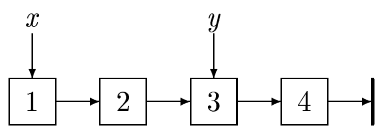


Oder kurz:



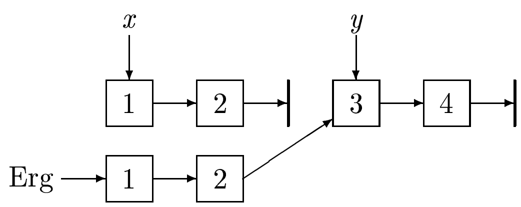
Bei dem Versuch, das zu implementieren, stellt sich heraus, daß das Problem nicht genau spezifiziert ist. Es existieren drei Möglichkeiten:

1. Ergebnis:



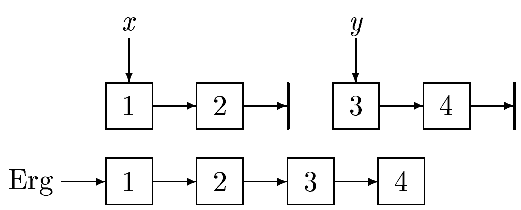
Verändert den Wert des 1. Arguments durch einen Seiteneffekt.

2. Ergebnis:



Seiteneffektfrei, aber es besteht die Möglichkeit, durch Manipulation des Ergebnisses, y nachträglich zu ändern.

3. Ergebnis



Jetzt sind x und y auch sicher vor Änderung durch Manipulation des Ergebnisses.

In funktionalen Sprachen wird Möglichkeit 1 nicht verwendet, weil sie Seiteneffekte benutzt. Möglichkeit 2 und 3 sind ohne Seiteneffekte nicht zu unterscheiden. Normalerweise wird Möglichkeit 2 verwirklicht (effizienter als 3).

Hilfsfunktion cons hängt Zahl vorne an Liste an:

```
cons(1, [2,3]) = [1,2,3]

PROCEDURE cons( n : INTEGER; x : list) : list;
VAR y : list;
BEGIN
  (* keine destruktiven Wertzuweisungen, sondern Initialisierungen *)
  new(y); y.first := n; y.rest := x;
  RETURN y
END cons;
```

```
PROCEDURE append( x, y : list) : list;
BEGIN
  IF (x = NIL) THEN RETURN y
    ELSE RETURN cons( x.first, append( x.rest, y ))
  END;
END append;
```

Beispiel:

```
  append([1,2],[3])
= cons(1, append([2],[3]))
= cons(1, cons(2, append( NIL, [3]))
= cons(1, cons(2, [3]))
= cons(1, [2, 3])
= [1, 2, 3]
```

Lösung in SML und Gofer

Eigentlich sind sowohl Listen, als auch append in Gofer vordefiniert, aber wir tun so, als wüßten wir das nicht.

Definition des Listentyps:

Gofer: `data List | Cons (Int, List)`

SML: `datatype List = Nil | Cons of Int* List`

Man hat nun definiert: Einen Typ `List` und die Konstruktoren, also eine Konstante `nil: List` und eine Funktion `cons: int* List` → `List`. Im Gegensatz zu Modula muß Cons nicht eigens definiert werden.

Die Darstellung einer Liste sieht weifolgt aus:
[1,2] als `Cons(1, Cons(2, Nil )`
[ ] als `Nil`

Gofer:	`append(nil, y) = y`
	`append(Cons( n, x), y ) = Cons(n, append(x, y))`
SML:	`fun append(nil, y) = y | append(cons(n, x), y) = Cons(n, append(x, y))`

Typangaben sind überflüssig, die Typen leitet der Compiler selbst her.

Bei Gofer ist eine Typangabe möglich: `append :: ( List, List ) -> List`

Bei SML sähe das so aus: `append : List * List -> List`

Das kann aber nicht direkt hingeschrieben werden, sondern nur in eine "signature", wenn der Funktionscode in dazugehöriger "structure" steht. Der Typ wird in dieser Form vom Compiler ausgegeben.

Eine andere Möglichkeit in Modula-artiger Syntax:

```
fun append(Nil, y : List) : List = y
|   append(cons(n : Int, x : List), y : List ) : List = ...
```

(Alle diese Angaben können unabhängig voneinander gemacht oder weggelassen werden)

Auch arithmetische Funktionen können über Fälle definiert werden:

i) `fac n = if == 0 then 1 else n * fac( n - 1 )`

ii)
```
fac 0 = 1
fac n = n * fac(n - 1)
```

Die Fälle werden von oben nach unten durchprobiert.

4.1.3 Polymorphie

In Modula-2 ist immer ein fester Elementtyp für Listen erforderlich (im vorangegangenen Beispiel `INTEGER`), in SML/Gofer kann er dagegen variabel sein.

Gofer: `data List elem = Nil | Cons( elem, List elem)`

SML: `datatype 'elem List = Nil | Cons of  'elem * 'elem List`

Dabei ist `List` der Typkonstruktor und `elem` die Typvariable. In SML steht der Typkonstruktor hinter seinem Argument, in Gofer davor. In SML haben Typvariablen eine besondere Syntax: `'name`

Es ist Sitte, die Typvariable `a, b, c,...` zu nennen (bzw. `'a, 'b, 'c, ...`).
`[] : 'a List` Compiler benutzt auch diese Namen.

4.1.4 Weitere Eigenheiten funktionaler Sprachen

Currying bzw. iterierte Funktionsanwendung

`avg(x, y) = (x + y) / 2.0` berechnet Mittelwert von 2 Zahlen. Dabei werden folgende Datentypen benutzt:

Gofer: `(Float, Float) -> Float`

SML: `real * real -> real`

Jede Funktion hat genau ein Argument (in diesem Falle ein Paar). Beispiele:

`p = (2.0, 4.0) ; avg p -> 3.0`

Andere Funktion: `avg' x y = (x + y) / 2.0`

Anwendung: `avg' 2.0 4.0 -> 3.0`

Funktionsanwendung klammert links: `( avg' 2.0 ) 4.0`

Also ist `avg' :: Float -> ...`

Ergebnis: `avg' 2.0` nimmt Argument 4.0 und liefert 3.0

Also: `avg' 2.0 :: Float -> Float`
`avg' :: Float -> ( Float -> Float )`
`Float -> Float -> Float`

Übergang von `avg` zu `avg'` heißt "*Currying*" (nach Haskell Curry). In Gofer, Miranda, Haskell ist es Sitte, Currying zu benutzen.

`f x y = ...` statt `f(x, y) = ...` bzw. `f x y z = ...` statt `f(x, y, z) = ...`

Auch vordefinierte Funktionen sind "curried". In SML ist Currying nicht ganz so verbreitet. In Gofer ist Currying auch bei Konstruktoren möglich, z.B.:
`data List a = Nil | Cons a ( List a )` (statt: `Cons( a, List a )`)
In SML ist Currying bei Konstruktoren unmöglich.

Infixoperatoren

In imperativen Programmiersprachen gibt es i.a. eine fest vorgegebene Zahl von Infixoperatoren

SML/Gofer: Neue Infixoperatoren definierbar
Gofer: `infixr 5 :+, +++`
SML: `infixr 5 :+ +++`
Rechtsass. Priorität (Cons) (append)
Gofer: `data List a = Nil | a :+ List a`
SML: `datatype 'a List = Nil | :+ of 'a * 'a List`

```
→ Nil +++ y = y
  (n :+ xr ) +++ y = n :+ (xr +++ y)
```

Aufhebung des Infixstatus:

Gofer:	`(+++)`	, also `x +++ y` entspricht `(+++) x y`
SML:	`op +++`	, also `x +++ y` entspricht `(op +++) (x, y)`
Gofer:	`(+++) ::`	`List a -> List a -> List a`
SML:	`+++ :`	`'a List * 'a List -> 'a List`

Das gilt für alle Infixe!

Überladung

Einige Funktionsnamen, z.B. + und *, stehen für mehrere verschiedene Funktionen unterschiedlichen Typs. Von denen wird die richtige Fubktion anhand der Argumenttypen ausgewählt (wenn möglich).

SML:
```
*: int  *  int -> int
*: real * real -> real
```

Jedes * in einem Ausdruck muß entscheidbar sein:

```
fun dup x = 2 * x
fun sqr x = x * x (geht nicht!!)
fun isqr (x: int) = x * x (ist richtig)
```

Überladung pflanzt sich nicht fort (auf sqr) und ist nicht erweiterbar (z.B. * auf Matrix).

Gofer:
```
* ::   Int -> Int   -> Int
* :: Float -> Float -> Float
```
sind Instanzen von
```
* :: Num a => a -> a -> a
```

(Zu lesen: Typ `a -> a -> a` für alle Typen `a` mit Eigenschaft `Num a` (`a` ist numerisch), "`Num`" ist eine Typklasse)

`dup x = 2 * x`	(ok)	`dup :: Int -> Int`
`sqr x = x * x`	(ok)	`sqr :: Num a => a -> a`

Typklassen sind erweiterbar, z.B. `Num` um Matrix complex. Für diese selbstdefinierten Typen ist `*` auch definierbar. Neue Typklassen können definiert werden.

4.1.5 Programmstruktur

Gofer

Es existiert ein interaktiver Interpreter, der eingegebene Ausdrücke auswertet. Definitionen können nicht direkt eingegeben werden, sondern müssen aus Dateien geladen werden.

Beispiel: Datei list.gs

```
(data List = ...
infixr : ...
Def. +++
```

Eingabe	Ausgabe
`1 + 2`	`3 ( 2 red., 7 cells )`
`Nil`	`Error : undefined Nil`
`: l list.gs`	....
`Nil`	`Nil ( .... )`
`( 1 :+ 2 :+ Nil ) +++ ( 3 :+ Nil )`	1 :+ 2 :+ 3 :+ Nil (...)
`:  t Nil`	`Nil :: List a`
`: t ( +++ )`	`( +++ ) :: List a -> List a -> List a`
`:e`	`[Editor auf zuletzt geladenem File]`
[Editor verlassen]	[zurück in Gofer, File wird wieder gelesen]
`: q`	[Ende]

Miranda

Hier funktioniert es so ähnlich.

SML

Interpreter erlaubt Ausdrücke und Definitionen (daher haben Definitionen Schlüsselworte wie `fun`). Alle Eingaben mit ";" abschließen. `use "Name";` liest Datei "Name" (mit Ausdrücken und Definitionen) ein. `"use"` ist kein spezielles Kommando, sondern eine Funktion, die als Seiteneffekt das File einliest und compiliert, bzw. auswertet.

Nachteil: Seiteneffekt
Vorteil: Argument kann beliebiger string-wertiger Ausdruck sein. Editor nicht dirket angeschlossen (aber Anschluß programmierbar).

4.2 Namen

In Gofer gibt es vier Arten von Namen:

	Variablen	Konstruktoren
Bezeichner	z.B. `x, y, append, a, elem,` ...	`Nil, Cons, List, Int,` ...
Operatoren	`+, *, +++,` ...	`:+`

4.2.1 Bezeichner ↔ Operatoren

a) Bezeichner: Buchstabe (Buchstabe | Ziffer | ' | _)*
Einige Bezeichner sind reserviert, z.B. `data, infix,` ...

b) Operatoren: Folgen aus `! $ : # % & * + . / < = > ? @ \ ^ | $ -`
Einige Operatoren sind reserviert, z.B. `:: = -> =>`

c) Bezeichner `->` Operator: ‘...‘
Beispiel: `f x y = x * x + y * y`
`2 ‘f‘ 3 = f 2 3 -> 13`
(Operatoren sind immer Infix, mit einer festen Priorität)
Operator `->` Bezeichner: (...)

d) Operatorschnitte (operator sections)
Beispiel: `( / 4 ) :: Num a => a -> a`
`( / 4 ) n = ( n / 4 )`
`( 4 / ) :: Num a => a -> a`
`( 4 / ) n = 4 / n` (Probleme mit “-”)

SML: kein Wesensunterschied zwischen Operatoren und Bezeichnern.
Ein Name ist Infix ⇔ er ist als Infix deklariert
Normal → Infix: durch Deklaration
Infix → Normal: durch `op`
Schnitte gibt es nicht.

4.2.2 Variable ↔ Konstruktor

Variable: Bezeichner mit Kleinbuchstaben am Anfang, Operator, der nicht mit “:” anfängt.

Konstruktor: Bezeichner mit Großbuchstaben am Anfang, Operator mit “:” am Anfang

a) **In Werte-Welt:** Variablen werden durch Definitionen an Werte gebunden:

```
x = 7    sqr n = n * n     first (cons n x) = n
```

Oder sie dienen als Namen für Parameter oder Teile eines Parameters, oder kommen in Ausdrücken vor.

Konstruktoren werden durch Typdefinitionen eingeführt:

```
data List a = Nil | Cons a ( List a )
```

Sie können in Mustern und in Ausdrücken vorkommen:

```
append ( Cons n x ) y = Cons n ( append x y )
```

Variablen dürfen in Mustern nur als Blätter vorkommen:

```
f ( x + y ) = ...    | g ( append x y ) = ...
```
(geht beides nicht)

b) **In Typ-Welt:** Konstruktoren sind feste (definierte) Typen (z.B. Int) und Typ-Funktionen (z.B. List) bzw. Typklassen (z.B. Num). Variablen stehen für unbestimmte Typen:

```
Nil :: List a
```

d) **SML:**
Typwelt: Variablen haben eigene Syntax: `'a`
feste Typen können beliebig geschrieben werden.

Wertewelt: Es gibt keinen äußerlich erkennbaren Unterschied zwischen Konstruktoren und Variablen. Ein Name ist ein Konstruktor gdw. er durch eine Typdefinition dazu gemacht wurde. Konsequenz:

```
Nill +++ y = y
```
(Schreibfehler)
```
(n :+ x) +++ y = n :+ (x +++ y)
```

Gofer: Fehlermeldung, Konstruktor `Nill` nicht deklariert.

SML: `Nill` ist Variable. Daher trifft der 1. Fall auf alle Argumente ⇒ Funktion ist ausführbar, tut aber etwas ganz anderes, als erwartet. Fehlerquelle!!

4.3 Typen

4.3.1 Vordefinierte einfache Typen

Ganze Zahlen

	Gofer	SML
Typ	`Int`	`int`
Übliche Operationen Vergleiche: gleich ungleich Verwendung	 `==` `/=` alle Typen der Klasse Eq	 `=` `<>` alle Typen außer Funktionen
unäres Minus	`-`	~
Beispielwerte	`1, -1`	`1, ~1`

Gleitkomma

	Gofer	SML
Typ	`Float`	`real`
Klasse	`Num`	-
Übliche Operationen mit `Int` überladen `1+2.0` verboten		
Konvertierung	`primIntToFloat::` `Int` → `Float`	`real:` `int` → `real`

Zeichen

	Gofer	SML
Typ	`Char`	`char`
Beispielwerte	`'a'` `'\n'`	#a, #"\n" (denn `'a'` ist Typvariable)

Wahrheitswerte

	Gofer	SML
Typ	`Bool`	`bool`
Werte	`True, False`	`true, false`
Operationen	`not`, &&, ‖ `False` && `y = False` `True` && `y = y` 2. Arg. wird nicht ausgewertet,wenn 1. Arg. `False` ist	`not, andalso, orelse` Achtung: `x andalso y` `x orelse y` sind keine Fkt.-Anw., sondern spez. Syntax, wo 2. Arg. nur bei Bedarf ausgewertet wird
`if`-Anweisung	`if` Bed `then` Ausdr_1 `else` Ausdr_2	

Auch in SML wird nur einer der beiden Ausdrücke ausgewertet.
Bsp.: `fac n = if n = 0 then 1 else n * fac(n-1)`
Vorsicht:

```
cond b x y = if b then x else y
fac n = cond (n = 0) 1 (n * fac(n-1))
```

- SML: `cond` wertet alle Argumente aus.
  Dieses `fac` terminiert also nie.
- Gofer: `cond` gibt die Argumente unausgewertet weiter an `if`.
  Daher wird nur bei Bedarf ausgewertet - Definition ist in Ordnung.

Einheitstyp

	Gofer	SML
Typ	`()`	`unit`
Elemente	()	()

In Gofer kaum gebraucht, in SML in Zusammenhang mit Wertzuw. benutzt:

:= ⇒ `'a ref * 'a -> unit`
f: `unit -> int` ⇒ Funktion, die beim i. Aufruf i zurückgibt
(in Gofer nicht definierbar)

Gofer:

() `->` $\tau \cong \tau$

4.3.2 Vordefinierte zusammengesetzte Typen

Tupeltypen

Deklaration: $(\tau_1, ..., \tau_n)$ $\quad \tau_1 ... * \tau_n$ $\quad (n \geq 2)$
Elemente: $(v_1, ..., v_n)$ $\quad (v_1 * ... * v_n)$ $\quad (v_i : \tau_i)$
Es gibt keine 1-Tupel! Also: $(v) \equiv v$.

Records

Gibt es in Gofer nicht. In SML:

- Typ: $\{l_1 : \tau_1, ..., l_n : \tau_n\}$
  Typen: $\tau_1, ..., \tau_n$
  Komponentennamen: $l_1, ..., l_n$
- Werte: $\{l_1 = v_1, ..., l_n = v_n\}$
  Die Komponenten können beliebig permutiert werden.
  #l selektiert Komponente l_i
  Typ $\approx \{..., l : \tau, ...\} \to \tau$
  Solche Typen gibt es aber nicht. #l ist überladen. Meist muß ein konkreter Typ angegeben werden. Es gibt keine elegante Methode, eine Komponente zu ändern und die anderen gleich zu lassen.

Listen

	Gofer	SML
Typ	$[\tau]$	τ `list`
Werte	$[v_1, ..., v_n]$ mit $v_i : \tau$	
leere Liste	[ ]	[ ] oder `nil`
nichtleere Liste vgl. Wert v hat Typ τ	: (Infix) `[1,2] = 1:2:[ ]` $v :: \tau$	:: (Infix) `1::2::nil` $v : \tau$
viele vordef. Funktionen	`null :: [a] -> Bool` `null [ ]    = True` `null(x:xs) = False` oder: (_ : _)	`null : 'a list -> bool` `fun null nil     = true` `|   null (x::xs) = false` oder: (_ :: _)

Sitte: x ist Element von xs.
In Muster ist _ (underscore) statt Name möglich, wenn dieser Name nicht gebraucht wird. Andere Funktion:

```
map :: (a->b) -> [a] -> [b]                (Gofer)
bzw.   ('a -> 'b) -> 'a list -> 'b list    (SML)
map f [ ] = [ ]
map f (x : xs) = f x : map f xs
```

Bewirkt: `map` f $[v_1, ..., v_n] = [f(v_1), ..., f(v_n)]$
z.B. `map(2 *) [1,2,3] -> [2,4,6]` (Gofer)
Operatorschnitt (\ `x -> 2 * x`).
In SML: `fn x => 2 * x)`
Zusammenhängen zweier Listen: `++` in Gofer, @ in SML.

Zeichenketten

Gofer:
Typ `String` ist identisch mit `[Char]`
$\Rightarrow$ alle Listenfunktionen sind auf Strings anwendbar.
Dazu gibt es einige eigene vordefinierte Funktionen + eigene Syntax:
`"abc"= ['a','b','c'] = 'a':'b':'c':[ ]`

SML:

`string` ist eigener Typ $\neq$ `char list`.
Werte z.B. `"abc"`, einige vordefinierte Stringoperationen, z.B.

- `explode: string -> char list`
- `implode: char list -> string`
  Damit sind Listenoperationen indirekt anwendbar.

Felder

Gofer:

relativ kompliziert:
`Array a b` ; `a` Typ der Indizes, `b` Typ der Feldelemente
z.B. `Int, Char, (Int, Int)`
Numerierung beginnt beliebig
Funktionale Arrays: werden kopiert bei Änderung einer Stelle.

SML:

`'a array` ; Indizierung immer `int`, immer von 0 an
Dabei: `a` Typ der Feldelemente
Imperative Arrays: nicht kopiert bei Änderung $\Rightarrow$ Seiteneffekte!
`'a vector` : Read-only-Array mit O(1) Zugriff auf alle Elemente
(vgl.: `'a list` : O(n) für Zugriff auf n-tes Element)

Referenztyp

`'a ref` (Typ der imperativ zuweisbaren Variablen)
Elemente: `ref` ν (Variable mit Inhalt ν)
`ref` ist Konstruktor.

`! : 'a ref -> 'a`	`:= : 'a ref * 'a -> unit`
(Beispiele frueher)	(nur SML)

Funktionstypen

τ_1 `->` τ_2

4.3.3 Benutzerdefinierte Typen

Gofer:

```
data List a = Nil | a :+ List a
data Bool  = True | False
```
(so vordefiniert)

Allgemein:

`data` Typname $\texttt{var}_1$ `...` $\texttt{var}_n$ `=` Fall_1 `|...|` Fall_m ; $n \geq 0$, $m \geq 1$

Fall ist: Konstruktor Typausdr_1 ... Typausdr_l ; $l \geq 0$

oder: Typausdr_1 Infix-Konstr. Typausdr_2

SML:

Bsp.:

```
datatype 'a List = Nil | :+ of 'a * 'a List
```

Allgemein: (für $n \geq 2$ und $m \geq 1$)

- `datatype` Typname = Fall_1 | ... | Fall_m
- `datatype` Typvar_1 Typname = Fall_1 | ... | Fall_m
- `datatype` (Typvar_1,...,Typvar_n) Typname = Fall_1 | ... | Fall_m
  Fall ist: Konstruktor bzw. Konstruktor `of` Typausdruck

Imperative Entsprechungen sind Aufzählungstypen, rekursive Pointer-Typen, variante Records/Unions. Durch die Definition werden der Typname und die Konstruktoren definiert. Jeder Konstruktor darf nur bei einem Typ benutzt werden. Rekursion ist möglich.
Aber: Konstruktorname darf mit Typname übereinstimmen:

```
data Number = Int Int | Float Float
```

Werte z.B. `Int 5, Float 3.7`
Beispiele: Bäume:

- Binärbaum mit Werten an Blättern:
  `data Tree = Leaf a | Tree a:^: Tree a`
  z.B. `Leaf 1:^: (Leaf 2:^: Leaf 3)`
- Beliebige Stelligkeit, alle Knoten mit Inhalt a:
  `data Tree a = Node a[Tree a]`
  z.B. `Node 1[Node 2[ ], Node 5[Node 0[ ]], Node 7[ ]]`

Typabkürzungen

Gofer:
Bsp.: `type` String = `[Char]` (vordef.)
`type` Pair `a = (a,a)`
Allg.: `type` Typname Typvar_1...Typvar_n = Typausdruck ; $n \geq 0$
SML: (für $n \geq 2$)

- `type` Typvar_1 Typname = Typausdruck
- `type` (Typvar_1,...,Typvar_n) Typname = Typausdruck

Unterschied zu `data(type)`: Keine Konstruktoren, keine Fälle, keine Rekursion, keine echt neuen Typen, sondern nur neue Namen für bereits bestehende Typen.

Bsp.: `type Alter = Int` (1)
`data Alter = Alter Int` (2)

Bei (1): `17 :: Alter`
Für `x :: Alter` ist `x+1 :: Alter`

Bei (2): Nicht `17 :: Alter`, sondern `Alter 17 :: Alter`
`x :: Alter` ⇒ `x+1` führt zu Typfehler.
braucht eigene Funktion:

```
incAlter (Alter n) = Alter (n+1)
incAlter :: Alter -> Alter
```

Nachteil: Funktionen wie incAlter müssen extra def. werden.
Vorteil: klare Unterscheidung `Alter <-> Int,...`

Abstrakte Typen

Bsp.: Keller mit Elementeanzahl und Op. `push, pop, top, number, empty`.

```
(1) type Stack a = (Int,[a])
(2) data Stack a = Stack Int [a]
```

Nachteil der beiden: Zahl kann unabh. vom Kellerinhalt verändert werden.

`(5,['a'])` bzw. `Stack 5 ['a']`

Falsche Werte können angegeben werden.
Lösung: Abstrakter Typ (hier Gofer, SML ähnlich)

```
type Stack a = (Int,[a])
    in push, pop, top, number, empty
empty :: Stack a
empty = (0,[ ])
number :: Stack a -> Int
number (n,_) = n
top :: Stack a -> a
top (_,x:_ ) = x
pop :: Stack a -> Stack a
pop(n,_:xs) = (n-1, xs)
push :: a -> Stack a -> Stack a
push x (n, ys) = (n+1,x:ys)
```

Typspezifikation nötig, um zu sagen, ob `pop` Typ `Stack a -> Stack a` oder `Stack a -> (Int,[a])` oder ... hat. Außerhalb der genannten Funktionen sind `Stack a` und `(Int,[a])` verschiedene Typen.

`5,['a'])` hat Typ `(Int,[Char])`, aber nicht `Stack Char`

Bem.:

1) `type T1 = ... in f, g`
   `type T2 = ... in g, h`
   g kann Intern-Darstellung beider Typen sehen.
2) Wenn die Funktionen *nicht* bei mehreren Typen stehen,
   kann man schreiben:
   ```
   type Stack a = (Int,[a]) in push :: a -> Stack a -> Stack a,
                                 pop  :: ...
   ```

Typschemata

Typ:	ohne Variablen
Typschema:	mit Variablen, z.B. `a -> a` oder `Num a => a -> a`
Verboten:	`Num a => a -> a) -> Bool` oder `(a => a -> a) -> Bool`

Typschemata sind kein Teil von Typausdrücken.

`(a -> a) -> Bool` bedeutet: $\forall$ `a : ((a -> a) -> Bool)`
nicht: ($\forall$ `a : a -> a) -> Bool`

4.4 Definitionen, Muster, Ausdrücke

4.4.1 Definitionen

Gofer

Wertdefinition besteht aus eventuell mehreren Regeln.
Eine einfache Regel sieht so aus: (für n $\geq$ 0, Definitionen können fehlen)

Name Muster_1... Muster_n = Ausdruck `where` Definitionen

oder

Muster_1 Infix-Name Muster_2 = ...

Bsp.: x = 1
```
f x y = (x + y) / 2.0
x 'f' y = ...
x^|^a = ...
```
Regeln sind immer rekursiv.
Regel mit Bedingungen: (für n $\geq$ 0, Definitionen können fehlen)

Name Muster_1... Muster_n | Bed_1 = Ausdr_1
| Bed_2 = Ausdr_2
...
| Bed_m = Ausdr_m
`where` Definitionen

Lokale Definitionen hinter `where` beziehen sich auf alle Bed_i und Ausdr_i.
Bsp.: `max x y = if (x <= y) then y else x`
oder:
```
max x y | x <= y = y
        | otherwise = x
```
Bsp.: `filter :: (a -> Bool) -> [a] -> [a]`
z.B.: `filter (>= 5) [1,5,2,6,3]` = `[5,6]`
```
filter p [ ] = [ ]
filter p (x : xs) | p x = x : ys
                  | otherwise = ys
                  where ys = filter p xs
```
Hier spart die lokale Definition etwas Schreibarbeit.
Kann auch Effizienz steigern durch Verhinderung von mehrfacher Auswertung.
SZerlegende"Definition:

Muster = Ausdruck (rekursiv!)

Kann auch Bedingungen und `where` beinhalten.
z.B.:
```
f :: Int -> (Int, Int)
(x, y) = f 5
```
Bindet x an 1. Komponente des Ergebnisses und y an 2. Komponente.
Bsp.: `filter2 :: (a -> Bool) -> [a] -> ([a],[a])`
1. Möglichkeit:
```
filter2 p xs = (filter p xs, filter (not.p) xs)
g.f = \x -> g (f x)
```
Ist ineffizient, denn p wird zweimal auf alle Listenelemente angewandt.
2. Möglichkeit:
```
filter2 p [ ] = ([ ] , [ ])
filter2 p (x : xs) | p x = (x : ys, zs)
                   | otherwise = (ys, x : zs)
                   where (ys, zs) = filter2 p xs
```

SML

Zwei Arten von Definitionen: (für n $\geq$ 1)

- `fun` Name Muster_{11}... Muster_{1n} = Ausdr_1
  | Name Muster_{21}... Muster_{2n} = Ausdr_2
  ...
  Mindestens ein Argument, rekursiv, keine Bed., kein `where`, Infix links möglich.

- `val` Muster = Ausdruck (z.B. `val x = 1`)
  Nicht rekursiv!
  Bsp.:
  ```
  fun filter2 p [ ] = ([ ] , [ ])
    | filter2 p (x :: xs) =
          let val (ys, zs) = filter2 p xs
          in if p x then (x :: ys, zs)
                    else (ys, x :: zs)
          end
  ```
  Dasselbe ginge auch in Gofer, nur ohne `fun, val, end`.

- Neudefinitionen sind in Gofer verboten, gilt auch für vordef. Funktionen.
  In SML erlaubt:
  ```
  val x = 1; fun f y = x + y;
  val x = 3;     f 3 = 4
  ```
  Statische Bindung, keine Wertzuweisung!

4.4.2 Muster

Beispiele:
Konstruktor Muster_1 ... Muster_n
Muster_1 Infix-Konstruktor Muster_2
(Muster_1 ,..., Muster_n)
[Muster_1 ,..., Muster_n]
Konstante Name (wird gebunden)
_ (underscore)

Gofer:
Name @ Muster

SML:
Name `as` Muster

Name wird an ganzen Wert gebunden, der gleichzeitig durch das Muster weiter zerlegt wird.

```
sortPair p @ (x,y) | x <= y = p
                   | otherwise = (y, x)
```

In der Implementierung ist Ergebnis p identisch mit dem Argument i.
Wenn man (x,y) geschrieben hätte, wäre ein neues Paar gebildet worden.

4.4.3 Ausdrücke

Bsp.: Konstante, Name, e_1 e_2 (Funktionsanw.)
(e_1 ,..., e_n) [e_1 ,..., e_n]

```
if e then e1 else e2
let Defs in e end
```

Gofer:

\Muster$_1$... Muster$_n$ `->` Ausdr

```
case l of [ ] -> ...
          x : xs -> ...
```

SML:

```
fn ... => ...
|   ... => ...
```

Bsp.: `\x -> x + 1`

```
case l of [ ] => ...
        | x :: xs => ...
```

4.4.4 Probleme mit Überladung

a) Zur Implementierung

Der Compiler transformiert das Programm, indem er den überladenen **Funktionen** ein zusätzliches Argument gibt, das die verschiedenen Algorithmen für die überladenen Grundfunktionen enthält.

Beispiel:

```
elem x ys   = any (\y -> (==) x y) ys)
⇝ elem d x ys = any (\y -> (==) d x y) ys)
elem 1 [1,2]  ⇝ elem d_Eq Int  1 [1,2]
```

Wörterbuch/Dictionary $\mathtt{d}_{\mathtt{Eq\ Int}}$ `= (==: prim Eq Int, /=: ...)`

3 Problemquellen:

- Ein konkreter Typ würde gebraucht, läßt sich aber nicht ermitteln (unresolved overload).
- Eine Transformation wird für **Konstanten** nicht durchgeführt, es sei denn, es wird durch durch eine Typspezifikation gefordert.
- Der Aufbau der Wörterbücher terminiert nicht.

b) Unbekannter Typ I: leere Liste und andere leere Objekte

`[1] == [1]` $\mapsto$ `True`, `[] == []` $\mapsto$ `unresolved`, `[1] == []` $\mapsto$ `False`
SML: `[] = []` $\mapsto$ `true`

```
perm [] = [[]]
perm (x:xs) = ... (==) ... ↦  perm :: Eq a => [a] -> [[a]]
perm [] ↦ unresolved
```

`perm` ohne `==` definiert: `[a] -> [[a]]` und dann `perm []` $\mapsto$ `[[]]`
Abhilfe: `perm ([]::[Int])`
Empfehlung: Statt `x == []` besser `null x` benutzen, wobei
`null [] = True` und `null (_:_) = False`
Der Mustervergleich zählt nicht als Gleichheitstest !

c) Überladene "Konstruktoren"

`sqr n = n*n` $\mapsto$ `sqr :: Num a => a -> a`
`sqr` $\mapsto$ `unresolved overload`
`sqr 4.0` $\mapsto$ `16.0`
`sqr = \n -> n*n` $\mapsto$ `unresolved`
Compiler sieht `sqr` als Konstante an, die nicht überladen wird (ohne besondere Aufforderung).

```
sqr :: Num a => a -> a
sqr = \n -> n*n
```
$\surd$

d) Rekursiver Wörterbuchaufbau bei falscher Instanzendeklaration

```
class C a where c :: a -> a
instance C int where c = id
instance C [a] => C a where c = id
```

Wird akzeptiert, passiert nichts Böses.
`c 5` $\mapsto$ `5`   `c []` $\mapsto$ `unresolved overload`
`c [1]` $\mapsto$ nicht terminiert bis Speicher verbraucht
Es wird ein Wörterbuch für `C[Int]` konstruiert mit Verweis auf das für `C [[Int]]`, das einen auf `C [[[Int]]]` hat usw. (normal umgekehrt; z. B. Wörterbuch `Eq[Int]` verweist auf `Eq Int`.

e) Unbekannter Typ II

`fromInteger :: Num a => Int -> a`
`fromInteger 3` $\mapsto$ `unresolved overload` (in welchen Typ transformieren ?)
`fromInteger 3 :: Float` $\mapsto$ `3.0`

Anderes Beispiel: Klasse der auflistbaren Typen
`class  List a where list :: [a]` (Auflistung aller Elemente im Typ)
`instance List Bool where list = [False,True]`
`instance (List a, List b) => List(a,b) where`
$\underbrace{\texttt{list}}_{(a,b)}$ `= [(x,y) | x <-` $\underbrace{\texttt{list}}_{a}$`, y <-` $\underbrace{\texttt{list}}_{b}$ `]`
`list` $\mapsto$ `unresolved`
`list :: [Bool]` $\mapsto$ `[False,True]`
`[x && y] | (x,y) <- list]` $\mapsto$ `[False,False,False,True]`
`card = length list` $\mapsto$ Fehler, Konstanten nur nach Aufforderung überladen
Vorher:
`card :: Int ?` Überladung nicht ausgedrückt
`card :: List @ => Int` wird als 'ambigous' abgelehnt, denn links von `=>` steht eine Typvariable, die rechts nicht vorkommt.
$\Rightarrow$ `card` wäre sowieso unbenutzbar, da man nie angeben kann, auf welchen Typ es sich bezieht (man bräuchte eine Funktion `Type -> Int`; solche Funktionen gibt es hier nicht).

4.5 Ein–/Ausgabe

4.5.1 Überblick

Ein–/Ausgabeoperationen dienen der Interaktion mit der Umwelt.

a) Teilaufgaben

- Dateien öffnen und schließen (etwa in C: `fopen, fclose`).
- Zeichen oder Strings lesen und schreiben (etwa in C: `scanf`, `printf`, `getchar`, `putchar`).
- Strings in andere Werte umwandeln und umgekehrt.

b) Ein–/Ausgabe und Seiteneffekte

`getchar :: () -> Char`	muß in einer rein funktionalen Welt bei jedem Aufruf das gleiche Zeichen liefern.
`putchar :: () -> Char`	würde bei verzögerter Auswertung sein Argument garnicht auswerten.

c) SML

Hat imperative Ein–/Ausgabe mit Seiteneffekten (hier die ältere und einfachere Variante).

- Öffnen/Schließen:
  `type instream` (zum Lesen geöffneter Files)
  `open_in : string -> instream` (öffnet Datei (`string`) zum Lesen)
  `std_in : instream` (Standardeingabe, meist Tastatur)
  `close_in : instream -> unit` (Schließen eines Files)
  `type outstream` (zum Schreiben geöffneter Files)
  `open_out : string -> outstream` (öffnet zum Überschreiben)
  `open_append : string -> outstream` (öffnet zum Anhängen an File)
  `std_out, std_err : outstream` (Standardausgabe)
  `close_out : outstream -> unit` (Schließen eines Files)
- Lesen/Schreiben:
  `output : oustream * string -> unit` (String in File schreiben)
  `input : instream * int -> string` (n Zeichen aus File lesen)
  `input_line : instream -> string` (Zeile aus File lesen)
  `end_of_stream : instream -> bool` (Fileende feststellen)

Die meisten dieser Funktionen verändern ihr Streamargument als Seiteneffekt.

d) Stringfunktionen

Vom 'alten' schwerfälligen System (welches System ??? A.d.L.) sollen hier nur einige elegante Überreste angeführt werden. Moderner ist das monadische System.

4.5.2 Stringfunktionen

a) Zeilen

`lines :: String -> [String]` (Dateiinhalt → Liste der Zeilen)
`unlines :: [String] -> String`
Beispiel: `"ab\n\ncd"` $\overset{\texttt{lines}}{\mapsto}$ `["ab","","cd"]`

b) Konvertieren in Strings

`type ShowS = String -> String` (Motiv: Eleganz und Effizienz)
`showChar :: Char -> ShowS | showChar = (:)`
`showString :: String -> ShowS | showString = (++)`
`shows :: Text a => a -> ShowS, shows 17 "xy"` ↦ `"17xy"`
`show :: Text a => a -> String`
`show x = shows x "", show [1]` ↦ `"[1]," show ['a']` ↦ `"\"a\""`
`show' :: a -> String` (primit. vordef. Fkt. des Interpreters für Meldungen)

4.5.3 Dateileseprimitiv

Öffnen und Lesen einer Datei.

```
primFopen :: String -> a -> (String -> a)
primFopen    name      x        f       =
   if  Datei "name" existiert und lesbar then f(Inhalt der Datei) else x
```

Beispiel: Anzahl der Zeilen in einer Datei bzw. −1, wenn die Datei nicht vorhanden ist: `linecount name = primFopen name (-1) (length.lines)`
Von `primFopen` abgeleitet:

```
openfile :: String -> String
openfile name =
   primFopen name (error("can't open file" ++ name)) id
```

4.5.4 Bildschirmdialog

a) Ergebnisdarstellung im Interpreter

- Ergebnistyp allgemein (nicht die folgenden Sonderfälle (A.d.L. ???)): Interpreter wendet `show' :: a -> String` auf das Ergebnis an und gibt diese Zeichenkette aus.
- Ergebnistyp `String`: Zeichenkette wird direkt ausgegeben.
- Ergebnistyp `Dialogue` oder `IO()`: Ergebnis wird gar nicht dargestellt. Es wird aber ausgewertet. Während dieser Ausgabe kommt es evtl. zu Ein–/Ausgabeereignissen.

b) Dialog–Operationen

`interact, run :: (String -> String) -> Dialogue`
`interact f, run f` bewirken:
$\boxed{\text{Tastatur}}$ → `String` → `f` → `String` → $\boxed{\text{Bildschirm}}$.
Der Dialog entsteht durch verzögerte Auswertung der Eingabe. Es wird immer so viel ausgewertet, wie ohne weitere Kenntnis der Eingabe möglich ist. wenn die Eingabe gebraucht wird (z. B. bei Muster `x:xs`), wartet das System darauf.

`interact f`: Jedes Zeichen wird 'ge–echot'. Zeichen können mit BACKSPACE gelöscht werden. Erst bei RETURN wird die Zeile an den Rechner geschickt. CONTROL–D zeigt das Ende der Eingabe an.

`run f`: Eingegebene Zeichen werden nicht 'ge–echot' und gehen direkt an den Rechner. das gilt auch für BACKSPACE, RETURN und CONTROL–D. Das Ende der Eingabe ist nicht anzeigbar. CONTROL–C wirkt noch.

c) Beispiel 1

`toUpper :: Char -> Char` wandelt Klein– nach Großbuchstaben.
`map toUpper :: String -> String`

	Eingabe	Bildschirm
`interact (map toUpper)`	abc	abc (Echo)
	<RETURN>	<nl>ABC<nl>
		(↑ Echo und Ausgabe)
	<CONTROL–D>	<PROMPT>
`run (map toUpper)`	abc	ABC
	<RETURN>	<nl>
	<CONTROL–D>	
	<CONTROL–C>	{Interrupted.}
		<PROMPT>

4.6 Logische Sprachen (Prolog)

Ein Prologprogramm definiert „Prädikate“ (Relationen); an das Programm kann man verschiedene Anfragen stellen:

- Gilt eine bestimmte Aussage?
- Für welche x gilt eine Aussage?

Als Antwort erhält man *yes*, *no* bzw. einen Wert für x. Es kann allerdings auch vorkommen, daß das Programm nicht terminiert.

4.6.1 Konkatenation von Listen

Listen in Prolog sind z.B. $[1,2,3]$ oder $[\,]$. Allgemein ist $[X|Xs]$ eine Liste mit einem ersten Element X und einem Rest Xs.

Wir definieren nun eine Relation $app(Xs, Ys, Zs)$, die bedeuten soll, daß die Liste Ys an die Liste Xs angehängt die Liste Zs ergibt.

```
app([], Ys, Ys).                          (1)
app([X|Xs], Ys, [X|Zs]):-app(Xs, Ys, Zs).  (2)
```

Nun können wir einige Anfragen an dieses Programm stellen:

1. $?- app([1],[2],[1,2])$.
   Regel (1) wird, allerdings mit neuen, freien Variablen ($app([\,], Ys_1, Ys_1)$) mit der Anfrage verglichen und gleichgesetzt (unifiziert).
   Da dies fehlschlägt, wird im 2. Schritt alles zurückgesetzt und die nächste Wahlmöglichkeit – Regel (2) – angewendet. Es wird also $app([X_2|Xs_1], Ys_2, [X_2, Zs_2])$ mit der Anfrage unifiziert, d.h. $X_2 = 1$, $Xs_2 = [\,]$, $Ys_2 = [2]$ und $Zs_2 = [2]$.
   Nun folgt der rekursive Aufruf $app(Xs_2, Ys_2, Zs_2)$ bzw. $app([\,],[2],[2])$.
   Zuerst wird wieder Regel (1) angewendet, also mit $app([\,], Ys_3, Ys_3)$ unifiziert, d.h. $[\,] = [\,]$, $Ys_3 = [2]$ und $[2] = [2]$.
   Da Regel (1) keine rechte Seite hat, ergibt sich die Antwort *yes*.

2. $? - app([1],[2],[3])$.
   Regel (1) schlägt wieder fehl, auch die zweite Regel läßt sich nicht unifizieren.
   Da es keine weiterer Möglichkeit mehr gibt, lautet die Antwort *no*.

3. $? - app([1],[2],Zs)$.
   Regel (1) schlägt fehl, die zweite Regel läßt sich dagegen mit der Anfrage unifizieren ($X_2 = 1, Xs_2 = [\,], Ys_2 = [2], Zs = [1|Zs_2]$).
   Es folgt der rekursive Aufruf $app([\,],[2],Zs_2)$, der mit Regel (1) unifiziert wird ($Ys_2 = [2], Zs_2 = [2]$).
   Das System hält an und gibt die erste Antwort aus: $Zs = [1,2]$
   Mit der Eingabe „;“ kann der Benutzer die Suche nach weiteren Lösungen anstoßen; da aber die zweite Regel fehlschlägt und es keine anderen Möglichkeiten gibt, lautet die zweite Antwort *no*, d.h. es gibt keine weiteren Lösungen.

4. $? - app(Xs,[2],[1,2])$.
   Regel (1) schlägt fehl (Ys_2 müßte zugleich [2] und [1,2] sein); Regel (2) ist erfolgreich ($X_2 = 1, Zs_2 = [2]$) und führt zum rekursiven Aufruf $app(Ys_2,[2],[2])$.
   Diese Anfrage läßt sich mit Regel (1) unifizieren ($X_2 = [\,], [2] = [2]$) und führt zur ersten Antwort des Systems: $Xs = [X_2|Xs_2] = [1]$.
   Das Anfordern weiterer Lösungen ergibt die Antwort *no*, da der rekursive Aufruf in der Regel (2) fehlschlägt.

5. $? - app([1],Ys,Zs)$.
   1. Antwort: $Zs = [1|Ys]$, d.h. Ys bleibt bei der Abarbeitung frei. Diese Antwort faßt unendlich viele Lösungen zusammen.
   2. Antwort: *no*.

6. $? - app(Xs,Ys,[1])$.
   Die Antwort ist *no*, da auch die zweite Regel fehlschlägt (Ys müßte zugleich [1] und [ ] sein).

7. $? - app(Xs,Ys,Zs)$.
   1. Antwort: $Xs = [\,], Ys = \_1, Zs = \_1$. Dabei steht _1 für eine neue, freie Variable.
   Das Anfordern weiterer Lösungen führt zu der Lösung $Xs = [\_2], Ys = \_3, Zs = [\_2|\_3]$, die Wiederholung zur Lösung $Xs = [\_4,\_5], Ys = \_6, Zs = [\_4,\_5|\_6]$ usw.

8. $? - app(Xs,Xs,Xs)$.
   Die erste Antwort lautet $Xs = [\,]$, das Anfordern weiterer Lösungen führt zu einer Endlosschleife.

9. $? - app(Xs,Ys,[1|Ys])$.
   Als Lösung erwartet man eigentlich $Xs = [1], Ys = \_1$. In der Praxis erhält man stattdessen eine Endlosschleife, da Prolog bei der Unifikation aus Effizienzgründen den Test wegläßt, ob eine Variable im Term, mit dem sie unifiziert werden soll, vorkommt; Regel (1) schlägt daher nicht fehl.

4.6.2 Syntax einer Teilsprache

Syntaxelemente		Beispiel
c	Konstantensymbole, „Atome“	`1,2,[],a,b,otto`
f	Konstruktoren, „Funktionssymbole“, beliebige, klein geschriebene Namen	`leaf, node`
p	Prozedurnamen, „Prädikatsymbole“, klein geschrieben	`app`
Literale	(Prozeduraufrufe und Prozedurköpfe)	$p(t_1, \ldots, t_n)$ für $n > 0$ und p für $n = 0$
X	Variablennamen, groß geschrieben	`Xs, X`
Terme t	Variablen, Konstanten oder Funktionssymbole	`X, c,` $f(t_1, \ldots, t_n)$ für $n > 0$ und t_i Terme
Regeln	L_0. (Fakt) oder $\underbrace{L_0}_{\text{Kopf}} :- \underbrace{L_1, \ldots, L_m}_{\text{Rumpf}}$ $(m > 0)$	`a(X,Y):-b(X),c(Y).`
Anfrage	$\underbrace{L_1, \ldots, L_m.}_{\text{Ziele}}$ $(m > 0)$	a(X,3).
Programm	Folge von Regeln	

Alle Regeln mit gleichem p und gleicher Stelligkeit m im Kopf bilden zusammen eine Prozedurdefinition, die ein Prädikat p/m definiert. (Beispiel: `app/3`)
Daher definieren Konstruktoren mit verschiedener Stelligkeit bei gleichem Namen verschiedene Prädikate.
Variablen, die in einer Regel oder Anfrage genau einmal vorkommen, darf man durch „_“ ersetzen.

4.6.3 Arithmetik

Falls man z.B. die Länge einer Liste bestimmen will, so kann man dies in einer funktionalen Sprache wie folgt tun:

```
length []     = 0
length (_:xs) = 1 + length xs
```

Formuliert man nun ein analoges Prolog-Programm, so erhält man nicht das gewünschte Ergebnis:

```
length([],0).
length([_|Xs],1+N):-length(Xs,N).
```

Denn die Anfrage `length([a,b],2).` ergibt `No` und `length([a,b],N).` liefert als Antwort `N = 1+(1+0)`.

Die arithmetischen Operatoren $+, -, *, /$ werden als Konstruktoren aufgefaßt; $1+1$ ist ein Term und daher verschieden von 2. Damit verbieten sich auch Anfragen wie $1 + 1$., da eine Anfrage nicht nur aus einem Term bestehen darf.

In Prolog existiert zur Lösung dieses Problems ein vordefiniertes Infix-Prädikat `is/2`:

t_1 `is` t_2

Dabei ist t_2 ein volldefinierter arithmetischer Term sein (d.h. es kommen keine freien Variablen vor). t_2 wird zu einem Wert v ausgewertet, mit dem dann t_1 unifiziert wird.

`X is 1+1.`	$\leadsto$	`X=2`
`2 is 1+1.`	$\leadsto$	`Yes`
`1+1 is 1+1.`	$\leadsto$	`No` (`1+1` nicht unifizierbar mit `2`
`X=3, Y is X+1.`	$\leadsto$	`X=3, Y=4`
`Y is X+1, X=3.`	$\leadsto$	Fehler! (Reihenfolge beachten!)

Nun können wir auch das Prädikat `length/2` richtig definieren:

```
length([],0).
length([_|Xs],N1):-length(Xs,N), N1 is N+1.
```

Falls man dieses Prädikat mit `length(Xs,2)` aufruft, terminiert das Programm nach Ausgabe von `Xs=[_1,_2]` und Anfordern weiterer Lösungen nicht. Daher gibt es meistens ein vordefiniertes Prädikat `length/2`, das am Ende terminiert.

Hier fehlen weitere Beispiele...

4.6.4 Variablenbindung

Eine Variable kann in Prolog vier verschieden Bindungszustände einnehmen:

völlig frei	in der Anfrage `fac(3,F)` ist `F` völlig frei.
„shared“	Nach einem `X=Y` zeigen `X` und Y auf die gleiche Hilfsvariable. Eine Anfrage `X=Y` ergibt `X=_1, Y=_1` und eine Anfrage `X=Y,X=3` ergibt `X=3,Y=3`.
teilgebunden	z.B. `X=f(Y)`, d.h. eine Variable wird an einen Ausdruck gebunden, in dem freie Variablen vorkommen.
vollgebunden (ground)	`X=`t, wobei t ein Term ohne freie Variablen ist.

In Prolog gibt es – anders als in rein funktionalen Sprachen – Seiteneffekte, allerdings können Variablen nur stärker gebunden werden (`X=1,X=2` funktioniert nicht, ergibt `No`).

Der Bindungsstatus einer Variable kann zur Laufzeit mit Hilfe von vordefinierten Prädikaten bestimmt werden:

`ground(`t`)`	erfolgreich, falls t keine freien Variablen enthält.
`var(`t`)`	erfolgreich, falls t eine freie Variable ist.
t_1 `==` t_2	vergleicht zwei Terme strukturell (keine Unfikation!). Für Variablen gilt: `X==Y` $\Longleftrightarrow$ beide Variablen sind „shared“.

4.6.5 Ein- und Ausgabe

Zur Ein- und Ausgabe in Prolog existieren zwei vordefinierte Prädikate `read/1` und `write/1`.

1. `read(`t`)` liest eine Term t' von der Tastatur ein und unifiziert t und t'. Die Eingabe des Terms t' wird mit „.“ abgeschlossen. Normalerweise wird `read` mit einer freien Variable als Argument aufgerufen: `read(X)` bindet `X` an den eingegebenen Term. Allerdings ist z.B. auch ein Aufruf `read([X,Y])` möglich. Hier muß der eingegebene Term die Form $[t_1, t_2]$ haben; `X` wird dann an t_1, `Y` an t_2 gebunden.

Bei einer anderen Eingabe schlägt der Aufruf fehl, wobei nicht nach einer alternativen Eingabe gefragt wird. Um eine richtige Eingabe zu erzwingen, kann man sich mit folgendem Prädikat behelfen:

```
tryread(X):-read(X); tryread(X).
```

2. `write(`t`)` schreibt t ohne Zeilenwechsel auf den Bildschirm (ein nachgestelltes `nl` erzwingt den Zeilenwechsel). Das Schreiben erfolgt, wenn `write` ausgewertet wird. Bei evtl. späteren Rücksetzen wird die Ausgabe nicht zurückgenommen.
   Es fehlen BEISPIELE

4.6.6 Cut

Der Cut („!“) ist eine Methode, um alternative Wahlmöglichkeiten abzuschneiden.

Habe eine Regel C die Form $A \longleftarrow L_1, \ldots, L_k, !, L_{k+2}, \ldots, L_n$. Wenn das aktuelle Ziel G mit dem Kopf von C unifizierbar ist und $L_1, \ldots, L_k$ erfolgreich sind, dann hat der Cut den folgenden Effekt: C wird ausgewählt, alternative Regeln, die mit G unifizierbar sein könnten, werden ignoriert. Falls ein $L_i, i > k+1$ fehlschlägt, so wird das Backtracking nur bis zum „!“ ausgeführt. Möglicherweise vorhandenen weitere Möglichkeiten in der Berechnung der $L_i, i \leq k$ werden aus dem Suchbaum geschnitten.
Der Cut schlägt fehl, falls das Backtracking den Cut erreicht. Der Suchprozeß geht dann mit der Wahlmöglichkeit weiter, die aktuell war, bevor die Regel mit dem Cut ausgewählt wurde.

Man kann den Cut benutzen, um Determinismus in Programmen auszudrücken und so die Berechnung beschleunigen, da weitere Möglichkeiten nicht mehr ausprobiert werden müssen.

Beispiel:

```
minimum(X,Y,X):-X≤Y,!.
minimum(X,Y,Y):-X>Y,!.
```

Bei der Berechnung des Minimums von zwei Zahlen können wir die Berechnung nach Erfüllung einer der beiden Regeln abbrechen, da die andere nicht mehr erfüllt werden kann.

Die bisherigen Cuts waren „Green Cuts“, da sie zwar die Programmausführung beschleunigen, aber die Semantik des Programms nicht verändern. Man kann diese Cuts einfach weglassen.

Eine zweite Art von Cuts sind sogenannte „Red Cuts“. Jeder Cut, der nicht grün ist, ist rot. Rote Cuts kann man nicht weglassen, ohne die Semantik eines Programms zu verändern. Mit roten Cuts kann man z.B. eine if-then-else Struktur erzeugen:

```
if_then_else(P,Q,R):-P,!,Q.
if_then_else(P,Q,R):-R.
```

4.6.7 Negation