Einführung in die Informatik I

Ein Skript zu der gleichnamigen Vorlesung von Prof. Dr. W. Brauer

Version 56

25. Oktober 1999

Inhaltsverzeichnis

Kapitel 1

Erste Gedanken zur Informatik

1.1 Was ist Informatik?

Im allgemeinen tut man sich schwer eine von allen akzeptierte einfache Antwort auf die Frage zu finden, was Informatik ist. Dies wird auch im Hinblick darauf schwer wenn man bedenkt wie schnell sich dieses Gebiet entwickelt.

Oft trifft man auf die folgende Formulierung: Informatik ist die Wissenschaft, Technik und Anwendung der maschinellen Verarbeitung und Übermittlung von Informationen. Allerdings ist diese Definition etwas zu eingeschränkt. Ich[1] meine, die Informatik läßt sich besser durch die folgenden drei Begriffe charakterisieren.

$$\left.\begin{array}{r}\text{Intelligenz}\\ \text{Formalisierung}\\ \text{Technik}\end{array}\right\} = \quad \textbf{In}\text{telligenz}\textbf{forma}\text{lisierungs}\textbf{t}\text{echn}\textbf{ik}$$

Ziel der Informatik ist die Entwicklung von artifiziellen Assistenzsystemen für den Menschen zur Unterstützung technischer und geistiger Fähigkeiten. Die Modellierung intelligenten Verhaltens erfolgt durch Reduktion der Realität auf Formalismen (logischer Aufbau) und Abstraktion (Beschränkung auf das Wesentliche).

Die Unterscheidung zwischen Reduktion und Abstraktion wird bewußt getroffen um anzudeuten, daß es im allgemeinen mehrerer und zum Teil irreversible Schritte bedarf um von der Realität zu den Modellen und Systemen der Informatik zu kommen. Bei Reduktion geht etwas verloren, daher kann man sie nicht ohne weiters rückgängig machen. Unter Abstraktion versteht man Verallgemeinerung und die Vernachlässigung von Details.

1.2 Aspekte Intelligenten Verhaltens

Man unterscheidet grob zwei Arten von Intelligenz. Die formale Intelligenz der Intellektuellen, des rational denkenden Vernunftmenschen, auf der einen Seite und die Intelligenz des täglichen Lebens, also die natürliche Intelligenz auf der anderen.

[1] Prof. Dr. W. Brauer

Formale Intelligenz	**Natürliche Intelligenz**
Rechnen	Reagieren
Logisches Schließen	Adaptieren
	Lernen
Suchen, Sortieren	
Formelmanipulation	Klassifizieren
Analyse formaler Sprachen	Gestalterkennung
	Bildversehen
Diagnostizieren	Sprachverstehen
Konfigurieren	
Planen	Kommunikation
	Soziales Verhalten
Entwicklung von Verfahren	
und Algorithmen	Prozeßsteuerung

1.3 Klassische Informatik (KI)

Kerninformatik und künstliche Intelligenz

- Vernunftgemäßes Handel und Denken
  - Rechnen mit den Natürlichen Zahlen
- Turingmaschine[2] als Prototyp eines Computers
  - nicht nur mit Zahlen sondern auch mit Zeichen (Symbolen), also Symbolverarbeitung

Logisches Denken baut auf einen "logischen Aufbau der Welt". Mittels Logik und diskreter Mathematik schafft man somit eine künstliche Welt, mit dem Ziel der Determiniertheit. Dazu ist Exaktheit und eine Vollständigkeit der Beschreibung von Nöten. Der Kontakt zur realen Welt wird durch Zivilisation und Technik hergestellt.

Das Denken kann sprachlich formuliert werden, Sprache wird mittels Symbolen aufgeschrieben. Komplexe Gedankengebilde entstehen durch systematische Zusammensetzung aus einfachen Gedanken (Fregesches Kompositionsprinzip). Die Informatik versucht, den "logischen Aufbau der Welt"im Sinne von Carnap[3] im Computer nachzuvollziehen. Die durch das Denken, den Intellekt, erschlossene Welt wird im Computer virtuell manipulierbar; also kann man auch rationales Planen und Handeln durch Computer unterstützen, und virtuelle Denkwelten aufbauen.

Übrigens ist dies Vorgehen auch mit dem der klassischen Physik (insbes. der Mechanik) in Übereinstimmung - darauf basieren im Grunde die Ingenieurwissenschaft und die technischen Anwendungen der klassischen Informatik.

Komplexe Systeme versteht man, d.h. die Gesetze für ihr Verhalten (die Naturgesetze der Physik) findet man, indem man die Systeme solange (rekursiv) in Subsysteme zerlegt, bis man Systeme erhält, die man mathematisch-analytisch behandeln kann. Das Verhalten des Gesamtsystems erhält man (rückwärts) durch (additives) Zusammensetzen. Stichwort: John Stuart Mills[4] Prinzip der "Gleichförmigkeit der Natur"(1843). Reduktion, Abstraktion und Fregesche Modularität ermöglichen die Konstruktion einer mit der Realität nur lose verbundenen Welt der maschinellen Symbolverarbeitung.

Dazu ein Beispiel: Die Platzreservierung für den ICE. Ein solches System ist ein schönes Informatik-System. Aber es hat nur einen losen Kontakt zur realen Welt.

[2] von Turing eingeführtes Konzept einer abstrakten Rechenmaschine
[3] Rudolf Carnap, (1891-1970), deutscher Philosoph
[4] John Stuart Mill , (1806-1873), britischer Philosoph

Es reserviert nicht Plätze sondern nur Ansprüche auf Plätze: Stellen Sie sich vor, Sie kommen mit Platzkarte auf den Bahnhof, steigen in den Wagen und Ihr Platz ist besetzt oder beschädigt. Was dann? Hier hilft nur zivilisiertes Verhalten oder technische bzw. manuelle Reparatur.

Die Systeme der traditionellen Informatik bilden eine vom menschlichen Intellekt geschaffene künstliche Welt. Der Kontakt dieser Systeme zur realen Welt geschieht über technische und zivilisatorische Schnittstellen.

1.4 Neuere Informatik (NI)

Verhalten in natürlicher Umwelt – Verfolgen von Zielen

Multiagentensysteme (Nicht-Fregesche[5]-Kombination)

- Neurale Netze
- Fuzzy Logic und Fuzzy Control
  - Reduktion auf kontinuierliche Mathematik (Analysis)
  - Spezialisierung auf Diskretheit und Sequentialität: Turingmaschine, klassische Logik, ...

Zu diesen Themen existieren verschiedene Forschungsprojekte

- ACON (Adaptive Control)
- Raumkognition: Qualitative Repräsentation von Bewegungsabläufen
- Sozionik[6]

Für die Einbettung von Informatiksystemen in die reale Welt existieren die folgenden Strategien:

- ohne Annahme totaler Beherrschbarkeit
- mit weitgehender Zuverlässigkeit
- mit direktem Sensor beziehungsweise Aktuator[7]-Kontakt

Die erfaßten Sensordaten werden zunächst vorverarbeitet um anschließend analysiert zu werden (Wissensextraktion). Die Verarbeitung der Daten geschieht unter Verwendung physikalischer und biologischer Prinzipien, psychologischer und mathematischer Verfahren (computational intelligence). Ein solches System sollte selbstverständlich über vernünftige Optimalität, robuste Brauchbarkeit und weitgehende Zuverlässigkeit verfügen.

[5] Frege = Komposition
[6] interdisziplinäres Forschungsgebiet zwischen Soziologie und Verteilter KI
[7] ?

Kapitel 2

Einführende Überlegungen

2.1 Vorgehensweise in der Informatik

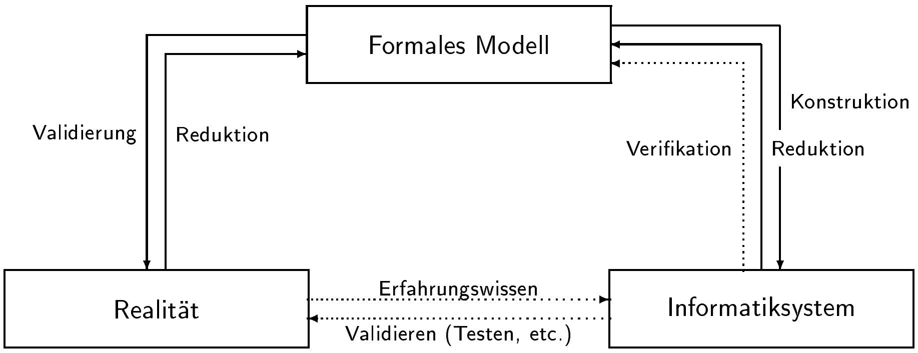
In der Informatik ist es wesentlich Wissen zu gewinnen, es wiederauffindbar zu speichern, zu verarbeiten und weiterzugeben sowie dieses Wissen anzuwenden, also Intelligenz erforderliche Aufgaben erledigen.

Beim Übergang von der Realität in den Rechner (Software Engineering) sind grob gesehen die folgenden Schritte nötig:

1. Analysieren
2. Formalisieren
3. Algorithmisieren
4. Implementieren

In der Informatik zielt die Modellbildung auf die Darstellung der wesentlichen Strukturen, Zusammenhänge und Vorgänge eines Anwendungsgebietes, unter dem Gesichtspunkt einer gegebenen Aufgabenstellung. Dies erfolgt durch formale Mittel wie etwa Programmiersprachen, Datenstrukturen, logische Formeln oder Grafiken. Solche formalen Modelle sind durch die zu modellierenden Realitätsaspekte, die theoretischen Möglichkeiten und Grenzen der Formalen Beschreibung und durch die Eigenschaften der zu konstruierenden Informationssysteme bestimmt.

Als **Informatiksystem**bezeichnet man die technische Realisierung eines formalen Modells.



Bei der Validierung wird überprüft ob das gewonnene Model noch der Wirklichkeit entspricht, da Realitätsverlust wegen Nichtübereinstimmung von Modell und Wirklichkeit eine häufige und schwer zu entdeckende Fehlerquelle ist. Die Validierung eines Modells gegen eine Spezifikation durch logische Schlüsse wird als Verifikation bezeichnet.

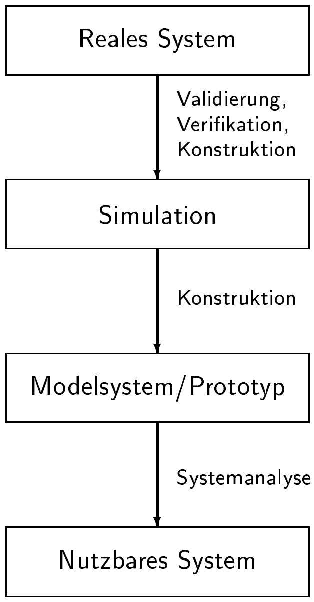
2.2 Systeme

- **System**: ist eine Kollektion von Gegenständen (Systemkomponenten / -bausteine) mit Beziehungen zwischen diesen.
- Systeme sind in eine Umgebung eingebettet daher unterscheidet man:
  - **abgeschlossene Systeme:** die Bausteine dieser Systeme haben keine Beziehung zu Gegenständen in der Umgebung; die Systemgrenzen sind fest und zeitlich nicht veränderlich. → Idealisierung
  - **offene Systeme:** es existieren Beziehungen zu anderen Gegenständen in der Umgebung (Schnittstelle des Systems/des Bausteins); die Zugehörigkeit von Gegenständen zu einem System kann wechseln.
  - **statische Systeme:** die zeitliche Veränderung spielt im gegebenen Zusammenhang keine Rolle. → Idealisierung
  - **dynamische Systeme:** das System ist in Bezug auf folgende Aspekte zeitlich veränderlich:
    1. Beziehung der Komponenten untereinander
    2. Menge der Komponenten (Verschiebung der Systemgrenzen, Verschwinden, Neubildung)
    3. Eigenschaften der Bausteine ändern sich
- Die Änderungen eines Systems beziehen sich auf dessen Zustand, das System als solches bleibt gleich. Die zeitliche Folge der Veränderungen des Systems bezeichnet man als das **Verhalten des Systems**.
- In Hinblick auf die Vorhersagbarkeit eines Systems kann man zwischen determnistischen und indetermnistischen Systemen unterscheiden.
  - **deterministische Systeme:** die Vergangenheit bestimmt das zukünftige Verhalten.
  - **indeterministische Systeme:** zukünftiges Verhalten ist zufällig oder wird nicht vorhersehbar von außen beeinflußt
- Der Systembegriff ist **rekursiv** da die Komponenten selbst Systeme sein können. In Bezug auf die Bedeutung, der Einzelkomponenten für das Verständnis des Gesamtsystems, kann man zwei Fälle unterscheiden.
  1. Für das Verständnis des Gesamtsystems sind nur die Zustände der Teilsysteme (Objekte) relevant. → Black Box
  2. Für das Verständnis des Gesamtsystems müssen die Beziehungen und mögliche zeitliche Veränderungen innerhalb der Teilsysteme betrachtet werden. → White Box
- Der rekursive Begriff ist wichtig weil er skalierbar ist. D.h. die Eigenschaften kleinerer Systeme lassen sich auf größere Systeme Übertragen.
- Man kann des weiteren zwischen künstlichen und natürlichen Systemen (Artefakte) unterscheiden. Künstliche Systeme dienen einen bestimmten Zweck, dem Systemziel.

– Berechnung von Funktionen (Start Stop)

$$\left.\begin{array}{l}\text{– Prozeßüberwachung}\\ \text{– Eingebettete Systeme}\\ \text{– Adaptive Systeme}\end{array}\right\}\quad \text{reaktive Systeme}$$

Die Konstruktion von Informatiksystemen



Siehe auch: Gerhard Goos: **Vorlesungen über Informatik** Band 1; 2.Auflage; 1.3 Systeme; Seite 18ff

2.3 Signale, Nachrichten, Informationen, Daten

Unter einem **Signal** versteht man die zeitliche Veränderung einer physikalischen Größe. Insbesondere zeichnen sich Signale durch ihre kurzzeitige Existenz aus. Im weiteren sind folgende Begriffe von Bedeutung:

- **Signalparameter**: die physikalischen Eigenschaften des Signals (zum Beispiel: Amplitude, Frequenz)
- **Signalübertragung**: die Weitergabe des Signals
- **Übertragungsmedium**: Art des Mediums, über welches das Signal weitergegeben wird (zum Beispiel: Licht, Schall)

Die dauerhafte Darstellung einer Mitteilung (auf einem physikalischen Medium) bezeichnet man als **Inschrift** oder schriftliche Darstellung. Unter dem Schriftträger versteht man das Medium, welches die Inschrift trägt.

Durch die Abstraktion vom Medium, der Übertragung oder Darstellung, und von Details der Signale bzw. Inschriften ergibt sich der Begriff der **Nachricht**: einer konkret (physikalisch) repräsentierten Mitteilung.

Unter dem **Bezugsystem** versteht man die Kenntnisse die man benötigt um einer Nachricht eine Bedeutung zuzuordnen. Eine Nachricht kann, abhängig vom jeweiligen Bezugssystem bzw. von der **Interpretationsvorschrift**, über verschiedene Interpretationen und Bedeutungen verfügen.

Ist einer Nachricht mit keiner Interpretationsvorschrift verbunden so ist sie möglicherweise mehrdeutig. Dazu ein Beispiel:

x^2 Der Buchstabe x mit dem oberen Index 2
Polynom mit der unbekannten x
die Vorschrift quadriere x
das Quadrat von x

Mehrdeutige Nachrichten besitzen in dem selben Bezugssystem unterschiedliche Bedeutungen. Widersprüchlichen Nachrichten kann man in einem Bezugsystem keinen Sinn zuordnen.

- **Information**: ist die Bedeutung einer Nachricht
- **Datum**: bezeichnet das Paar Nachricht und zugeordnete Information (Plural Daten)
- **Datenverarbeitung**: ist die bedeutungstreue Verarbeitung von Nachrichten. Das heißt das Ergebnis der Verarbeitung hat als Information eine Relevanz bzw. Bedeutung.

Iteniertes Vorgehen:

"Die Temperatur beträgt 20° C."

Dieser Satz stellt eine Nachricht dar die eine Vorschrift für ihre Interpretation enthält. Damit ist dieser Satz also ein Datum. Die Angaben von "Temperatur" und "Celsius" sind die Vorschrift zur Interpretation von "20°".

Siehe auch: Gerhard Goos: **Vorlesungen über Informatik** Band 1; 2.Auflage; 1.1 Signal, Datum, Information; Seite 1ff

2.4 Wissen

Wissen befähigt uns, mit Nachrichten etwas anzufangen zu können, sie zu verstehen. In der Informatik ist vor allem mitgeteiltes, geschriebenes Wissen wichtig. Zur Darstellung dieses Wissens existieren Wissensrepräsentationen und Datenstrukturen. Man kann zwischen scharfem (Logik) und unscharfem Wissen (Fuzzy-Methoden)unterscheiden.

Ein Beispiel für die Darstellung von Wissen sind die natürlichen Zahlen:

Darstellung:	Strichzahlen:	$\|, \|\|, \|\|\|, \|\|\|\|, \|\|\|\|\|, ...$
(Repräsentation)	Dezimalzahlen:	$1, 2, 3, 4, 5, ...$
	Binärzahlen:	$1, 10, 11, 100, 101, ...$
Operationen:	Dekrement:	-1
	Inkrement:	$+1$

Die unterste Ebene der Wissensrepräsentation ist die Codierung von Mitteilungen durch Zeichenfolgen. So existieren:

- verschiedene Alphabete (geordnete Zeichenvorräte)
- verschiedene Zuordnungen von Bestandteilen der Mitteilungen und Zeichenfolgen.

Siehe auch: Gerhard Goos: **Vorlesungen über Informatik** Band 1; 2.Auflage; 1.1.1 Wissen; Seite 4ff

2.5 Informationssysteme

Wir definieren A als eine Menge von Informationen. Eine Information bezeichnet dabei den abstrakten Inhalt einer Nachricht. Durch die Elemente der Menge

A sind die Informationen repräsentationsunabhängig gegeben. R ist eine Menge von Repräsentationen. Eine **Repräsentation** ist die äußere, konkrete Form einer Nachricht (im Gegensatz zu ihrem Inhalt). Die Interpretation I ordnet einer gegebenen Repräsentation r (einer Nachricht) einen abstrakten Informationsgehalt $I_{[r]}$ zu. Eine Interpretation entspricht also einer Abbildung:

$$I : R \to A$$

(A, R, I) wird als **Informationssystem** bezeichnet. Damit entspricht ein Informationssystem dem Begriff einer Abbildung aus der Mathematik.

Beispiel: (Repräsentationssysteme für natürliche Zahlen)

$A =$	Zahlen	die Menge aller Strichzahlen
$R =$	Strichzahlen	
$I_{[\vert\vert\vert]} =$	die Zahl "drei"	

Siehe auch: Manfred Broy: **Informatik Eine grundlegende Einführung** Band 1; 2.Auflage; 1 Information und ihre Repräsentation; Seite 8ff

2.6 Aussagen und Boolesche Algebra

2.6.1 Aussagen

Aussagen sind sprachliche Gebilde, die Zustände der Welt oder Ausschnitte von ihr beschreiben.

"Eine Aussage ist ein sprachliches Gebilde, von dem es sinnvoll ist zu sagen es ist wahr oder falsch. "(Aristoteles 284 - 322 v. Chr.)

Diese Definition ist aber nicht zufriedenstellend, da sie zum Beispiel Selbstbezogenheit zuläßt. Um zu einer präzisen Definition dieses Begriffes zu gelangen, muß man ein eingeschränktes Repräsentationssystem für Aussagen und ihre Kombinationen verwenden. Diese Einschränkung, auf ganz bestimmte sprachliche Formeln, wird durch ein System von Aussageformen definiert. Ein solches System wird durch die Aussagenlogik bereitgestellt. In der Aussagenlogik, lassen sich Aussagen durch Konstruktion (Frege) aus einfacheren (elementaren) Aussagen darstellen. Aus den Wahrheitswerten der elementaren Aussagen ergibt sich der Wahrheitswert der zusammengesetzten Aussage

2.6.2 Boolesche Terme

Durch ein Bezugsystem (die reale Welt oder ein Ausschnitt von ihr) wird festgelegt ob eine Aussage wahr oder falsch ist. Für den weiteren Umgang mit Aussagen wollen wir im folgenden kurz einige Begriffe definieren:

- **Elementare (atomare) Aussagen**: Eine Aussage ist genau dann elementar, wenn sie nicht aus weiteren Aussagen zusammengesetzt ist. Elementare Aussagen ohne Identifikatoren sind entweder wahr oder falsch.
- **Zusammengesetzte Aussagen**: entstehen durch die Verknüpfung elementarer Aussagen durch logische Operatoren ("und", "oder", "nicht", ...)
- **spezielle Aussagen**: True: ist immer wahr
  False: ist immer falsch
- **Identifikatoren**: können als Platzhalter für später frei dafür einsetzbare Aussagen aufgefaßt werden. Von solchen Aussageschemata läßt sich in der Regel nicht einfach sagen, ob sie wahr oder falsch sind, da dies von der Belegung mit den Wahrheitswerten abhängt.

Sei E als die Menge von Bezeichnungen von elementaren Aussagen definiert und ID eine (unendliche, abzählbare) Menge von Identifikatoren. Seien weiterhin $x, y, z \in ID$. Die Menge der Booleschen[1] Terme mit (freien) Bezeichnern aus ID ist unendlich und definiert sich induktiv wie folgt:

1. True und False sind Boolesche Terme.
2. Alle atomaren Aussagen E und alle Identifikatoren ID sind Boolesche Terme.
3. Ist t ein Boolescher Term so ist auch $\neg t$ ein Boolescher Term
4. Sind t_1 und t_2 Boolesche Terme, so sind auch $(t_1 \vee t_2)$, $(t_1 \wedge t_2)$, $(t_1 \Rightarrow t_2)$ und $(t_1 \Leftrightarrow t_2)$ Boolesche Terme

Die Booleschen (logischen) Operatoren sind:

$$\left.\begin{array}{lll} \neg t_1 & \text{“nicht”}t_1 & \text{Negation} \\ t_1 \wedge t_2 & t_1 \text{ “und”}t_2 & \text{Konjunktion} \\ t_1 \vee t_2 & t_1 \text{ “oder”}t_2 & \text{Disjunktion} \\ t_1 \Rightarrow t_2 & t_1 \text{ “impliziert”}t_2 & \text{Implikation} \\ t_1 \Leftrightarrow t_2 & t_1 \text{ “ist äquivalent zu”}t_2 & \text{Äquivalenz} \end{array}\right\} \begin{array}{l}\text{Boolesche}\\ \text{Operatoren}\end{array}$$

Ordnet man die Booleschen Operatoren bezüglich ihrer Bindekraft, so ergibt sich die folgende Reihenfolge (je kleiner die Zahl desto stärker bindet der Operator):

1. $\neg$
2. $\wedge$
3. $\vee$
4. $\Rightarrow$ und $\Leftrightarrow$

Ungeklammerte Aggregate von Operatoren gleicher Bindungsstärke werden von links nach rechts aufgelöst.

Boolesche Terme enthalten atomare Aussagen und Identifikatoren. Atomare Aussagen stehen für bestimmte konstante Aussagen und Identifikatoren stehen für frei wählbare Aussagen. Es wird gesagt in dem Booleschen Term t kommt der Identifikator x frei vor, falls:

1. t genau aus dem Identifikator x besteht
2. t von der Form $(\neg t_1)$ ist und in t_1 der Identifikator x frei vorkommt
3. t von der Form $(t_1 \vee t_2)$ ist und in t_1 oder t_2 der Identifikator x frei vorkommt

2.6.3 Die Boolesche Algebra der Wahrheitswerte

Wahrheitswerte sind die einfachste und fundamentalste Art von Informationen. Die **Menge der Wahrheitswerte** $\mathbb{B}$ besteht aus genau zwei unterscheidbaren Elementen. Im folgenden wird die Menge $\mathbb{B} =_{\text{def}} \{\mathsf{L}, \mathsf{0}\}$ zur Repräsentation von Wahrheitswerten verwendet.

1. Identität: $\mathbb{B} \to \mathbb{B}$

$$\begin{array}{l} \mathsf{L} \to \mathsf{L} \\ \mathsf{0} \to \mathsf{0} \end{array}$$

[1] George Boole, 1815 - 1864, britischer Mathematiker und Logiker

2. Negation: $\mathsf{not}\colon \mathbb{B} \to \mathbb{B}$

b	not(b)
L	0
0	L

3. Disjunktion: $\mathsf{or}\colon \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

or	L	0
L	L	L
0	L	0

Or liefert nur falsch wenn beide Elemente falsch sind

4. Konjunktion: $\mathsf{and}\colon \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

and	L	0
L	L	0
0	0	0

And liefert nur wahr wenn beide Elemente wahr sind

5. Implikation: $\mathsf{impl}\colon \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

impl	L	0
L	L	0
0	L	L

6. Äquivalenz: $\mathsf{equiv}\colon \mathbb{B} \times \mathbb{B} \to \mathbb{B}$

equiv	L	0
L	L	0
0	0	L

2.6.4 Interpretation Boolescher Terme

Um Boolesche Terme interpretieren zu können muß für jede atomare Aussage festgelegt sein, ob sie wahr oder falsch ist. **Geschlossene Boolesche Terme**, also Boolesche Terme ohne Identifikatoren, können interpretiert werden, indem man ihnen durch ausrechnen einen Wahrheitswert zuordnet. Boolesche Terme mit Identifikatoren können interpretiert werden, indem ihnen nach der Belegung eines freien Identifikators, mit einem Wahrheitswert, ein Wahrheitswert zugeordnet wird. Die Interpretation Boolescher Terme mit freien Identifikatoren wird im folgenden mathematisch definiert:

Die **Boolesche Belegung** β einer Menge ID ist eine Abbildung $\beta : ID \to \mathbb{B}$ mit $\beta \in \{\mathsf{L},\mathsf{0}\}$ die jedem Identifikator aus der Menge ID einen Wahrheitswert zuordnet. Die Menge der Belegungen wird mit ENV (environment) bezeichnet.

Für eine gegebene Boolesche Belegung β läßt sich jedem Booleschen Term vermöge der Interpretation I_β ein Wahrheitswert zuordnen (für $I_\beta[a]$ ein unabhängig von β festgeschriebener Wert).

$I_\beta[t]$ Interpretation des Termes t bei der Belegung β

$$
\begin{array}{ll}
I_\beta[\mathsf{True}] & =_{\text{def}} \mathsf{L} \\
I_\beta[\mathsf{False}] & =_{\text{def}} \mathsf{0} \\
I_\beta[x] & =_{\text{def}} \beta(x), \qquad \text{für } x \in \mathsf{ID} \\
I_\beta[\neg t] & =_{\text{def}} \mathsf{not}(I_\beta[t]) \\
I_\beta[t_1 \wedge t_2] & =_{\text{def}} \mathsf{and}(I_\beta[t_1], I_\beta[t_2])
\end{array}
$$

Im folgenden wird mit:

1. W_{Bool} die Menge der Booleschen Terme und mit
2. $W_{\text{Bool}}(ID)$ die Menge der Booleschen Terme mit Identifikatoren aus ID

bezeichnet. Ein Boolescher Term kann somit als Abbildung aufgefaßt werden, die jeder Belegung einen Wahrheitswert zuordnet.

$$I : W_{\text{Bool}} \to (\mathsf{ENV} \to \mathbb{B}) \text{ mit } I[t](\beta) = I_\beta[t]$$

$$\text{Beispiel:} \quad I_\beta[x] = \beta(x) \qquad \beta \in \{\mathsf{L}, \mathsf{0}\}, \mathsf{ENV} \to \mathbb{B}$$

Sowohl $(\mathbb{B}, W_{\text{Bool}}, I_\beta)$ als auch $(\mathbb{B}, W_{\text{Bool}}(ID), I_\beta)$ bilden (für beliebige Belegungen β) Informationssysteme, ebenso:

$$((\mathsf{ENV} \to \mathbb{B}), W_{\text{Bool}}(ID), I)$$

Auch für Terme ohne freie Identifikatoren ist durch I_β eine Interpretation gegeben. Für einen geschlossenen Booleschen Term t ist der Wert $I_\beta[t]$ unabhängig von der Belegung β. Daraus folgt, daß ein Boolescher Term ohne Identifikatoren die Informationen L oder $\mathsf{0}$ repräsentiert. Ein Boolescher Term mit freien Identifikatoren repräsentiert eine Abbildung von Belegungen in die Wahrheitswerte, beziehungsweise für eine vorgegebene Belegung wieder einen Wahrheitswert.

Ein Boolescher Term t mit den (freien) Identifikatoren $x_1, ..., x_n$ kann zur Definition einer n-stelligen Booleschen Abbildung (Funktion):

$$f : \mathbb{B}^n \to \mathbb{B}$$

verwendet werden. Die Abbildung ist (für $b_1, ..., b_n \in \mathbb{B}$) durch:

$$f(b_1, ..., b_n) = I_\beta[t]$$

gegeben, wobei für die Belegung β gelte: $\beta(x_i) = b_i$ für alle $i, 1 \leq i \leq n$.

$$\mathbb{B}^n = \underbrace{\mathbb{B} \times ... \times \mathbb{B}}_{\text{Menge aller } n\text{-Tupel}} \Rightarrow \text{d.h. Menge aller } n\text{-Tupel aus } \mathbb{B}$$

2.6.5 Gesetze der Booleschen Algebra

Ein Gleichungsgesetz für die Boolesche Algebra besteht aus den Termen (t_1, t_2) die durch ein $=$, in der Bedeutung der semantischen Gleichheit verknüpft werden. Boolesche Terme heißen äquivalent, wenn $I_{[t_1]} = I_{[t_2]}$, das heißt, wenn $I_\beta[t_1] = I_\beta[t_2]$ für β aus ENV. Im weiteren sind x, y und z Identifikatoren, die für beliebige Boolesche Terme oder Werte stehen.

Die Konstanten true und false sind definiert durch:

$$\begin{array}{lcl} \mathsf{true} & = & (x \vee (\neg x)) \\ \mathsf{false} & = & (\neg \mathsf{true}) \end{array} \qquad (x \text{ ist Identifikator})$$

Im weiteren gelten die folgenden Identitäten:

$$\begin{array}{lcl} x \wedge y & = & (\neg((\neg x) \vee (\neg y))) \\ x \Rightarrow y & = & ((\neg x) \vee (y)) \\ x \Leftrightarrow y & = & ((x \wedge y) \vee ((\neg x) \wedge (\neg y))) \end{array}$$

Weitere Boolesche Operatoren sind wie folgt definiert:

$$\begin{array}{lcll} x \oplus y & = & (x \wedge (\neg y)) \vee ((\neg x) \wedge y) & \text{exklusives Oder} \\ x / y & = & (\neg x) \vee (\neg y) & \text{NAND-Funktion} \\ x \| y & = & (\neg x) \wedge (\neg y) & \text{NOR-Funktion} \end{array}$$

Gesetze der Booleschen Algebra	
$\neg\neg x = x$	Involutionsgesetz
$x \wedge y = y \wedge x$ $x \vee y = y \vee x$	Kommutativgesetz
$(x \wedge y) \wedge z = x \wedge (y \wedge z)$ $(x \vee y) \vee z = x \vee (y \vee z)$	Assoziativgesetz
$x \wedge x = x$ $x \vee x = x$	Idempotenzgesetz
$x \wedge (x \vee y) = x$ $x \vee (x \wedge y) = x$	Absorptionsgesetz
$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$ $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$	Distributivgesetz
$\neg(x \wedge y) = (\neg x) \vee (\neg y)$ $\neg(x \vee y) = (\neg x) \wedge (\neg y)$	Gesetz von De Morgan
$x \vee (y \wedge \neg y) = x$ $x \wedge (y \vee \neg y) = x$	Neutralitätsgesetz

Daß die Gesetze der Booleschen Algebra im Sinne des eingeführten Interpretationsbegriffes tatsächlich vernünftig gewählt sind beweist das folgende Theorem:

Theorem: (Verträglichkeit der Gesetze der Booleschen Algebra mit der Interpretation). In allen als Gesetze der Booleschen Algebra gegebenen Gleichungen sind die linke und die rechte Seite äquivalent.

Dies bedeutet, für jedes Gesetz der Booleschen Algebra $t_1 = t_2$ gilt $I_\beta[t_1] = I_\beta[t_2]$ für alle Belegungen β. Ein entsprechender Beweis wird im folgendem an zwei Beispielen vorgenommen:

1. $\mathsf{true} = (x \vee (\neg x))$

x	$(x \vee (\neg x))$
0	L
L	L

2. $x = x \wedge (x \vee y)$

x	y	$x \vee y$	$x \wedge (x \vee y)$
0	0	0	0
0	L	L	0
L	0	L	L
L	L	L	L
ENV			
$\mathbb{B}^2$		$\mathbb{B}$	$\mathbb{B}$

2.6.6 Anwendung der Gesetze der Booleschen Algebra

Mit den Gesetzen der Booleschen Algebra können wir rein semantisch, also mit Termen, rechnen. Dabei werden Terme durch gleichbedeutende Terme ersetzt.

Beispiel:
$$
\begin{array}{lll}
 & y \wedge \neg y & \\
= & \neg y \wedge y & \text{Kommutativgesetz} \\
= & \neg y \wedge \neg\neg y & \text{Involutionsgesetz} \\
= & \neg(y \vee \neg y) & \text{De Morgan} \\
= & \neg\mathsf{true} & \text{Definition von } \mathsf{true} \\
= & \mathsf{false} & \text{Definition von } \mathsf{false}
\end{array}
$$

Für Gleichungen mit semantisch äquivalenten Termen gelten die klassischen Regeln der **Äquivalenzrelation**. Es gilt für beliebige Terme t_1, t_2, t_3:

1. $t \sim t$ — Reflexivität
2. gilt $t_1 \sim t_2$ so gilt auch $t_2 \sim t_1$ — Symmetrie
3. gilt $t_1 \sim t_2$ und gilt $t_2 \sim t_3$ so gilt auch $t_1 \sim t_3$ — Transivität

Definition der Äquivalenzrelation auf eine Menge M:

Relation: $m \sim m' \quad m, m' \in M \qquad$ (m und m' in Relation)

"="(semantisch äquivalent) ist in Bezug auf Boolesche Terme die Äquivalenzrelation, weil "="auf $\mathbb{B}$ echte Gleichheit ist, also Äquivalenzrelation.

Konsistentes Ersetzen von Identifikatoren auf der linken und auf der rechten Seite von Gleichungssystemen.

Beispiel: $x \wedge y = y \wedge x$
Simultanes Ersetzen von x durch y und y durch $\neg y$
$y \wedge \neg y = \neg y \wedge y$

Die **Substitution** ist eine Operation auf Boolesche Terme. Sie gestattet es, in Termen freie Identifikatoren durch Terme zu ersetzen. Seien t_1, t_2 Boolesche Terme mit freien Identifikatoren aus ID und x ein Identifikator; mit $t_1[t_2/x]$ bezeichnen wir denjenigen Term, den wir aus t_1 erhalten indem wir den Identifikator x an allen Stellen in t_1 durch t_2 ersetzen.

Die Substitution entspricht einer dreistelligen Abbildung (statt dem Funktionssymbol werden eckige Klammern und der Schrägstrich verwendet):

$$
.[./.]: \quad \underbrace{W_{\mathsf{Bool}}(ID)}_{t_1} \times \underbrace{W_{\mathsf{Bool}}(ID)}_{t_2} \times \underbrace{(ID)}_{x} \to \underbrace{W_{\mathsf{Bool}}(ID)}_{t_1[t_2/x]}
$$

welche den gegebenen Booleschen Termen t_1, t_2 und einem gegebenen Identifikator x einen Booleschen Term $t_1[t_2/x]$ zuordnet. Der Werteverlauf der Substitution ist durch folgende Regeln definiert:

x, y beliebige Identifikatoren
t, t_1, t_2 beliebige Boolesche Terme

$$
\begin{array}{rll}
\mathsf{false}[t/x] & =_{\text{def}} \mathsf{false} & \\
\mathsf{true}[t/x] & =_{\text{def}} \mathsf{true} & \\
a[t/x] & =_{\text{def}} a & a = \text{ atomare Aussage} \\
x[t/x] & =_{\text{def}} t & \\
y[t/x] & =_{\text{def}} y & \text{falls } x \text{ und } y \text{ verschieden} \\
(\neg t_1)[t/x] & =_{\text{def}} (\neg t_1[t/x]) & \\
(t_1 \vee t_2)[t/x] & =_{\text{def}} (t_1[t/x] \vee t_2[t/x]) &
\end{array}
$$

Analoge Definitionen ergeben sich für die Terme, die mit den Operatoren $\wedge, \Rightarrow, \Leftrightarrow$ gebildet werden. Man beachte, daß die Substitution keine Abbildung zwischen Booleschen Werten sondern eine Abbildung zwischen Booleschen Termen ist.

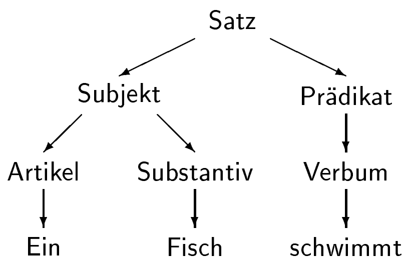
Sind $x_1, ..., x_n$ paarweise verschiedene Identifikatoren, so schreiben wir auch $t[t_1/x_1, ..., t_n/x_n]$ für den Term, der durch die **simultane Substitution** der Identifikatoren $x_1, ..., x_n$ in t durch die Terme $t_1, ..., t_n$ entsteht.

Siehe auch: Manfred Broy: **Informatik Eine grundlegende Einführung** Band 1; 2.Auflage; 1.2 Aussagen als Beispiel für Informationen; Seite 8ff

2.7 Formale Sprachen

2.7.1 Chomsky-Grammatiken

In den 50er Jahren beschäftigten sich N. Chomsky[2]und andere Linguisten mit dem Problem die Struktur von Sätzen in natürlicher Sprache als Ableitungsbaum darzustellen. Hier ein Beispiel für die Zeichenreihe Ein Fisch schwimmt.



N. Chomsky nannte diese Systeme Grammatiken und ihre Regeln Produktionen. In einer Grammatik unterscheidet man syntaktische Begriffe (Satz, Verbum) von den Wörtern der zu beschreibenden Sprache, die man als Einzelzeichen eines Zeichenvorrates Σ betrachtet. Diesen Zeichenvorrat bezeichnet man als **Terminalalphabet**, die in ihm enthaltenen Zeichen als **Terminale** oder als **terminale Zeichen**. Im Gegensatz dazu bezeichnet man den Vorrat der syntaktischen Variablen N als **Nichtterminalalphabet** und analog dazu die in ihm enthaltenen Zeichen als **Nichtterminale** oder **nichtterminale** Zeichen.

Ziel der Angabe einer Grammatik G ist es, die terminalen Zeichenreihen zu beschreiben, die einen ausgezeichneten syntaktischen Begriff S, dem **Startsymbol, Axiom** oder **Ziel** der Grammatik, entsprechen. Als **Vokabular** der Grammatik bzw. der Formalen Sprache bezeichnet man die Vereinigung $V = N \cup \Sigma$. Unter den **Produktionen** P versteht man eine endliche Menge von Regeln $l \to r$ mit Zeichenreihen über V (d.h. mit Zeichen aus V). Das Quadtupel $G = (\Sigma, N, P, S)$ bezeichnet man als **Chomsky-Grammatik** oder Phasenstrukturgrammatik.

2.7.2 Zeichenfolgen

Im folgenden bezeichnen wir mit A das **Alphabet**: eine gegebene endliche Menge von Zeichen für die eine lineare Ordnung definiert ist. Eine beliebige Folge von Zeichen $x_1, ..., x_n \in A$ bildet ein **Wort** der Länge n. Man schreibt $< x_1 ... x_n >$ für dieses Wort. Als Kurzdarstellung für die Länge eines Wortes schreibt man $|x_1 ... x_n| = n$. Allgemein sprechen wir auch von einen $n - Tupel$. A^i bezeichnet die Menge aller $i - Tupel$.

[2]Noam A. Chomsky, geb. 1928, amerikanischer Linguist

Für A^0 vereinbaren wir das **leere Wort** ε. Man beachte, daß A^1 nicht identisch zu der Menge A ist. Vielmehr sollte sorgfältig zwischen dem einelementigen Wort $< a >$ und dem Element a unterschieden werden. Unter A^+ versteht man die Menge aller nicht leeren endlichen Worte über A. Somit gilt:

$$A^+ = A \cup AA \cup AAA \cup ... = \bigcup_{i=1}^{\infty} A = A^i$$

Für A^+ existiert eine Verknüpfungsoperation. Sie wird als Hintereinanderstellen oder Konkatenation bezeichnet. Der zugehörige Operator ist $\circ$. Wenn $w =< a_1...a_n >$ und $v =< b_1...b_m >$ gegeben sind, so ergibt die Konkatenation $w \circ v$ das Wort $wv =< a_1...a_nb_1...b_m >$. Die Summe der Zeichenfolgen w und v ergibt dabei die Länge der Zeichenfolge wv, es gilt:

$$|w| = n \quad \text{und} \quad |v| = m \quad \Rightarrow \quad |wv| = n + m$$

A^* repräsentiert die Menge aller endlichen Worte, daher gilt: $A^* = A^+ \cup \{\varepsilon\}$. Für Konkatenation von Zeichenreihen $u, v, w \in A^*$ gelten die folgenden algebraischen Gesetze:

$$(u \circ v) \circ w = u \circ (v \circ w) \qquad \text{Assoziativität}$$

$$w \circ \varepsilon = w = \varepsilon \circ w \qquad \varepsilon \text{ ist neutrales Element}$$

die Menge A^+ ist eine freie Halbgruppe. Unter einer **Halbgruppe** versteht man eine Menge M mit einer zweistelligen Operation, also einer Operation mit zwei Operanden, die wieder Ergebnisse in M liefert. Für diese Operation muß das Assoziativgesetz gelten. A^* bildet eine Halbgruppe mit dem neutralen Element (Einselement) ε; in der Algebra wird eine solche Struktur **Monoid** genannt.

Sprache wird durch sukzessives Anwenden von Regeln (Produktionen P) $l \to r$, beginnend mit dem Startzeichen S, erzeugt. Dazu existieren die sogenannten **Metaregeln**. Eine Metaregel beschreibt, die Anwendung einer Menge von Produktionen. Man wählt beliebig irgendeine Regel, die anwendbar ist aus, und wendet sie an einer beliebigen Stelle im Wort, an der sie anwendbar ist, an. Bei der Anwendung der Produktion $l \to r$ auf ein Wort w geht man wie folgt vor.

- Prüfe ob das Wort l in w als Teilwort vorkommt oder anders ausgedrückt prüfe ob $w = slt$, dabei stehen s und t für beliebige Worte einschließlich des leeren Wortes ε.

- Wenn diese Bedingung erfüllt ist, dann ist $l \to r$ anwendbar. Dies bedeutet, daß das Vorkommen von l in w durch r ersetzt wird.

- Somit wurde das Wort $w = slt$ in das Wort $w' = srt$ umgewandelt.

Eine Produktion P $u \to v$ bestimmt eine Relation zwischen Worten $\Rightarrow_p : w \Rightarrow_p w'$ gilt genau dann, wenn $w = sut$ und $w' = svt$ Der Begriff der **Nichtterminiertheit** bedeutet, daß man durch die iterierte (wiederholte) Anwendung von Produktionen $\Rightarrow_p^*$ ($* =$ endlich oft einschließlich Null) über Worten letztendlich beliebig lange Worte erhalten kann. Natürlich kann man ein Wort nicht beliebig oft verkürzen, hier kommt es zwangsläufig zu einer Terminierung. Es gilt: $w \Rightarrow_p^* w'$, wenn es Worte $w_1, ..., w_n$ gibt mit:

$$w = w_1, w_i \Rightarrow_p w_{i+1}, w_n = w' \quad (i = 1, ..., n)$$

2.7.3 Die Sprache einer Grammatik

Um eine Grammatik und die daraus resultierende Sprache zu charakterisieren werden im allgemeinen nur die Produktionen angegeben:

$$P: \ l \rightarrow r_1, l \rightarrow r_2, ... : \text{ kurz } l \rightarrow r_1|r_2|...$$

Damit im weiteren Klarheit über die verwendeten Symbole herrscht, will ich den Syntax der wesentlichen Begriffe definieren bzw. wiederholen.

GROSSBUCHSTABEN	Nichtterminale
KLEINBUCHSTABEN	Terminale
S	Startzeichen
$\Rightarrow_{(p)}$	Ein-Schritt-Ableitungsrelation (Anwendung einer Produktion
$\Rightarrow^*_{(p)}$	allgemeine Ableitungsrelation (Produktion wird endlich oft angewendet)

Das Wort v heißt aus w ableitbar wenn gilt $w \Rightarrow^* v$, daß heißt $w = w_1 \Rightarrow w_2 \Rightarrow ... \Rightarrow w_{n-1} \Rightarrow w_n = v$. Ein Wort w, das in endlich vielen Schritten aus S abgeleitet werden kann ($S \Rightarrow^* w$) und das keine Nichtterminale enthält, nennt man eine **Satzform** oder **Phrase**. Unter $L(G)$ versteht man die **Sprache** oder den Sprachschatz einer Grammatik G. Die Sprache läßt sich somit, als Menge aller Worte die nur aus Terminalen bestehen und aus S ableitbar sind verstehen.

$$L(G) = \{w \in \Sigma^* | S \Rightarrow^*_G w\}$$

2.7.4 Produktions-, Grammatik- und Sprachtypen

Typen von Produktionen			
Produktion	**Produktionstyp**	**Eigenschaften**	**Gruppentyp**
$l \rightarrow r$	allgemein	$l, r \in V^*$beliebig	CH - 0
$l \rightarrow \varepsilon$	ε-Produktion	$l, r \in V^*, r = \varepsilon$	
$l \rightarrow r$	beschränkt	$l, r \in V^*, 1 \leq \|l\| \leq \|r\|$	CH - 1
$uAv \rightarrow urv$	kontextsensitiv	$A \in N, u, v, r \in V^*, r \neq \varepsilon$	
$A \rightarrow r$	kontextfrei	$A \in N, r \in V^*$	CH - 2
$A \rightarrow Bx$	linkslinear	$A, B \in N, x \in \Sigma$	CH - 3
$A \rightarrow xB$	rechtslinear		
$A \rightarrow x$	terminierend	$A, B \in N, x \in \Sigma$	

Unterliegt eine Grammatik keinen Einschränkungen so nennt man sie eine Grammatik vom **Chomsky-Typ 0** oder kurz **CH-0-Grammatik**. Beinhaltet eine CH-0-Grammatik zusätzlich die ε-Produktion, so können die Satzformen durch Anwendung von Produktionen verkürzt werden. Ist eine Grammatik hingegen beschränkt oder **kontextsensitiv** so bezeichnet man sie als Grammatik vom **Chomsky-Typ 1** oder kurz **CH-1-Grammatik**. Bei einer solchen Grammatik werden die Satzformen durch die Anwendung von Produktionen verlängert oder sie bleiben gleich lang. Eine Grammatik deren Produktionen **kontextfrei** sind nennt man eine Grammatik vom **Chomsky-Typ 2** oder kurz **CH-2-Grammatik**. Enthält eine solche Grammatik keine ε-Produktionen, so bezeichnet man sie als **ε-frei**. Eine Grammatik wird dann als regulär, als eine Grammatik vom **Chomsky-Typ 3** oder kurz **CH-3-Grammatik**, bezeichnet, wenn sie terminierenden und ε-Produktionen nur links- und rechtslineare Produktionen enthält.

Eine Grammatik heißt vom Typ i, wenn die Typen jede ihrer Produktionen vom Typ $k \geq i$ sind und mindestens eine Produktion vom Typ i ist. Eine Sprache $L \subseteq \Sigma^*$ heißt vom Typ i wenn es eine Grammatik G vom Typ i gibt mit $L = L(G)$. Eine **Sprachklasse** vom Typ i umfaßt alle Sprachen vom Typ i. Um den in diesen Abschnitt gebrachten Stoff verständlicher zu machen möchte ich ein Beispiel einfügen:

$$\left.\begin{array}{r} AB \to Ab, A \to a, B \to b \\ S \to AS, S \to B \end{array}\right\} \text{Grammatik vom Typ CH-1}$$

Die Sprache $L(G)$ dieser Grammatik G ergiebt sich wie folgt:

$$S \Rightarrow_{(S \to AS)} AS \Rightarrow^*_{(S \to AS)} A...AS \Rightarrow_{(S \to B)} A...AB \Rightarrow^*_{(A \to a)} a...aB$$

$$a...aB \Rightarrow_{(B \to b)} a...ab$$

$$S \Rightarrow_{(S \to AS)} AS \Rightarrow^*_{(S \to AS)} A...AS \Rightarrow_{(S \to B)} A...AB$$

$$A...AB \Rightarrow_{(AB \to Ab)} A...Ab \Rightarrow^*_{(A \to a)} a...ab$$

$$S \Rightarrow_{(S \to B)} B \Rightarrow_{(B \to ba)} b$$

Damit läßt sich die Sprache $L(G)$ wie folgt darstellen:

$$L(G) = \{a^n b | n \geq 0\}$$

Für diese Sprache kann man auch eine andere Grammatik angeben:

$$\left.\begin{array}{r} S \to aS \\ S \to b \end{array}\right\} \text{Grammatik vom Typ CH-3 (rechtslinear)}$$

$$S \Rightarrow_{(S \to aS)} aS \Rightarrow^*_{(S \to aS)} a...aS \Rightarrow_{(S \to b)} a...ab$$

Satz: Die Chomsky-Sprachgrammatiken bilden eine echte Hierarchie.

Beweisansatz:

$L_2 = \{a^n b^n | n \geq 1\}$ kontextfrei aber nicht rechtslinear

$L_1 = \{a^n b^n c^n | n \geq 1\}$ CH-1 aber nicht kontextfrei

$L_0 = L_{\text{Halteproblem}}$ CH-0 aber nicht CH-1

Beispiel einer Grammatik für L_1:

$$\left.\begin{array}{r} S \to abc | aAbc, bB \to Bb \\ aB \to aaA, aB \to aa \end{array}\right\} \text{Grammatik vom Typ CH-1}$$

$$S \Rightarrow aAbc \Rightarrow abAc \Rightarrow abBbcc \Rightarrow aBbbcc \Rightarrow aaAbbcc$$

A ist Bote nach rechts
B ist Bote nach links

Sprachen vom Typ 1 sind **entscheidbar** , d.h. wenn eine Grammatik G und ein Wort $w \in \Sigma$ gegeben sind so existiert ein “Programm”für den Computer, das in endlich vielen Schritten feststellt, ob $w \in L(G)$ oder nicht.

Im folgenden sei ein Entscheidungsverfahren für Grammatiken G vom Typ CH-1 vorgestellt mit dem es möglich ist festzustellen, ob ein Wort $w \in \Sigma^*$ Das Verfahren beruht darauf alle Ableitungen $S \Rightarrow^* v$ mit $|v| \leq |w|$ zu bilden. Alle Satzformen u $S^* \Rightarrow^* u \Rightarrow^* v$ sind kürzer als v. D.h. es existieren endlich viele solche Ableitungen. Entweder kommt w am Ende einer solchen Ableitung vor oder es ist nicht in der Sprache enthalten.

Generell existieren zwei Verfahren um zu testen ob ein Wort w Element einer Sprache $L(G)$ ist.

1. **Erzeugungsverfahren** (**Top-Down-Verfahren**): Dieses Verfahren beruht darauf eine Ableitung $S \Rightarrow_G u_1 \Rightarrow_G u_2 \Rightarrow_G ... \Rightarrow_G w$ zu finden. Bei dem Typ CH-1 ist dieses Verfahren immer Stop-Entscheidbar. Bei Typ CH-0 hingegen ist keine Terminationsbedingung vorhanden.

2. **Reduktionsverfahren** (**Bottom-Up-Verfahren**): Dieses Verfahren geht genau entgegengesetzt vor. Es startet mit w und versucht Produktionen rückwärts anzuwenden. D.h. es sucht Produktionen $l \rightarrow r$, so daß v ein Teilwort von $w = urv$ ist. Wenn dies der Fall ist, so wird w zu ulv reduziert. Diese Schritte werden solange wiederholt, bis man das Startzeichen S erhält. Ist dies der Fall, so ist $w \in L(G)$.

2.7.5 Formale Sprachen

Wenn A eine Menge von Zeichen ist, so heißt eine Teilmenge S von A^* **formale Sprache**. Ein Beispiel für formale Sprachen sind die Booleschen Terme. Die Menge der Booleschen Terme mit Identifikatoren aus ID lassen sich als eine Teilmenge von $(ID \cup \{\neg, \vee, \wedge, \Rightarrow, \Leftrightarrow, \underline{\text{true}}, \underline{\text{false}}, (,)\})^*$ darstellen. Die Menge der korrekt gebildeten Booleschen Terme entspricht einer formalen Sprache.

Es existieren verschiedene Methoden, um eine formale Sprache ($S \subseteq A^*$) auszuzeichnen. Für die natürliche Sprachen definieren die Wörterbücher, welche Zeichenfolgen die Wörter bilden und die Grammatiken, nach welchen Regel die Wörter zu Sätzen zusammengefügt werden.

In der Informatik existieren für die formalen Sprachen sogenannte **Notationen**, die beschreiben nach welchen Regeln die Wörter aufgebaut werden. Wenn die formale Sprache innerhalb eines Informationssystems zur Repräsentation von Informationen dient, so bezeichnet man die Regeln für den Aufbau der Wörter als **Syntax** und die Bedeutung der Sprache als **Semantik**. Die Funktion, die jeden Wort der Sprache eine Bedeutung zuordnet heißt Semantikfunktion.

Siehe auch: MANFRED BROY: **Informatik Eine grundlegende Einführung** Band 1; 2.Auflage; 1.3.4 Zeichensequenzen; Seite 26ff und: GERHARD GOOS: **Vorlesungen über Informatik** Band 1; 2.Auflage; 1.6.3 Chomsky-Grammatiken; Seite 33ff

2.8 Berechenbarkeit

2.8.1 Textverarbeitung und Rechnen

In diesen Abschnitt wollen wir uns auf theoretischem Niveau dem Prinzip der Verarbeitung von Zeichen und dem Berechnen von Werten im Rechner annähern. Eine Berechnung wird dabei als Zeichenverarbeitung aufgefaßt. Dazu wollen wir von einem möglichst einfachen Konzept ausgehen das mit Teams von sogenannten Agenten arbeitet. Dabei gehen wir von folgenden Voraussetzungen aus:

- Die Zeichenfolge die bearbeitet wird ist eindimensional, also linear $\rightarrow$ Papierstreifen.

- Der "Papierstreifen" ist in einzelne Bereiche – "Karos" – aufgeteilt. In jedem dieser "Karos" kann nur ein Zeichen stehen.

- Es sind nur die zwei Zeichen '1' und '0' möglich (0 steht für ein leeres "Karo"), d.h. es existieren nur Werte über $\{1, 0\}$.

- Das Rechnen ist vollständig möglich.

- Der Papierstreifen ist nach links und rechts beliebig um leere Karos verlängerbar.
- Die Zahl $n \in \mathbb{N}$ wird durch n Einsen dargestellt.

Bei einer Addition zweier Zahlen, würde die Null, die zwischen den beiden Zahlen steht, auf Eins gesetzt und anschließend die führende Eins in eine Null umgewandelt werden. Das Ganze würde dann in etwa wie folgt aussehen:

$$\boxed{0}\underbrace{\boxed{1}\boxed{1}\boxed{...}\boxed{1}}_{n}\boxed{0}\underbrace{\boxed{1}\boxed{...}\boxed{1}}_{m}\boxed{0} \quad \Rightarrow$$

$$\boxed{0}\boxed{0}\underbrace{\boxed{1}\boxed{...}\boxed{1}\boxed{1}\boxed{1}\boxed{...}\boxed{1}}_{n+m}\boxed{0}$$

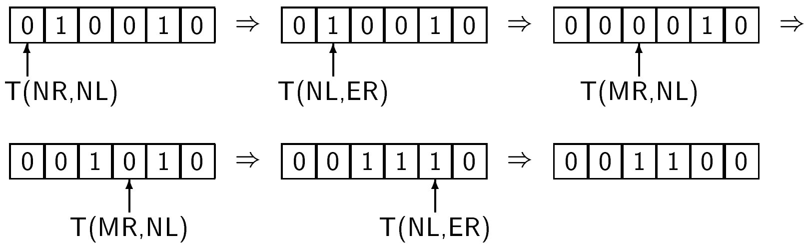
Jeder Akteur kann nur wenige Aktionen ausführen. Ein Akteur kann zunächst feststellen ob ein Karo leer (Null) oder markiert (Eins) ist. Auf der Grundlage dieses Wertes ist der Akteur des weiteren in der Lage abhängig vom eingelesenen Zeichen einen weiteren Akteur (X oder Y) aufzurufen. Dies geschieht nach folgenden Entscheidungsschema:

```
if (karo==0) rufe X;
else rufe Y;
```

Ein Akteur kann ein Karo markieren M(mark), d.h er setzt es von Null auf Eins oder er kann ein Karo löschen E(erase), in diesem Fall setzt er es von Eins auf Null. Dieses Konzept entspricht im wesentlichen dem Konzept der Turingmaschine von 1936. Um die Arbeit der Akteure zu koordinieren müssen weitere Regeln aufgestellt werden.

- Der Beginn liegt immer direkt links neben der ersten Eins. Er wird durch einen Punkt unter dem entsprechenden Karo angezeigt.
- Die erste Aktion eines Akteurs ist M,E oder N. N steht dabei für "tue nichts" (der Karoinhalt wird nicht geändert).
- Ein Akteur kann sich entweder einen Schritt nach links L oder nach rechts R wenden. Wichtig ist hier das er wirklich nur einen Schritt gehen kann.
- Nun kann der Akteur entweder einen weiteren Akteur aufrufen, oder die Bearbeitung der Zeichenfolge durch die banale Stoppaktion S terminieren.

Turings[3] These (Behauptung) ist nun: Mit diesem System von Agenten ist alles, was mit Computern überhaupt berechenbar ist ausführbar bzw. berechenbar. Anhand eines Beispiels möchte ich nun näher auf die Abfolge eines solchen Algorithmus eingehen.



[3] Alan Mathison Turing (1912-1954), britischer Mathematiker

Es zeigt sich, daß eine weitere Strukturierung notwendig ist. Die Akteure müssen benannt werden. Auch sollte die Reihenfolge der einzelnen Aktionsgruppen einheitlich sein. Damit kommen wir zu folgendem Syntax: $\mathsf{T}_i := \mathsf{T}(\mathsf{HBA},\mathsf{H'B'A'})$

H $\mathsf{H'}$	Handlung	M E oder N
B $\mathsf{B'}$	Bewegung	R oder L
A $\mathsf{A'}$	Aufruf	eines neuen Teams T_j oder Stop S

Das obige Beispiel würde nun wie folgt aussehen:

$\mathsf{T}_1 := \mathsf{T}(\mathsf{NRT}_2,\ \mathsf{NLS})$
$\mathsf{T}_2 := \mathsf{T}(\mathsf{NLS},\ \mathsf{ERT}_3)$
$\mathsf{T}_3 := \mathsf{T}(\mathsf{MRT}_4,\ \mathsf{NLS})$
$\mathsf{T}_4 := \mathsf{T}(\mathsf{MRT}_5,\ \mathsf{NLS})$
$\mathsf{T}_5 := \mathsf{T}(\mathsf{NLS},\ \mathsf{ERS})$

Ein weiteres Beispiel wäre das Team für die Nachfolgerberechnung: $n \to n+1$, das eine Zeichenfolge folgendermaßen umformt:

$$\boxed{0}\boxed{0}\boxed{1}\boxed{\dots}\boxed{1}\boxed{0} \quad\Rightarrow\quad \boxed{0}\boxed{1}\boxed{1}\boxed{\dots}\boxed{1}\boxed{0}$$

$\mathsf{T}_1 := \mathsf{T}(\mathsf{MLS},\ \mathsf{NRS})$

Ein denkbare Lösung für unser Beispiel am Anfang dieses Abschnittes, der Addition $(n,m) \to n+m, n \neq 0 \neq m)$, wäre:

$$\boxed{0}\underbrace{\boxed{1}\boxed{1}\boxed{\dots}\boxed{1}}_{n}\boxed{0}\underbrace{\boxed{1}\boxed{\dots}\boxed{1}}_{m}\boxed{0} \quad\to$$

$$\boxed{0}\boxed{0}\underbrace{\boxed{1}\boxed{\dots}\boxed{1}\boxed{1}\boxed{1}\boxed{\dots}\boxed{1}}_{n+m}\boxed{0}$$

$\mathsf{T}_1 := \mathsf{T}(\mathsf{NRT}_1,\ \mathsf{ERT}_2)$
$\mathsf{T}_2 := \mathsf{T}(\mathsf{MLT}_3,\ \mathsf{NRT}_2)$
$\mathsf{T}_3 := \mathsf{T}(\mathsf{NRT}_4,\ \mathsf{NLT}_3)$
$\mathsf{T}_4 := \mathsf{T}(\mathsf{NRS},\ \mathsf{NLS})$

Eine Funktion $f : \mathbb{N} \times \dots \times \mathbb{N} \to \mathbb{N}$ heißt T berechenbar, wenn es ein endliches Team $\mathsf{T}_1, \dots, \mathsf{T}_n$ gibt, so daß bei Start mit T_1 auf dem ersten leeren Karo ganz links (mit $\bullet$ bezeichnet) , das Team nach endlich vielen Schritten stoppt (bei $\bullet$) und auf dem Streifen $f(n_1, n_2, \dots, n_k)$ steht. Das heißt, ist der Startwert der Funktion

$$\underset{\bullet}{\boxed{0}}\underbrace{\boxed{1}\boxed{\dots}\boxed{1}}_{n_1}\boxed{0}\underbrace{\boxed{1}\boxed{\dots}\boxed{1}}_{n_2}\boxed{0}\boxed{1}\boxed{\dots}\boxed{1}\boxed{0}\underbrace{\boxed{1}\boxed{\dots}\boxed{1}}_{n_k}\boxed{0}$$

so steht nach der Anwendung der Funktion für jedes k-Tupel:

$$\boxed{0}\underbrace{\boxed{1}\boxed{\dots}\boxed{1}}_{f(n_1,n_2,\dots,n_k)}\boxed{0}$$

Es existieren mathematisch klar definierte Funktionen über $\mathbb{N}$, die nicht berechenbar sind.

2.8.2 Das Busy - Beaver - Spiel (T. Rado, 1960)

Wettbewerb der Kategorie n: Zu $n \in \mathbb{N}$ ist ein Team T_1 ... T_n Sieger, wenn es eine maximale Zahl von Einsen auf einen zunächst leeren Streifen schreibt und stoppt (maximal: mindestens so viele wie jedes andere Team).

Der Sieger für den Fall $n = 3$ ist:

$\mathsf{T}_1 := \mathsf{T}(\mathsf{MRT}_2, \mathsf{NLT}_3)$
$\mathsf{T}_2 := \mathsf{T}(\mathsf{MRT}_3, \mathsf{NRS})$
$\mathsf{T}_3 := \mathsf{T}(\mathsf{MLT}_1, \mathsf{ELT}_2)$

Der Ablauf dieses Algorithmus ist im folgenden kurz abgedruckt.

0	0	0	0	1	1	1	0
				1	2	3	

0	0	0	1	0	1	1	0
			2	3	1		

0	1	1	1	1	1	1	0
	1	3	1	3			

0	1	1	1	1	1	1	0
		2	S				

Der Wettbewerb definiert eine Funktion R: $\mathbb{N} \to \mathbb{N}$: $\mathsf{R}_{(n)}$ = Zahl der Einsen , die der Sieger der Kategorie n auf einen leeren Streifen Schreibt.

0	1	2	3	4	5	6
0	1	4	6	13	?	?

Für 5 und 6 sind noch keine Lösungen bekannt. Die bisher erzielten Maximalwerte sind:

$\mathsf{R}_{(5)}$=4096
$\mathsf{R}_{(6)}$=95524079

Wie man aus diesen Zahlenwerten unschwer erkennen kann, ist die Funktion $\mathsf{R}_{(n)}$ streng monoton wachsend ($\mathsf{R}_{(n)}$ >$\mathsf{R}_{(n)}$ für $n > 0$). Feststellung: $\mathsf{R}_{(n)}$ ist eine klar definierte mathematische Funktion, mit allgemeiner Vorschrift zur Bestimmung von $\mathsf{R}_{(n)}$ für jedes n.

1. Stelle alle Teams von n Akteuren auf, daß sind $((3 * 2) * (n + 1))^{2n}$ Teams.
2. Starte jedes Team mit leeren Streifen.
3. Zähle die Einsen jeweils bei Stop.
4. Bestimme das Maximum.

Theorem: $\mathsf{R}_{(n)}$ ist nicht T-berechenbar und damit überhaupt nicht berechenbar. Es gibt kein Programm, das $\mathsf{R}_{\ (n)}$ für jedes n berechnet.
Vorüberlegungen zum Beweis: Überlegen wie Teams aus weiteren Teams zusammenzubauen sind.

Hintereinanderschaltung:

$$\begin{array}{lllll}
\text{Teams} & \mathsf{G}: & \mathsf{T}_1 & \text{und} \quad \mathsf{F}: & \mathsf{T}_1 \\
 & & \mathsf{T}_2 & & \mathsf{T}_2 \\
 & & \mathsf{T}_3 & & \mathsf{T}_3 \\
 & & \vdots & & \vdots \\
 & & \mathsf{T}_n & & \mathsf{T}_m
\end{array}$$

$$\begin{array}{lll}
\text{bilde Team} & \mathsf{G} \circ \mathsf{F}: & \left.\begin{array}{l} \mathsf{T}_1 \\ \mathsf{T}_2 \\ \mathsf{T}_3 \\ \vdots \\ \mathsf{T}_n \end{array}\right\} \\
 & & \left.\begin{array}{l} \mathsf{T}_{n+1} \\ \mathsf{T}_{n+2} \\ \mathsf{T}_{n+3} \\ \vdots \\ \mathsf{T}_{n+m} \end{array}\right\}
\end{array}$$

Der Aufruf eines Akteurs, bleibt unverändert wie bei G aber S muß stets durch T_{n+1} ersetzt werden.

Bei jedem Akteur T_i :=(... T_j, ... T_k)müssen i, j, k durch $i+n, j+n, k+n$ ersetzt werden.

Spezielle Teams für den Beweis:

$$\begin{array}{ll}
\mathsf{B}_4 & \mathsf{T}_1 := \mathsf{T}(\mathsf{MLT}_2,\ \mathsf{ELT}_3) \\
 & \mathsf{T}_2 := \mathsf{T}(\mathsf{MRT}_1,\ \mathsf{NLT}_1) \\
 & \mathsf{T}_3 := \mathsf{T}(\mathsf{NLS},\ \mathsf{NLT}_4) \\
 & \mathsf{T}_4 := \mathsf{T}(\mathsf{MRT}_4,\ \mathsf{ERT}_2)
\end{array}$$

B_4 ist eine Variante eines Siegers der Kategorie 4 dort nur $\mathsf{T}_3 := \mathsf{T}(\mathsf{MLS,NLT}_4)$. Sie schreibt 12 Einsen auf einen leeren Streifen und stoppt 3 Karos links davon. Schaltet man Teams der Art B_4 hintereinander, $\mathsf{B}_4^m = \mathsf{B}_4 \circ ... \circ\ \mathsf{B}_4$, so schreibt diese Kombination $12 * m$ Einsen hintereinander. $\mathsf{B}_4^{'}$ entstehe aus B_4 durch ändern von T_3 zu $\mathsf{T}_3 := \mathsf{T}(\mathsf{NRS,NLT}_4)$. Das entstehende Team verhält sich wie B_4, bleibt aber direkt neben den Einsen stehen. Für $n \in \mathbb{N}$ ist $\mathsf{D}_{(n)} = \mathsf{B}_4^{(n-1)} \circ\ \mathsf{B}_4^{'}$ mit $4n$ Akteuren, das auf einen leeren Streifen $12n$ konsekutive[4] Einsen schreibt und dann direkt links davon stoppt.

Nun ergänzt man das Team T_1 ... T_n, das $\mathsf{R}_{(n)}$ Einsen erzeugt (Sieger), um den Akteur $\mathsf{T}_{n+1} := \mathsf{T}(\mathsf{MRS},\ \mathsf{NRT}_{n+1})$ und ersetze in T_1 ... T_n alle S durch T_{n+1}. Durch das Anfügen dieses Akteurs wird eine weitere Eins, rechts von den anderen Einsen, geschrieben.

T_1 ... $\mathsf{T}_{(n+1)}$ schreibt $\mathrm{R}_{(n)} + 1$ Einsen
Also gilt $\mathsf{R}_{(n+1)} \geq \mathsf{R}_{(n)}$

Indirekter Beweis: (Gegenteil annehmen und dann Widerspruch herleiten)

Annahme: Es existiert ein Team TR mit k Akteuren, das $\mathsf{R}_{(n)}$ für jedes n berechnet.

TR liefert zu: $\begin{array}{|c|c|c|c|c|}\hline 0 & 1 & ... & 0 & 0 \\ \hline\end{array}$ stets $\begin{array}{|c|c|c|c|c|}\hline 0 & 1 & ... & 1 & 0 \\ \hline\end{array}$

Beachte: $\mathsf{D}\ _{(n)} \circ\ \mathsf{TR}$ ist ein Team aus $n + k$ Akteuren und schreibt auf einen leeren Streifen erst $12n$ Einsen und dann macht TR daraus $\mathsf{R}_{(12n)}$ Einsen. $\mathsf{D}\ _{(n)} \circ$

[4] unmittelbar aufeinanderfolgend

TR ist sicher nicht besser als der Sieger der Kategorie $4n + k$ des Busy-Beaver Wettbewerbes. Damit gilt:

$$\mathsf{R}_{(4n+k)} \geq \mathsf{R}_{(12n)}$$

Nun benutzen wir die Monotonie von R. Da k eine feste natürliche Zahl ist existiert ein $n_0 \in \mathbb{N}$ mit $12n_0 > 4n_0 + k$ (wenn $n_0 \geq \frac{k}{8}$). Somit gilt:

$$\mathsf{R}_{(12n_0)} \geq \mathsf{R}_{(4n_0+k)}$$

Damit ergibt sich folgender **Widerspruch** für $n = n_0$:

$$\mathsf{R}_{(4n_0+k)} \geq \mathsf{R}_{(12n_0)} > \mathsf{R}_{(4n_0+k)}$$

Nun stellt sich die Frage warum R nicht T - berechenbar ist, obwohl der verwendete Berechenbarkeitsbegriff der allgemeinste ist (Das ist nicht bewiesen!) und die Busy-Beaver Funktion mathematisch klar definiert ist. Die Antwort auf diese Frage fällt nicht schwer. Die Busy-Beaver Funktion läßt sich nicht berechnen, weil man mit keinem allgemeingültigen Verfahren feststellen kann ob ein Team je stoppt. Dies wird als Halteproblem bezeichnet. Das **Halteproblem** (egal ob für die Turingmaschine oder für Programme in einer universellen Programmiersprache) ist nicht entscheidbar.

2.9 Algorithmen

2.9.1 Einführung

Erklärung: Ein Algorithmus ist ein durchführbares Verfahren, das auf präzis definierten Einzelaktionen (Schritten) beruht und dessen Ablauf im Detail exakt beschrieben ist. Die Einzelaktionen müssen effektiv ausführbar sein (mit endlich vielen Hilfsmitteln in endlicher Zeit). Die Beschreibung eines Algorithmus muß endlich lang und eindeutig sein.

Zusammenfassend lassen sich die folgenden Aussagen über Algorithmen treffen:

- ein Algorithmus ist ein schrittweises effektiv ausführbares Verfahren
- er verfügt über eine endliche eindeutige Beschreibung
- die Anwendung erfolgt ausschließlich auf zulässige Daten
- es werden ausschließlich effektive Operationen ausgeführt (im Sinne von tatsächlich ausführbar)
- das Ergebnis liegt nach endlich vielen Schritten vor

Bei der Formulierung von Algorithmen ergeben sich die folgenden Probleme, deren korrekte Umsetzung jeweils Nachzuweisen ist:

1. Der Algorithmus muß für zulässige Eingaben terminieren. Es existiert aber kein allgemeingültiges Verfahren das feststellt ob der Algorithmus terminiert, daher ist ein individueller Beweis nötig.
2. Der Algorithmus muß an sich korrekt sein.

Beispiele für Algorithmen:

a) Die Teams von Akteuren sind ein Möglichkeit zur Beschreibungen für Algorithmen.

b) Ein weiteres Beispiel ist der Euklidische[5] Algorithmus zur Berechnung des größten gemeinsamen Teilers natürlicher Zahlen.
Die Aufgabenstellung für diesen Algorithmus läßt sich wie folgt formulieren: Gegeben seien zwei natürliche Zahlen $a, b \in \mathbb{N}$ mit $a \neq 0 \neq b$; gesucht ist ein maximales t das a und b teilt. Der Algorithmus größter gemeinsamer Teiler zur Berechnung der natürlichen Zahl $t = \mathsf{ggT}(a, b)$ nach Euklid lautet dann wie folgt:

   1) Falls $a = b$ dann $\mathsf{ggT}(a, b) = a$
   2) Falls $a < b$ dann $\mathsf{ggT}(a, a - b)$
   3) Falls $a > b$ dann $\mathsf{ggT}(b - a, b)$

   Die Korrektheit der ersten Regel ist evident[6]. Die Regeln zwei und drei beruhen auf den folgenden mathematischen Gesetzen:

$$\mathsf{ggT}(a, b) = \begin{cases} \mathsf{ggT}(a, b - a), \text{ falls } b > a \\ \mathsf{ggT}(a - b, b), \text{ falls } a > b \end{cases}$$

c) Sortieren von Karteikarten: enthält der Stapel zwei aufeinanderfolgende Karten in falscher Reihenfolge, so vertausche diese, andernfalls ist der Stapel sortiert.

Rekursion (Selbstbezogenheit): Rekursive Algorithmen zerlegen ein gegebenes Problem (durch Selbstaufruf) in immer kleinere Teilprobleme, die schließlich sehr einfach lösbar sind. Diese Lösungen ergeben in richtiger Reihenfolge zusammengesetzt schließlich die Lösung des Gesamtproblems. Wichtig wie bei allen Algorithmen ist auch hier die Terminierung. Sie ist aber meist schon durch die Struktur der Rekursion gegeben, da die Argumente des Algorithmus bei jedem Aufruf kleiner werden und meist aus dem Bereich der natürlichen Zahlen kommen.

Bemerkung: Auch nichtdeterministische Beschreibungen sind erlaubt, wenn die Ergebnisse stets determiniert sind. (Beweis nötig)

Als Beispiel dient uns ein Algorithmus für das Sortieren eines Kartenstapels x (Eingabe).

1. Bilde den neuen sortierten Stapel y auf folgende Weise: Per Definition sei y zu Beginn leer.
2. Ist x nicht leer, so wähle eine beliebige Karte aus x und füge sie an der "richtigen" Stelle ein. Danach wende den Schritt wieder auf x und y an.
3. Wenn x leer ist, dann ist y der sortierte Stapel.

Der Nachweis der Terminierung und der Determiniertheit fällt bei diesem einfachen Algorithmus nicht schwer. Die Terminierung ergibt sich aus der Tatsache das $x \in \mathbb{N}$ bei jedem Schritt kleiner wird und der Algorithmus bei $x = 0$ beendet wird. Die Determiniertheit ergibt sich daraus, daß das Ergebnis nicht von der Wahl der Karte abhängt, da diese in jedem Fall an der richtigen Stelle in y eingefügt wird.

Im folgenden sind eine Reihe von Merkmalen zur Klassifizierung von Algorithmen angegeben. Ein Algorithmus heißt für eine Eingabe:

- **terminierend**, wenn er stets (für alle zulässigen Schrittfolgen) nach endlich vielen Schritten endet.

[5] Euklid, 4./3. Jahrhundert vor Christus, griechischer Mathematiker
[6] offenkundig, klar ersichtlich

- **deterministisch**, wenn in der Auswahl der Verarbeitungsschritte keine Freiheit besteht.
- **determiniert**, wenn das Resultat des Algorithmus eindeutig bestimmt ist.
- **sequentiell**, wenn die Verarbeitungsschritte hintereinander ausgeführt werden.

2.9.2 Textersetzungsalgorithmen

Will man zu einer präzisen Algorithmenbeschreibung, also zu einer maschinell bearbeitbaren, verarbeitbaren und ausführbaren Beschreibung eines Algorithmus übergehen, so ist eine formale Sprache zur Algorithmenbeschreibung und eine exakte Begriffsbildung die Effektivität der elementaren Verarbeitungsschritte nötig. Im einfachsten Fall verwenden Algorithmen Wörter über einer vorgegebenen Zeichenmenge als Ein- und Ausgabe. Da sich durch Wörter Informationen in vielfältiger Weise repräsentieren lassen, kann man Algorithmen stets auf diese Weise darstellen.
Wir definieren V als einen Zeichenvorrat; eine Ersetzungsregel über V ist dann durch das Paar $(v, w) \in V^* \times V^*$ gegeben. Man schreibt:

$$v \to w$$

Als **Textersetzungssystem** über V bezeichnen wir die endliche Menge R von Ersetzungsregeln. Die Anwendung der Regeln erfolgt wie bei den Chomsky - Grammatiken, mit dem Unterschied, daß kein Startzeichen S existiert. Ein solches Textersetzungssystem entspricht also einer Grammatik vom Typ CH-0 ohne Startsymbol. Man kann im weiteren zwischen terminierenden und nichtterminierenden Textersetzungssystemen unterscheiden. Also:

$$\text{Textersetzungssystem} = (V, R) \text{ für } R \subseteq V^* \times V^* \text{ und } V = \text{Vokabular}$$

Bemerkung: der Ablauf eines Textersetzungsalgorithmus ist im allgemeinen nicht deterministisch, trotzdem kann das Ergebnis determiniert sein.

Beispiele:
Die Addition natürlicher Zahlen:

$$V_a = \{|, >, <, +\}, \qquad R: \quad >+< \quad \to \quad \varepsilon$$

$$\text{zulässige Eingabe:} \quad < \underbrace{|\dots \quad \dots|}_{n} >+< \underbrace{|\dots \quad \dots|}_{m} > \quad \Rightarrow \quad < \underbrace{|\dots \quad \dots|}_{n+m} >$$

2.9.3 Markovalgorithmen

Oft werden deterministische Textersetzungsalgorithmen bevorzugt betrachtet. Sie besitzen für jedes Eingabewort genau eine Berechnung und erzeugen somit im Fall der Terminierung genau eine Ausgabe. Ein Beispiel für solche deterministische Algorithmen sind die sogenannten Markovalgorithmen[7]. Bei diesen Algorithmen sind die Regeln linear geordnet. Diese Ordnung wird durch die Numerierung der Regeln erreicht:

$$i: \quad v \to w \qquad i = 1, 2, ..., n$$

Optional kann man gewisse Regeln als haltend kennzeichnen. Dies geschieht mit Hilfe eines Punktes hinter dem Pfeil:

$$i: \quad v \to_{\bullet} w$$

[7] benannt nach dem russischen Mathematiker A.A.MARKOV, geb. 1903

Im weiteren verfügen die Markovalgorithmen über **Metaregeln**, die angeben in welcher Weise die einzelnen Produktionen anzuwenden sind. Die Metaregeln für ein gegebenes Wort $u \in V$ sind:

I Sind mehrere Regeln auf u anwendbar, so nehme man von diesen Regeln die mit der kleinsten Nummer.

II Ist die Regel an mehreren Stellen in u anwendbar, so ist die linkeste Stelle in u zu wählen.

III Ist die Regel als haltend gekennzeichnet, so terminiert der Markovalgorithmus.

Das Ergebnis des Markovalgorithmus, ist das bei der Terminierung vorliegende Wort.

Ein Beispiel für einen Markovalgorithmus:

$$
\begin{array}{rlcl@{\qquad}rlcl}
1: & aaY & \rightarrow & aYa & 6: & Xb & \rightarrow & bYbX \\
2: & abY & \rightarrow & bYa & 7: & Y & \rightarrow & Z \\
3: & baY & \rightarrow & aYb & 8: & Z & \rightarrow & \varepsilon \\
4: & bbY & \rightarrow & bYb & 9: & X & \rightarrow_{\bullet} & \varepsilon \\
5: & Xa & \rightarrow & aYaX & 10: & \varepsilon & \rightarrow & X
\end{array}
$$

Für die Eingabe aba läuft der Algorithmus wie folgt ab:

$$
\begin{array}{clclclc}
aba & \Rightarrow_{10} & Xaba & \Rightarrow_{5} & aYaXba & \Rightarrow_{6} & \\
aYabYbXa & \Rightarrow_{2} & aYbYabXa & \Rightarrow_{5} & aYbYabaYaX & \Rightarrow_{3} & \\
aYbYaaYbaX & \Rightarrow_{1} & aYbaYbaX & \Rightarrow^{*}_{7+8} & abaabaX & \Rightarrow_{\bullet 9} & abaaba
\end{array}
$$

Wie unschwer zu erkennen ist, verdoppelt dieser Algorithmus alle Wörter u mit $u \in \{a,b\}^*$. Die Wortverdopplung ist mit Markovalgorithmen nicht ohne Hilfszeichen (X) möglich.

Bemerkung: jedes Team von Akteuren $(\mathsf{T}_1, \ldots, \mathsf{T}_n)$ läßt sich durch einen Markovalgorithmus simulieren. Der Zeichenvorrat V eines solchen Markovalgorithmus wäre $V = \{0, 1, \mathsf{T}_1, \ldots, \mathsf{T}_n\}$. Der Akteur T_1 mit dem der Algorithmus startet

0	1	...	1	0
•				
T_1				

wird durch $\mathsf{T}_1 01$ repräsentiert. Die allgemeine Situation

..	..	x	y	z	..	..
			T_i			

wird durch $\mathsf{x T_1 yz}$ dargestellt. Die Stoppregel ist mit $\mathsf{S} \rightarrow {}_{\bullet}\varepsilon$ definiert.

Für $i = 1, ..., n$ sind die T_i in den Regeln zu ersetzen. So müßte zum Beispiel T_i = (MRX,ELY) ersetzt werden durch:

$$
\begin{array}{lcl}
\left.\begin{array}{lcl}
\mathsf{T}_i 00 & \rightarrow & \mathsf{1X0} \\
\mathsf{T}_i 01 & \rightarrow & \mathsf{1X0} \\
\mathsf{T}_i 0 & \rightarrow & \mathsf{1X0}
\end{array}\right\} & \begin{array}{l} \mathsf{MRX} \\ \text{wenn rechts neues Karo nötig} \end{array} \\
\left.\begin{array}{lcl}
\mathsf{0T}_i 1 & \rightarrow & \mathsf{Y00} \\
\mathsf{1T}_i 1 & \rightarrow & \mathsf{Y10} \\
\mathsf{T}_i 1 & \rightarrow & \mathsf{Y00}
\end{array}\right\} & \begin{array}{l} \mathsf{ELY} \\ \text{wenn links neues Karo nötig} \end{array}
\end{array}
$$

Allgemein kann gesagt werden, das jede T-berechenbare Funktion auch durch einen Markovalgorithmus berechnet werden kann und umgekehrt.

Jeder Markovalgorithmus A definiert eine Abbildung

$$f_A : V^* \to V^*$$

die durch die folgenden beiden Regeln definiert ist:

1. $f_A(t) = r$ wenn das Wort r das Ergebnis der Anwendung des Markovalgorithmus A auf t ist.

2. $f_A(t) = \perp$ wenn Anwendung des Markovalgorithmus A auf t nicht terminiert. $\perp$ ist ein besonderes Zeichen ("**Bottom**"), welches das fehlende Resultat einer nichtterminierenden Berechnung symbolisiert.

Diese Definition gilt für alle determinierten Textersetzungssysteme.

Eine Teilmenge $L \subseteq \Sigma^*$ heißt **entscheidbar**, wenn ein Markovalgorithmus E existiert, der für jedes $\mu \in \Sigma^*$ terminiert und 1 liefert falls $\mu \in L$ und sonst 0. Das heißt f_E ist die charakteristische Funktion.

$$L \subseteq \Sigma^* : f_E(\mu) = \begin{cases} 1 \text{ falls } \mu \in L \\ 0 \text{ falls } \mu \notin L \end{cases}$$

Ein **semi-entscheidbarer** Markovalgorithmus S_e liegt vor, wenn S_e für jedes $u \in L$ mit 1 terminiert aber für $u \in \Sigma^* \backslash L, \quad (u \notin L)$ nicht. Das heißt L ist der Definitionsbereich der von S_e indizierten Abbildung $f_{S_e} : \Sigma^* \to \{0, 1\}$. Die Aussage ein Algorithmus ist semi-entscheidbar entspricht dabei der Aussage er sei abzählbar.

Theorem: Die Menge der Sprachen $L \subseteq \Sigma^*$ vom CH-Typ 0 (der mit Chomsky - Grammatiken definierten Sprachen) ist gleich der Menge der semi-entscheidbaren Teilmengen $L \subseteq L^*$.

- Jede kontextsensitive Sprache ist entscheidbar.
- Nicht jede Teilmenge $L \subseteq \Sigma^*, \quad (|\Sigma| \geq 2)$ ist semi-entscheidbar (die meisten sind es nicht).
- es gibt semi-entscheidbare nicht entscheidbare $L \subseteq \Sigma^*, \quad (|\Sigma| \geq 2)$ (z.B. die Menge der Teams von Akteuren, die halten).

Siehe auch: Manfred Broy: **Informatik Eine grundlegende Einführung** Band 1; 2.Auflage; 2 Rechenstrukturen und Algorithmen; Seite 31ff und: Gerhard Goos: **Vorlesungen über Informatik** Band 1; 2.Auflage; 1.6.1 Markov-Algorithmen; Seite 31ff

Kapitel 3

Programmiersprachen

3.1 Aufgaben einer Programmiersprache

Programmiersprachen haben die Aufgabe eine standardisierte Notation für Algorithmen zur Verfügung zu stellen, welche die folgenden Eigenschaften erfüllt:

- Die Programmiersprache muß für den Menschen lesbar sein.
- Sie muß für den Computer ausführbar sein.
- Für umfangreiche Algorithmen, sollte sie mit guten Strukturierungsmittel ausgestattet sein.
- Es muß eine exakte Beschreibung der Bedeutung der Sprache vorhanden sein.
- Sie muß automatisch in Maschinensprache des Computers übersetzt werden können.
- Es müssen Sprachmittel zur Beschreibung von Daten und Operationen zur Verfügung gestellt werden.

Es ist möglich die Programmiersprachen in verschiedene Gruppen einzuteilen. Die wesentlichen sind im folgenden kurz aufgeführt.

1. **Funktionale Programmiersprachen**(Applikative Sprachen):
   - z.B. Gofer, LISP, LOGO
   - verfügen über mathematische Notation zur Berechnung mittels Funktionen
   - wesentliche Sprachmittel sind:
     - Funktionsdefinition und -anwendung (Funktionsapplikation)
     - Bedingte Ausdrücke (`if ... then ... else`)
2. **Imperative Programmiersprachen**(Zuweisungsorientierte Sprachen):
   - z.B. Pascal, C
   - Zuweisung von Daten an den Speicher, meist indirekt über Bezeicher (`x = 7;`)
   - wesentliche Sprachmittel sind:
     - Anweisungen für komplizierte Berechnungen von Werten
     - Fallunterscheidungen, Schleifen

3. **Logische Funktionale Programmiersprachen**(deklarierte Sprachen):
   - z.B. Prolog
   - wesentliche Sprachmittel sind:
     – Bedingungen für die Lösung der Aufgabe
     – Bedingungen für die Art des Ergebnisses
4. **Objektorientierte Programmiersprachen**:
   - z.B. Java, C++
   - Weiterentwicklung der imperativen Programmiersprachen
   - zusätzliche Sprachmittel sind:
     – stark erweiterte Strukturierungsmöglichkeiten
5. **Maschinennahe Programmiersprachen / Assembler**:

Es bestehen aber auch andere Unterscheidungsmöglichkeiten. So kann man zwischen sequentiellen (Ausführen von Befehlen hintereinander) und parallelen Programmiersprachen (Ausführung von Schritten parallel auf mehreren Rechnern) unterscheiden.
Eine Programmiersprache wird durch ihren Syntax (Notation) und ihre Semantik (Bedeutung) gekennzeichnet.

3.2 Syntaxbeschreibung

Die Syntaxbeschreibung einer Programmiersprache erfolgt zumeist mittels der sogenannten Backus-Nauer-Form[1], abgekürzt BNF. Das ist eine spezielle Notation für kontextfreie Grammatiken. Zusätzlich werden Kontextbedingungen vereinbart (formal mit Logik).

3.2.1 Reguläre Ausdrücke

Sei Σ ein Alphabet von Terminalzeichen. So definieren reguläre Ausdrucke über Σ eine formale Sprache $L \subseteq \Sigma^*$ vom CH-Typ 3 (reguläre Sprache).
Satz: Zu jeder CH-Typ-3-Sprache L (reguläre Sprache) existiert ein regulärer Ausdruck, der L beschreibt und umgekehrt.

Induktive Definition regulärer Ausdrücke:

1. Terminale Zeichen: Für jedes $a \in \mathbb{Z}$ ist a ein regulärer Ausdruck über Σ mit der Sprache $L(a) = a$.
2. Vereinigung: Sind R und Q reguläre Ausdrücke über Σ mit den Sprachen $L(R)$ und $L(Q)$,so ist $R|Q$ (gesprochen "R oder Q")ein regulärer Ausdruck mit $L(R|Q) = L(R) \cup L(Q)$
3. Konkatenation: Sind R und Q reguläre Ausdrücke über Σ mit den Sprachen $L(R)$ und $L(Q)$,so ist RQ (gesprochen "R gefolgt von Q") ein regulärer Ausdruck mit $L(R|Q) = L(R)L(Q) = \{uv|u \in L(R), v \in L(Q)\}$.
4. Ist R ein regulärer Ausdruck über Σ mit der Sprache $L(R)$, so ist:

[1] von John Backus 1959 erfunden und von Peter Nauer erstmals zur Beschreibung von Algol 60 eingesetzt

Optionen :
$\{R\}$ regulärer Ausdruck für : $L(R) \cup \{\varepsilon\}$

Iteration :
$\{R\}^+$ regulärer Ausdruck für : $\bigcup_{i=1}^{\infty} (L(R))^i = \{u_1, ..., u_n\}, u_i \in L(R), n \geq 1$

Iteration :
$\{R\}^*$ regulärer Ausdruck für : $\{R\}^+ \cup \varepsilon$

leere Sprache :
$\{\}$ regulärer Ausdruck für : $\emptyset$

Ein Beispiel:

Grammatik:

$$\begin{array}{lcl} S & \rightarrow & aS_1|S_2 \\ S_1 & \rightarrow & aS_1|b \\ S_2 & \rightarrow & bS_2|a \end{array}$$

regulärer Ausdruck: $\{a\}^+|\{b\}^*a$

Um formale Sprachen strukturierter zu beschreiben, erlaubt die BNF-Notation die Einführung von Bezeichnern. Ein solcher Bezeichner e wird zur besseren Lesbarkeit durch spitze Klammern '<''>'begrenzt. <e> ist dann eine sogenannte **syntaktische Einheit**. Der Ausdruck

$< \mathsf{e} > ::= R$

weist der syntaktische Einheit <e> die Bedeutung des regulären Ausdrucks R über Σ zu. Dies wird durch das folgende Beispiel verdeutlicht:

$<\mathsf{dyck}> ::= <\mathsf{dyck}><\mathsf{dyck}>|\{a<\mathsf{dyck}>b\}, \qquad \Sigma = \{a, b\}$

DYCK²-Sprache: wohlgeformte Klammer Ausdrücke

Natürlich, kann man dieses Beispiel auch mit einem Textersetzungsalgorithmus realisieren. Es würde dann wie folgt aussehen:

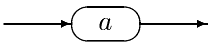
$\mathsf{S} \rightarrow \mathsf{SS}|a\mathsf{S}b|\varepsilon$

Wie in dem Beispiel erkennbar werden auch rekursive Definitionen zugelassen. Also <e>::= R, wobei in R selbst wieder ein <e> auftreten darf.

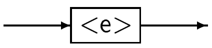
Um BNF-Sprachen zu lesen muß man diese also wie induktive Definitionen interpretieren. Man beginnt mit den Regeln, die nicht rekursiv sind und setzt dann iterativ die anderen Regeln ein.

3.2.2 Grafische Notation für die BNF-Darstellung

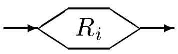
Syntaxdiagramme sind gerichtete Graphen, also ein Netz aus Knoten und gerichteten Verbindungen. Sie stellen eine Sonderform der Beschreibung von BNF-Sprachen dar und sind stellenweise übersichtlicher als die textuelle Darstellung.



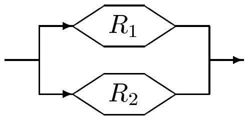
Das Terminalzeichen a



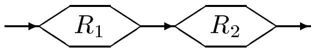
Die syntaktische Einheit <e>

Seien $R_i, i = 1, 2$ durch 

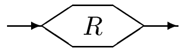
dargestellt, dann steht:

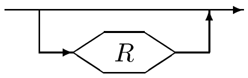


für $R_1|R_2$ und

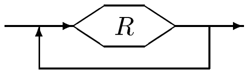


für R_1R_2

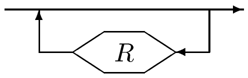
Sei weiterhin R durch  dargestellt, dann steht:



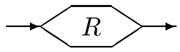
für $\{R\}$,



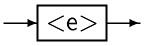
für $\{R\}^+$ und



für $\{R\}^*$

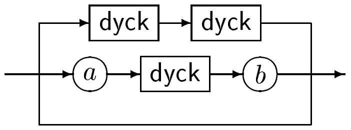
e : 

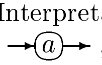
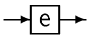
e ist Bezeichner des Graphen R, (<e>::= R). Ein solcher Graph wird als Syntaxdiagramm bezeichnet.



Ist unabhängig vom Syntax in dem es steht als Aufruf des Syntaxdiagramms für e aufzufassen.

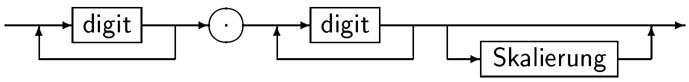
Ein Beispiel für ein solches Syntaxdiagramm liefert uns die Funktion dyck:



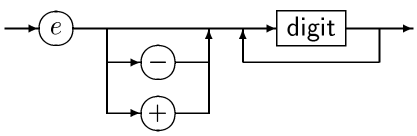
Bei der Interpretation von Syntaxdiagrammen geht man wie folgt vor. Stößt man auf ein  , so heißt dies, daß man "a niederschreiben"muß. Dabei sollte man mittels Syntaxanalyse prüfen ob eine erlaubte Zeichenfolge erzeugt wird. Stößt man hingegen auf ein  so bedeutet dies, daß man an dieser stelle das Syntaxdiagramm für <e> benutzen soll, man dieses Syntaxdiagramm also quasi einsetzen soll.

Syntaxdiagramme und reguläre Ausdrücke in denen nur Terminale aus Σ vorkommen stellen sogenannte **reguläre Mengen** dar (das sind CH-3 Sprachen). Syntaxdiagramme im Allgemeinen stellen genau die CH-2 Sprachen dar.

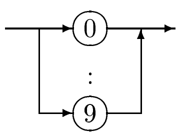
Ein weiteres Beispiel für Syntaxdiagramme anhand der Gleitkomma-Zahlen:



<Skalierung>:



<digit>:



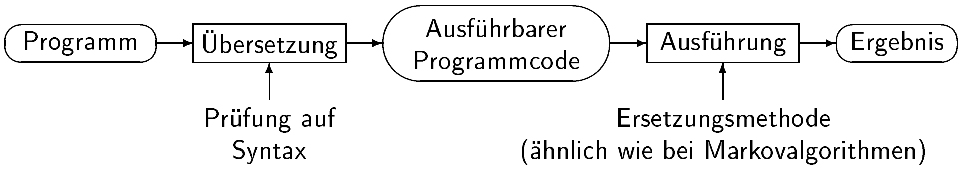
Siehe auch: MANFRED BROY: **Informatik Eine grundlegende Einführung** Band 1; 2.Auflage; 3. Programmiersprachen; Seite 77ff und: GERHARD GOOS: **Vorlesungen über Informatik** Band 1; 2.Auflage; 1.6.3 Chomsky-Grammatiken; Seite 39f

Kapitel 4

Funktionale Programme

4.1 Einführung in Gofer

GOFER-Programme werden vor ihrer Ausführung einer Vorverarbeitung oder **Übersetzung** unterworfen. Dabei wird die Konsistenz wichtiger Eigenschaften des Programms überprüft und das Programm in eine für die Interpretation geeignete, interne Darstellung überführt.



Funktionale Programme bestehen vor allem aus eigenen Funktionsdefinitionen $fv_1v_2 = F$ und Ausdrücken die mit Hilfe dieser Funktionen gebildet werden. Ein Beispiel für die Definition einer Funktion liefert uns der Kreisumfang:

```
pi = 3.1415926
kreisumfang r = 2.0 * pi * r
kreisfläche r = pi * r * r
```

Am GOFER-Prompt kann man einen Ausdruck mit Hilfe dieser und weiterer Standardfunktionen berechnen.

```
?  kreisumfang 2.0
12.5664
```

Funktionen sind Definitionen auf Tupeln von Wertemengen und bilden in Wertemengen ab. Diese Wertemengen bezeichnet man auch als Typ oder Sorte. Der Typ bzw. die Sorte einer Funktion gibt an, welchen Typ die Argumente dieser Funktion haben und welchen Typ das Ergebnis hat. In GOFER werden n-stellige Funktionen $fx_1x_2...x_n$ wie folgt aufgefaßt:

$$((...\underbrace{(fx_1)}_{\text{(n-1) stellige Funktion für festes } x_1}x_2)...)x_n)$$

$$((...\underbrace{((fx_1)x_2)}_{\text{(n-2) stellige Funktion für festes } x_1, x_2}...)x_n)$$

$$\underbrace{((...((fx_1)x_2)...)}_{\text{1 stellige Funktion für festes } x_1,...,x_{n-1}} x_n)$$

$$\underbrace{((...((fx_1)x_2)...)x_n)}_{\text{0 stellige Funktion für den Ergebniswert}}$$

Der Typ einer n-stelligen Funktionen ergibt sich dann wie folgt:

$(\mathtt{Typ}_1 \rightarrow (\mathtt{Typ}_2 \rightarrow (\mathtt{Typ}_3 \rightarrow \ \ldots \ \rightarrow \mathtt{Typ}_n)...))$

$\mathtt{Typ}_1,...,\mathtt{Typ}_{n-1}$ sind die Typen der Parameter
$\mathtt{Typ}_n$ ist der Ergebnistyp

Dabei kann man das Symbol `->` wie einen Operator verwenden, der dann rechtsassoziativ ist. Daher kann man auf die Klammerung in Typdefinitionen in Gofer verzichten.

Die Rekursion, also der Selbstaufruf von Funktionen, ist in Gofer besonders wichtig, da es als Funktionale Sprache keine Schleifen enthält. Die wichtigsten Typen von Rekursion sind:

- **lineare Rekursion**: in einer rekursiven Funktionsdefinition tritt in jedem Zweig einer Fallunterscheidung höchstens ein Aufruf der Funktion auf.
- **kaskadenartige Rekursion**: in einer rekursiven Funktionsdefinition treten in mindestens einem Zweig einer Fallunterscheidung zwei oder mehrere rekursive Aufrufe der Funktion auf.
- **vernestete Rekursion**: liegt vor, wenn in den Parameterausdrücken eines rekursiven Aufrufes der Funktion weitere rekursive Aufrufe der Funktion auftreten. Ein Beispiel:

```
qsum n = if(n<10)then n else qsum(qsum(n/10)+ (mod n 10))
```

Wertemengen mit Operationen bezeichnet man als **Rechenstrukturen** bzw. **Datentypen**. Beispiele für Rechenstrukturen sind:

Bool: Wahrheitswerte

Konstanten:	`True, Flase`
Operationen:	`not`, `==` (semantische Gleichheit), `&&`, `||`

Int: Ganze Zahlen
Dezimalzifferfolgen ohne führende Nullen

Operationen:	unäre:	`-, abs`
	binäre:	`+, -, *, ... , >, <, max, ==`

In Gofer existieren bereits zahlreiche Rechenstrukturen. Es ist aber ebenso möglich Rechenstrukturen selbst zu definieren.

Ein Ausdruck ist durch die Rechenstrukturen bestimmt, die zu den Funktionen gehören, die in diesen Ausdruck vorkommen. So existieren entsprechend der beiden Beispiele oben:

Boolesche Ausdrücke:	(wie die Booleschen Terme aber mit einer anderen Schreibweise)
Arithmetische Ausdrücke:	(ähnlich wie Boolesche Ausdrücke)

4.2 Bezeichner, Operatoren, Ausdrücke

Um Typen Funktionen und Parameter benennen zu können benötigt man **Bezeichner**. In GOFER müssen Bezeichner mit einem Buchstaben beginnen und können dann aus beliebig langen(?!?) Kombinationen von Buchstaben, Ziffern, Unterstrichen (`_`) und Apostrophen (`'`) bestehen. In BNF sind Bezeichner wie folgt definiert:

<Bezeichner> ::= <Buchstabe> |
<Bezeichner><Buchstabe> |
<Bezeichner><Ziffer> |
<Bezeichner><Unterstrich> |
<Bezeichner><Apostroph>

Allerdings ist bei der Wahl der Bezeichner Achtung geboten da es in GOFER, wie in jeder Programmiersprache, **reservierte Bezeichner** (**Wortsymbole**, **Schlüsselwörter**) gibt die nicht für die Benennung benutzereigener Strukturen verwendet werden dürfen. Zusätzlich gib es in GOFER zahlreiche **vordefinierte** oder **Standardfunktionen** für deren Namen das gleiche gilt. Weiterhin gilt das Bezeichner die mit Großbuchstaben beginnen Typen und Konstruktoren vorbehalten sind.

In GOFER können zweistelligen Funktionen wie Infixoperatoren verwendet werden in dem man sie in Apostrophe einschließt und umgekehrt Infixoperatoren wie zweistelligen Funktionen gebraucht werden, in dem man sie in Klammern einschließt.

Allgemein		Beispiel	
Präfixform	Infixform	Präfixform	Infixform
$f a\ b$	a ‘f‘ b	max $a\ b$	a ‘max‘ b
$(\tau)a\ b$	$a\ \tau\ b$	$(+)ab$	$a+b$

In der folgenden Tabelle sind die Vorrangregeln für die wichtigsten Infixoperatoren in fallender Priorität aufgeführt. Generell binden Funktionsaufrufe in Präfixform stärker als alle Infixoperatoren. Die Assoziativität beschreibt die Reihenfolge, in der Ausdrücke mit gleichrangigen Operatoren ausgewertet werden. Bei Linksassoziativität wird von links nach rechts `(1 - 2 - 3 = (1 - 2) - 3)`und bei Rechtsassoziativität von rechts nach links `(1 ^ 2 ^ 3 = 1 ^ ( 2 ^ 3))` geklammert.

Vorrang	Assoziativität	Operator
9	links	`!!`
9	rechts	`.`
8	rechts	`^`
7	links	`*`
7	—	`/, ‘div‘, ‘rem‘, ‘mod‘`
6	links	`+, -`
5	rechts	`++, :`
4	—	`==, /=, <, >, <=, >=`
3	rechts	`&&`
2	rechts	`||`

Wichtig für wohl jede Programmiersprache sind bedingte Ausdrücke. Sie erlauben es abhängig von einer Bedingung zwischen Alternativen zu wählen.

`if` b `then` e_1 `else` e_2 $b \in$ Bool e_1, e_2 müssen gleichen Typ haben
`case` e `of` $e_1, e_2, e_3, \ldots$

Nun einige Beispiele für GOFER-Funktionen:

1. ```
   fc::Int->Int
   fc n = if n==0 then 1 else n * fc(n-1)
   ```
2. ```
   fac::Int->Int
   fac 0 = 1
   fac (n + 1) = (n + 1) * fac n
   ```
3. ```
   min x y = if x <= y then x else y
   ```
4. ```
   ggT::Int->Int->Int
   ggt a b = if a==b then a else
        if a < b then ggT b a else ggT b(a - b)
   ```
5. ```
   ggT a = 0
   ggT a b = if a < b then ggT b a else ggT b(a - b)
   ```

4.2.1 Gültigkeitsbereiche

In GOFER-Programmen können nur gebundene Bezeichner oder Operatoren vorkommen (Funktionsdefinition und Ausdruck zusammen bilden einen geschlossen Term) Die Bedeutung eines Bezeichners b im Ausdruck F wird nach folgenden Regeln gebunden:

1. b kann lokal innerhalb von F definiert sein (`let, where`).
2. b kann Parameter einer Funktionsdefinition sein. Beim Aufruf der Funktion f wird das gegebene Argument a für b eingesetzt, also F[a/b] ausgewertet. b hat also a als Bedeutung.
3. b kann Funktionsname oder Operatorbezeichner sein. Dann bedeutet b in F die definierte Funktion. Bei der Auswertung von F wird diese Funktion aufgerufen. War die Definition b v = G, so wird beim Aufruf von b a durch G[a/v] ersetzt.

Die Stelle an der einem Bezeichner seine Bedeutung zugeordnet wird, wird **Definition** oder **Vereinbarung** des Bezeichners genannt. Da bei Parametern die Zuordnung eines Argumentes, also die eigentliche Definition, erst während des Aufrufes der Funktion erfolgt, spricht man hier statt dessen auch von einer **Spezifikation**. Unter den **Anwendungen** eines Bezeichners, versteht man alle die Stellen, an denen der Bezeichner mit der ihm zugeordneten Bedeutung verwendet wird. Denjenigen Teil eines Programmes oder eines Ausdruckes, in dem die Anwendungen des Bezeichners die Bedeutung aus der gegebenen Vereinbarung haben, bezeichnet man als **Gültigkeitsbereich** des Bezeichners. Bei der Suche nach der Bindung eines Bezeichners wendet GOFER die obigen Regeln in der Reihenfolge ihrer Numerierung an (1-2-3).

Wichtig ist auch, daß lokal vereinbarte Bezeichner gleichnamige globale Bezeichner überdecken. Dazu ein Beispiel:

```
b = 3
f a b = (g b) + a
g a = a + b
```

In der Funktion `f` wird die globale Definition von `b` dem Parameter `b` überdeckt. In `g` hingegen wird wieder die globale Definition von `b` verwendet.

Ein Mittel um Code in GOFER strukturierter zu gestalten stellt die Verwendung einer nachgestellten Vereinbarung von Bezeichnern mittels `where` dar. So kann man
```

statt:

```
f a b = (a + b) * (a + b)
```

auch:

```
f a b = x * x where x = a + b
```

schreiben. Die nachgestellte Vereinbarung von Bezeichnern mittels `where` betrifft Fall 1 der Bindungsregeln. Es ist ebenso möglich diese Bezeichner mittels `let` vor ihrer Verwendung zu definieren. Unser Beispiel würde dann wie folgt aussehen:

```
f a b = let x = a + b in x * x
```

In GOFER (aber nicht in allen funktionalen Sprachen) werden die Argumente faul - das heißt so spät wie möglich - ausgewertet. So erfolgt die Substitution F[a/b] aus Fall 2 der Bindungsregeln nur bei akutem Bedarf, d.h. wenn die Berechnung von F dies erfordert. Auch im Fall 3 wird der Aufruf b a nur bei Bedarf durch F[a/b] ersetzt. Ferner wird ein Ausdruck der an mehreren Stellen auftritt nur einmal ausgewertet. Dazu ein Beispiel: :

```
?  square * square where square = 3 * 3
81
(3 reductions, 9 cells)
?  (3 * 3) * (3 * 3)
81
(5 reductions, 11 cells)
```

Bei der Auswertung von `?  square * square where square = 3 * 3` geht Gofer wie folgt vor:

1. Berechne den Wert von `square` durch Reduktion von $3 * 3$ zu 9.
2. Setze die 9 für `square` im Ausdruck `square * square` ein.
3. Reduziere `9 * 9` zu 81.

Die faule Auswertung kann in GOFER auch verhindert werden. Mit Hilfe der Funktion `strict` wird die sogenannte strikte Auswertung erzwungen.

Siehe auch: GERHARD GOOS: **Vorlesungen über Informatik** Band 1; 2.Auflage; 5.2 Grundlagen funktionaler Programmiersprachen; Seite 226ff

4.3 Daten und elementare Datentypen

Die folgende Übersicht zeigt die elementaren (eingebauten) Datentypen von Gofer:

Typ	Bedeutung
`Bool`	boolesche Werte
`Int`	ganze Zahlen
`Float`	Gleitpunktzahlen
`Char`	Einzelzeichen
[t]	Listen mit Elementen des Typs t
`String`	`[Char]`, Texte
$(t_1, ..., t_n)$	n-Tupel
()	Typ des Null-Tupels (**Einheitstyp**)
t_1 -> t_0	Funktionstyp

4.3.1 Boolesche Werte

Der Datentyp `Bool` verfügt über die beiden Werte `True` und `False`. Bei der Auswertung des Ausdruckes `p && q && r` , wird der zweite Boolesche Ausdruck `q && r` nur dann berechnet, wenn `p True` ist, da ja für denn Fall das `p False` ist der gesamte Ausdruck `False` ist.

4.3.2 Integer und Float

Integer Zahlen werden wie für ganze Zahlen üblich geschrieben, also ohne Nachkommastellen. Gleitpunktzahlen werden als Dezimalbrüche mit dem amerikanischen Dezimalpunkt statt dem Komma geschrieben. Diese Unterscheidung ist unbedingt einzuhalten. Wichtig beim Umgang mit diesen Datentypen ist, daß sie nur einen beschränkten Zahlenbereich umfassen. Der Integer Datentyp von Gofer ist mit 16 Bits codiert dementsprechend umfaßt er nur den Zahlenbereich von -32786 bis 32767. Der Datentyp `Float` umfaßt Werte zwischen -10^{38} und 10^{38}. Allerdings muß man beachten das die Genauigkeit der Float-Zahlen nur 8 Stellen beträgt. Aufgrund dieser Einschränkungen gelten nicht alle Gesetze der Arithmetik für diese Datentypen (Assoziativgesetz, Distributivgesetz). Ebenso sollte man es tunlichst vermeiden zwei Gleitkommazahlen auf Gleichheit zu testen.

4.3.3 Zeichen

Der Typ `Char` umfaßt alle ASCII Zeichen deren Dezimalcode zwischen 0 und 127 liegt. Die beiden Funktionen `ord::Char->Int` und `chr::Int->Char` wandeln Zeichen in den entsprechenden Dezimalcode um und umgekehrt.

4.3.4 Listen

Listen werden in GOFER in eckigen Klammern geschrieben. Die einzelnen Elemente werden durch Kommata getrennt: `[1,2,3]` ist eine Liste mit drei Elementen. Die leere Liste wird mit `[]` bezeichnet. Es ist des weiteren möglich Listen mit dem Doppelpunktoperator zu verlängern: `1:[2] = 1:2:[] = [1,2]` Listen können auch geschachtelt werden, so ist `[[2],[3,4]]` eine Liste deren Elemente selbst wieder Listen sind.

Der Typ einer Liste wird mit `[t]` bezeichnet. Dies erzwingt, daß alle Elemente einer Liste bzw. alle Elemente von Unterlisten über den selben Typ verfügen müssen. Es ist eine verbreitete Konvention Listen durch ein an den Elementbezeicher angehängtes `s` zu bezeichnen. Das Muster `(x:xs)` bezeichnet mit `x` das erste Element einer Liste und mit `xs` den Rest der Liste. Dieses Muster paßt als Argument auf alle Listen, die nicht leer sind.

Das $(n + 1)$ Element einer Liste erhält man mit `xs !! n`. Dabei `n` muß zwischen 0 und `length xs -1` liegen. `n` wird als Index bezeichnet. Zwei Listen kann man mit Hilfe des Operators ++ miteinander verknüpfen, wobei aber beide Listen über den selben Typ verfügen müssen. Vergleichsoperationen auf Listen setzen voraus, daß die Vergleichsoperation für den Typ der Liste definiert ist.

4.3.5 Funktionen als Werte

Funktionen die andere Funktionen als Argument haben werden als **Funktionale** oder **Funktionen höherer Ordnung** bezeichnet. Ein Beispiel aus der Mathematik ist der Differentialoperator. Er ist ein Funktional das die Funktion $f(x)$ angewendet wird um die Ableitung $f'(x)$ dieser Funktion zu erhalten.

Um auch zusammengesetzte Funktionen als Argumente an Funktionale übergeben zu können, benötigt man die Funktionskomposition. Sie wird in durch einen als Operator definierten Punkt wiedergegeben.

$$(f.g)\ x == f(g\ x)$$

Die Funktionskomposition ist rechtsassoziativ, also gilt bei einstelligem g, h und zweistelligem f:

$$(f.g.h)\ x\ y == (f.g)(h\ x)\ y == f(g(h\ x))\ y$$

Bei der Faltung werden die Elemente einer Liste mit Hilfe einer zweistelligen Funktion links- oder rechtsassoziativ miteinander verknüpft. Ist die Liste leer so wird dem Ergebnis das neutrale Element der Funktion zugeordnet, falls ein solches existiert.

Die **Faltung** ist also eine Funktion, welche eine andere Funktion als Parameter besitzt, und diese auf die Elemente einer Liste anwendet. Somit ist die Faltung ein Funktional. Die linksassoziative Form der Faltung läßt sich wie folgt definieren:

```
foldl::(t->t'->t)->t->[t']->t
foldl f z [] = z
foldl f z (y:ys) = foldl f (f z y) ys
```

`f` ist die Funktion die auf die Liste angewendet werden soll
`z` ist das neutrale Element

Die rechtsassoziative Form ist:

```
foldr::(t'->t->t)->t->[t']->t
foldr f z [] = z
foldr f z (y:ys) = f y (foldr f z ys)
```

Für die Multiplikation lauten die entsprechenden Funktionen damit:

`productl xs = foldl (*) 1 xs` linksassoziativ
`productr xs = foldr (*) 1 xs` rechtsassoziativ

Die Funktionen `foldr` und `foldl` können bei assoziativen Operationen (Addition, Multiplikation) eingesetzt werden.

Auch die Anwendung einer einstelligen Funktion auf alle Elemente einer Liste ist ein oft benötigtes Funktional:

```
map::(t-> t')->[t]->[t']
map f [] = []
map f (x:xs) = (f x) :  map f xs
```

Ein Beispiel hierfür ist:

```
quad x = x*x

?map quad [1,2,3,4]
[1,4,8,16]
```

Werden zwei einstellige Funktionen f, g auf die `map` Funktion angewendet so gilt wegen der Assoziativität der Funktionskomposition:

$$\texttt{map}(f.g)\ xs == \texttt{map}\ f(\texttt{map}\ g\ xs)$$

Des weiteren kann es von Interesse sein nur diejenigen Elemente einer Liste auszuwählen, die bestimmten Bedingungen genügen:

```
filter::(t->Bool)->[t]->[t]
filter p [] = []
filter p (x:xs) |p x = x:xs'
                |otherwise = xs'
                where xs' = filter p xs
```

Mit dieser Funktion ist es zum Beispiel möglich alle geraden Zahlen aus einer Liste heraus zu sortieren:

```
?filter even [1,2,3,4,5,6,7,8,9]
[2,4,6,8]
```

Funktionen können jedoch nicht nur als Parameter an andere Funktionen übergeben werden, sondern sie können auch das Ergebnis einer Funktion sein. Als Beispiel dienen hier die `nand` und die `nor` Funktion:

```
nand_oder_nor f a b = f (not a) (not b)
nand a b = nand_oder_nor (||) a b
nor a b = nand_oder_nor (&&) a b
```

Man bezeichnet `nand` und `nor` als partielle Instanzen von `nand_oder_nor`.

4.3.6 Polymorphismus und Overloading

Die Verwendung von Typvariablen für die Kennzeichnung der Typen wird als **Polymorphie** bezeichnet. Somit ist es möglich ein und die selbe Funktion auf Argumente verschiedenen Typs anzuwenden. Ein Beispiel dafür ist die `map` Funktion:

$$\texttt{map::}\ \underbrace{\texttt{(a->b)}}_{\substack{\text{Typ des}\\ \text{Funktions-}\\ \text{argumentes}}}\ \texttt{->}\ \underbrace{\texttt{[a]}}_{\substack{\text{Typ des}\\ \text{Argumentes}}}\ \texttt{->}\ \underbrace{\texttt{[b]}}_{\text{Ergebnistyp}}$$

`a,b` beliebige Typen

Generell: Ein Objekt dessen Typ Variablen für Typen enthält wird als polymorphes Objekt bezeichnet es besitzt polymorphe Typen. Andere polymorphe Funktionen sind zum Beispiel:

```
(++)::[a] -> [a] -> [a]
concat::[[a]] -> [a]
```

Anders funktioniert das **Überladen** von Funktionen. Hier werden einem Funktionssymbol oder Operator unterschiedliche Funktionen zugeordnet. Welche Funktion gemeint ist, wird statisch in Abhängigkeit vom Typ der Argumente entschieden. Ein Beispiel dafür ist die Addition, sie ist sowohl für Gleitkommazahlen als auch für die Ganzen Zahlen definiert.

4.3.7 Texte

Texte sind Listen von Zeichen (`"abc"== ['a', 'b', 'c']`), also ist der Typ von Texten `[Char]`. Texte können somit endlich viele Zeichen enthalten. Enthält ein Text keine Zeichen `""` so wird er als leerer Text bezeichnet. Der leere Text entspricht der leeren Liste `[]`. Der Texttyp ist also lediglich ein Synonym für einen schon bekannten Typ.

```
type String = [Char]
```

Damit sind auf Texte alle bekannten Listenoperationen anwendbar.

4.3.8 Tupel

Tupel ($t_1, \ldots, t_n$) besitzen im Gegensatz zu Listen eine feste Länge. Des weiteren können die Elemente eines Tupels von verschiedenem Typ sein. Das Tupel mit der Länge Null wird als 0-Tupel `()` bezeichnet, es ist praktisch bedeutungslos. Das 1-Tupel `(t)` ist identisch mit seinem einzigen Element `t`. Mit 2-Tupeln läßt sich schon wesentlich mehr anstellen. Sie können beispielsweise für die komplexen Zahlen oder Koordinatenpaare verwendet werden. Tupel werden also im wesentlichen dazu verwendet zusammengehörige Werte als eine Einheit aufzufassen.

4.4 Muster (Pattern)

Eine Funktion zur Berechnung der Fakultät läßt sich in GOFER wie folgt definieren:

```
fak1 n = if n == 0 then 1 else n * fak1(n - 1)
```

Ebenso währe die folgende Definition denkbar:

```
fak2 0 = 1
fak2 n = n * fak2(n - 1)
```

Allerdings prüfen beide Funktionen nicht ob `n` $\geq$ `0` gilt. Diese Überprüfung kann man mittels sogenannter **bewachter Muster** hinzufügen:

```
fak3 n| n == 0 = 1
      | n >= 1 = n * fak3(n - 1)
```

Ein Muster paßt, wenn der Parameter auf das Argument paßt und die rechts vom Strich gegebene Bedingung erfüllt ist. Diese Bedingung wird als **Wächter** bezeichnet. Pro Wächter (| condition) muß eine Definition für den Funktionswert vorgenommen werden. Es sind endlich viele Wächter erlaubt. Bei Funktionen mit mehreren Parametern, können diese vom Wächter in Beziehung zueinander gesetzt werden. Die allgemeine Form eines bewachten (guarded) Muster ist wie folgt definiert:

$$
\begin{array}{ll}
\mathsf{f}\ \mathsf{x}_1,\ \mathsf{x}_2\ \ldots\ \mathsf{x}_n & |\ \mathsf{cond}\ 1 = \mathsf{e}_1 \\
 & |\ \mathsf{cond}\ 2 = \mathsf{e}_2 \\
 & \ldots. \\
 & |\ \mathsf{cond}\ \mathsf{m} = \mathsf{e}_m \\
 & |\ \mathsf{otherwhise} = \mathsf{e}_{m+1}
\end{array}
$$

cond n	Boolesche Ausdrücke
e_n	Ausdrücke mit Werten vom Typ des Funktionswertes

Ein weitere Möglichkeit die Ausführung von Funktionen an Bedingungen zu knüpfen stellen die **(n + k) Muster** dar. Das $(n + 1)$ Muster paßt auf ein Argument a, wenn gilt das $a \geq 1$. Da auf $n + 1 = a$ geprüft wird, erfolgt letztendlich ein Test auf $n \geq 0$. Eine Fakultätsfunktion ließe sich nun auch wie folgt realisieren:

```
fak4 0 = 1
fak4 (n + 1) = (n + 1) * fak4 n
```

Der **Joker _** dient als Bezeichner eines Parameters, auf dem im Funktionsrumpf nicht zugegriffen werden soll, daher kann er dort nicht verwendet werden. Allerdings können Joker mehrfach in dem selben Ausdruck vorkommen. Im folgenden sind Muster aus Gofer kurz zusammengestellt.

Bezeichnung	**Beispiel**	**Erläuterung**
Variable	`v`	einfache Variable, paßt immer
Konstante	`3, 'c'`	paßt auf den Wert des entsprechenden Typs
Joker	`_`	paßt immer
Tupel	`(x,y)`	paßt auf entsprechendes Tupel
Listen	`[]`	paßt auf leere Liste
	`[x]`	paßt auf einelementige Liste, andere Anzahlen analog
	`(x:xs)`	paßt auf Liste mit mindestens einem Element, andere Anzahlen analog
(n + k)-Muster	`(n + 1)`	paßt auf ganze Zahlen ≥ k, k muß Konstante sein

Siehe auch: Gerhard Goos: **Vorlesungen über Informatik** Band 1; 2.Auflage; 5.3 Daten und elementare Datentypen; Seite 235ff

4.5 Benutzerdefinierte Datenstrukturen

Die Definition neuer Typen erfolgt durch eine sogenannte Vereinbarungen (engl. declarations):

$\texttt{data Datentyp } a_1 \texttt{ ... } a_n \texttt{ = } constr_1 \texttt{ | ... | } constr_m$

Datentyp:	Name des neuen Datentyps mit der Stelligkeit n
$a_1 \ldots a_n$:	verschiedene Typvariablen
$constr_1 \ldots constr_m$:	Konstruktoren

Die Konstruktoren beschreiben wie die Elemente (Werte) des neuen Datentyps konstruiert werden. Die Form der Konstruktoren stellt sich wie folgt dar:

$\texttt{Name } typ_1 \ldots typ_r$

`Name` steht für die Bezeichnung des Konstruktors, sie beginnt immer mit einem Großbuchstaben. Mit dieser Definition ist eine Funktion folgenden Typs definiert:

$typ_1 \texttt{ -> ... -> } typ_r \texttt{ -> Datentyp } a_1 \ldots a_n$

Wichtig sind des weiteren die Konstruktoroperatoren $typ_1 \oplus typ_2$, wobei $\oplus$ die neue Bezeichnung für einen Konstruktoroperator ist. Ein Konstruktoroperator beginnt immer mit einem Doppelpunkt (:). Generell gilt das in den Typausdrücken der Konstruktoren nur die Typen $\mathtt{a}_i$ vorkommen dürfen. Zum besseren Verständnis diese Themas folgen nun einige Beispiele:

```
data Day = Sun|Mon|Tue|Wed|Thu|Fri|Sat

what_shall_I_do::Day -> String
what_shall_I_do Sun = "relax"
what_shall_I_do Sat = "go shopping"
what_shall_I_do _ = "looks like ...  "

data Temp = Celsius Float | Fahrenheit Float

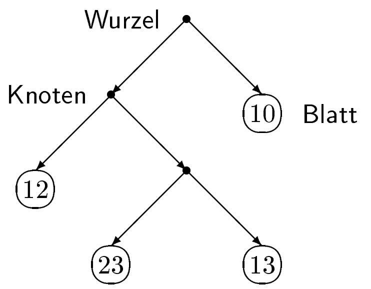
freezing::Temp -> Bool
freezing (Celsius tmp) = tmp <= 0.0
freezing (Fahrenheit tmp) = tmp <= 32.0


data Set a = Set[a]   Typ und Konstruktor gleich bezeichnet

Set[1,2,3]            ist Element des Datentyps Set Int
Set['a','b']          ist Element des Datentyps Set Char
```

4.5.1 Binärbäume

Bäume sind sehr nützliche abstrakte Datenstrukturen mit einem breit gefächerten Anwendungsgebiet. Ein **Baum** ist ein kreisloser ungerichteter Graph. Im allgemeinen besitzt ein Baum an jedem Knoten beliebig viele Kinder. Ein Baum heißt **Binärbaum**, wenn jeder Knoten genau zwei Kinder besitzt.



Ein Baum ist ein Graph mit Kanten (Äste) und Knoten (Wurzel, innere Knoten, Blätter)

Eine Möglichkeit in GOFER einen Binärbaum als Datenstruktur zu definieren ist die folgende:

$$\texttt{data Tree a =}\ \underbrace{\texttt{Leaf a}}_{\text{Konstruktor-funktion}}\ \mid\ \underbrace{\texttt{Tree a :\^{}: Tree a}}_{\text{Konstruktoroperator (macht aus zwei Bäumen einen)}}$$

Unser Bildchen von oben würde in GOFER wie folgt aussehen:

```
(Leaf 12:^:(Leaf 23:^:Leaf 13)):^:Leaf 10
```

Eine mögliche Funktion auf Bäume wäre das konvertieren des Baumes in eine lineare Datenstruktur:

```
leaves::  Tree a -> [a]
leaves (Leaf l) = [l]
leaves (l:^:r) = leaves l ++ leaves r
```

4.5.2 Keller

Keller gehören wie auch die Bäume zu den wichtigsten Datentypen in der Informatik. Sie eignen sich zum Beispiel zum rekursiven Abarbeiten von geschachtelten Termen. Die Spezifikation dieses abstrakten Datentyps Keller ist im folgenden kurz vermerkt:

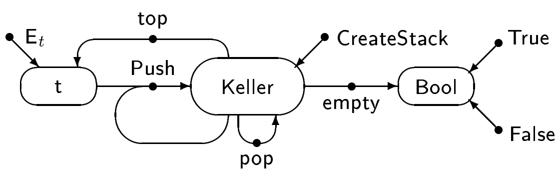
```
data Keller t = CreateStack | Push(Keller t) t

pop::Keller t -> Keller t
top::Keller t -> t
empty::Keller t -> Bool

pop(Push k x) = k
top(Push k x) = x
empty(CreateStack) = True
empty(Push k x) = False
```

Axiome

In einem Syntaxdiagramm stellt sich dieser Datentyp wie folgt dar:



Es ist auch möglich die Implementierung eines Kellers mittels Listen vorzunehmen:

```
type Keller t = [t] Typsynonym
createStack = []
push k x = x:k
top (x:xs) = x
pop (x:xs) = xs
empty [] = True
empty _ = False
```

Im allgemeinen ist aber ein Beweis nötig, der zeigt das diese Implementierung die Axiome erfüllt.

4.5.3 Schlangen

Unter Schlangen versteht man in der Informatik Warteschlangen. Am bekanntesten sind **FIFOs** (First In First Out) und **LIFOs** (Last In First Out). Die ersteren geben das am längsten wartende Element als erstes aus der zweite Typ verfährt genau umgekehrt, es handelt es sich also um einen Keller. Es gibt jedoch noch weitere

Typen von Schlangen. Zum Beispiel gewichtete Schlangen mit Gewichten oder Prioritäten für ihre Elemente. Eine allgemeine Signatur für Schlangen (Queues) in GOFER ist:

```
data Schlange t = CreateQueue | Enqueue(Schlange t) t
dequeue::  Schlange t -> Schlange t
front::  Schlange t -> t
empty::  Schlange t -> Bool
```

Die entsprechenden Axiome für eine FIFO Queue sind:

```
dequeue(Enqueue CreateQueue a) = CreateQueue
dequeue(Enqueue s a) = Enqueue(dequeue s)a
front(Enqueue CreateQueue a) = a
front(Enqueue s a) = front s
empty CreateQueue = True
empty(Enqueue s a) = False
```

Schlangen lassen sich, wie wir ja schon am Spezialfall des Kellers gesehen haben, sehr einfach mittels Listen implementieren.

Siehe auch: GERHARD GOOS: **Vorlesungen über Informatik** Band 1; 2.Auflage; 6 Abstrakte Datentypen; Seite 287ff

Kapitel 5

Zur Theorie der Programmierung

5.1 Rechenstrukturen und Abstrakte Datentypen

5.1.1 Rechenstrukturen und Abstrakte Datentypen

Unter einer **Rechenstruktur** versteht man eine Datenmenge (Werte) mit darauf definierten (konkreten) Funktionen. Die Spezifikation einer Rechenstruktur wird als **Abstrakter Datentyp (ADT)** bezeichnet. Diese Spezifikation muß für den "normalen" Benutzer nicht ersichtlich sein. Er spricht dieses Objekt nur über seine Schnittstelle an (Kapselung , Geheimnisprinzip).

Definition **Rechenstrukturen**: Seien S (Sorten) und F (Funktionen) Mengen von Bezeichnern; eine Rechenstruktur A besteht aus einer Familie $\{s^A : s \in S\}$, von konkreten Trägermengen s^A von Werten, und einer Familie $\{f^A : f \in F\}$, von Funktionen bzw. Operatoren f^A zur Manipulation von Werten.

$$A = (\{s^A : s \in S\}, \{f^A : f \in F\})$$

s^A ist eine Menge mit der Sortenbezeichnung s. Für jede Funktion f^A existiert ein $n \in \mathbb{N}$, so daß f^A eine n-stellige Funktion ist, mit:

$$f^A : s_1^A \times s_2^A \times ... \times s_n^A \to s_{n+1}^A$$

Eine nullstellige Funktion entspricht einer Konstante.

Jedes s^A enthält das spezielle Element $\perp$ (Bottom) zur Repräsentation der Tatsache, daß Funktionen bei Anwendung auf bestimmte Argumente kein Ergebnis liefern $f^A(...) = \perp$. Eine Funktion heißt strikt, wenn gilt: Ist eines der Argumente der Funktion $\perp$, so ist auch das Ergebnis von f^A $\perp$. In GOFER sind Funktionen im allgemeinen nicht strikt (wegen lazy evaluation).

5.1.2 Signaturen

Um festzulegen welche Funktionssymbole und Sorten in einer Rechenstruktur auftreten und in welcher Weise die Funktionssymbole sinnvoll miteinander verknüpft werden können, werden **Signaturen** verwendet.

Definition: Eine Signatur Σ ist definiert als Paar: $\Sigma = (S, F)$ wobei S die Sorten (d.h. Namen für die Trägermengen) bezeichnet und F die Funktionssymbole mit Angabe ihrer Funktionalität.

Für jede Funktion $f \in F$ sei eine Funktionalität $\mathsf{fct} \in S$ gegeben. Nun schreibt man zur Angabe der Funktionalität einer Funktion:

$$\mathsf{fct}\ f = (s_1, s_2, ..., s_n) s_{n+1}$$

5.1.3 Grundterme

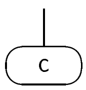
Sei W_Σ^s die Menge der **Grundterme** der Sorte s mit $s \in S$ induktiv definiert durch:

1. Jedes nullstellige Funktionssymbol $f \in F$ mit $\mathsf{fct}\ f = s$ ist Grundterm der Sorte s.

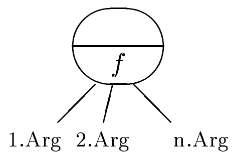
2. Jede Zeichenreihe $f(t_1, ..., t_n)$ mit $f \in F$ und $\mathsf{fct}\ f = (s_1, s_2, ..., s_n)s$ ist ein Grundterm der Sorte s, falls t_i ein Grundterm der Sorte s_i, für alle $i = 1, ..., n$, ist.

Mit W_Σ wird die Menge aller Grundterme zu Σ bezeichnet. Grundterme lassen sich auch als Bäume darstellen. Die Knoten entsprechen dabei dem Funktionssymbol $f \in F$ und die Anzahl der Kinder eines Knotens der Stelligkeit der Funktion bzw. der Anzahl der Argumente.

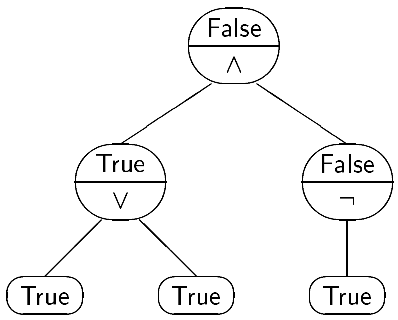
Nullstellige Funktion:



n-stellige Funktion:



Beispiel: $(\mathsf{True} \vee \mathsf{True}) \wedge (\neg\ \mathsf{True})$



Zweiteilung der Knoten für Werberechnung bzw. Auswertung der Funktion.

Sei eine Signatur Σ mit $\Sigma = (S, F)$ gegeben. Dann existiert auf W_Σ zu jedem $f \in F$ eine konkrete Funktion f_W über W_Σ. Sei $\mathsf{fct}\ f = (s_1, ..., s_n)s_{n+1}$, dann ist:

$$f_W : W_\Sigma^{s_1} \times W_\Sigma^{s_2} \times ... \times W_\Sigma^{s_n} \rightarrow W_\Sigma^{s_{n+1}}$$

Seien weiterhin $a_1...a_n$ Grundterme mit a_i Element die Menge der Grundterme der Sorte s_i ($a_i \in W_\Sigma^{s_i}, i = 1...n$); dann gilt:

$$f_W(a_1...a_n) = f(a_1...a_n)$$

Damit ist (W_Σ, F_W) eine Grundtermalgebra:

$$F_W = \{f_W | f \in F\}$$

Sie wird als **initiale Grundtermalgebra** bezeichnet, da man bei der induktiven Definition der Terme mit Konstanten beginnen muß. Ein Beispiel für eine initiale Grundtermalgebra ist die folgende Definition der natürlichen Zahlen:

date Nat = Null | Succ Nat

Grundterme sind:

Null, Succ(Succ(... (Null) ..))

5.1.4 Axiome

Wann werden Axiome für Abstrakte Datentypen und Rechenstrukturen benötigt?

1. Wenn nicht je zwei verschiedene Grundterme auch zwei verschiedene Werte der Rechenstruktur darstellen sollen. Mit der Bedingung, daß verschiedene Terme den gleichen Wert repräsentieren ist es möglich Ausdrücke auszuwerten. Zum Beispiel bei der Rechenstruktur Bool:

   Axiom: True $\vee$ True = True

2. Wenn zusätzlich zu den Konstruktoren weitere Operationen bzw. Funktionen auf die Rechenstruktur benötigt werden.

Mittels der Axiome wird die Grundtermalgebra W_Σ in Σ auf konkrete Rechenstrukturen abgebildet (Implementierung). Die Auswertung von Ausdrücken ist eine Umformung von Grundtermen in einfachere Formen.

5.1.5 Terme mit Identifikatoren

Zu $\Sigma = (S, F)$ kommt eine Familie $X = \{X_s | s \in S\}$ von paarweise disjunkten Mengen X_s von Identifikatoren, die auch disjunkt zu F und S sind. Mit $W_\Sigma(X)$ bezeichnen wir die um X erweiterte Termalgebra $W_{\Sigma'}$ mit $\Sigma' = (S, F \cup \{x_s | x_s \in X_s, s \in S\}$ und fct $x_s = s$ für $x_s \in X_s$, daß heißt x_s ist nullstellig.

$W_\Sigma(X)$ ist also die Menge der Terme mit freien Identifikatoren aus X, dabei können diese Terme auch frei von Identifikatoren sein. Es gilt damit $W_\Sigma = W_\Sigma(X)$.

Ein Term mit mindestens einer Variable repräsentiert eine Funktion. Die Funktionsdefinition (rechte Seite) ist ein Term mit freien Identifikatoren. Das Anwenden der Funktion auf konkrete Argumente wird als Ersetzen der freien Identifikatoren durch Grundterme bezeichnet oder auch als Substitution.

Siehe auch: Manfred Broy: **Informatik Eine grundlegende Einführung** Band 1; 2.Auflage; 2.2 Rechenstrukturen; Seite 45ff

5.2 Substitution und Termersetzung

5.2.1 Substitution

Die formale Definition der Substitution ist: Sei t ein Term der Sorte s mit freien Identifikatoren x, y der Sorte s. Sei weiterhin $f \in F$ mit fct $f = (x_1, ..., x_n)x_{n+1}$ und damit t_i ein Term der Sorte s_i $(i = 1...n)$. Dann gilt:

1. $x[t/x] = t$ $\quad$ $[t/x]$ = ersetze x durch t
2. $y[t/x] = y$ $\quad$ für $y \neq x$
3. $f(t_1, ..., t_n)[t/x] = f(t_1[t/x], ..., t_n[t/x])$

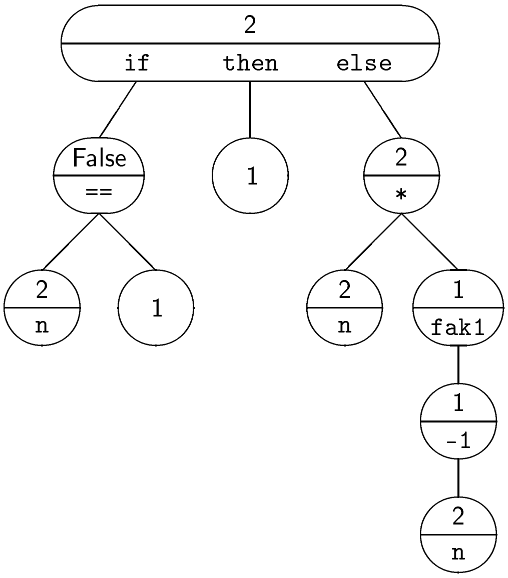
In Bezug auf die Darstellung der Terme als Bäume bedeutet die Substitution ein Ersetzen der Blätter mit Identifikatoren durch die entsprechenden Bäume.

Anwendung auf Gofer

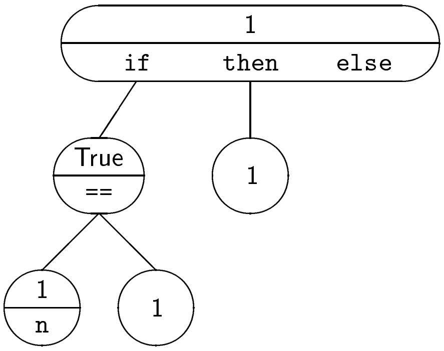
In Bezug auf ein GOFER-Programm entspricht die Funktionsdefinition einem Term mit freien Identifikatoren. Ein Ausdruck entspricht einem Grundterm. Die Ausführung eines Programmes läßt sich als iterierte Substitution oder genauer Termersetzung verstehen. Als Beispiel verwenden wir die Fakultätsfunktion:

```
fak1 n = if (n==1) then 1 else n * fak1(n -1)
```

Formularbaum:



Kopie des Formularbaums für den rekursiven Aufruf der Funktion:



5.2.2 Termersetzung

Die Bedeutung eines funktionalen Programmes (in GOFER) ist das Ergebnis von Termersetzungsfolgen (Reduktionsfolgen). Bei der Termersetzungssemantik funktionaler Programme erfolgt die Termersetzung analog zu der Textersetzung der Markovalgorithmen bzw. Chomsky-Grammatiken.

Ein Termersetzungssystem ist eine Menge von Regeln der Form $l \to t$. Dabei sind l und r Terme), jedoch können anders als bei Textersetzungssystemen in l und r aber Variable vorkommen. Diese Regeln stammen von den Axiomen der Allgemeinen Datentypen (zum Beispiel `not True` $\to$ `False`) und auch von Funktionsdefinitionen. So zum Beispiel:

```
fak2 0 → 1
fak2 n → n * fak2(n-1)
```

In einem Termersetzungsschritt wird ein Unterterm t an einer Stelle p mittels der Regel $l \to r$ ersetzt durch:

1. Suche simultane Substitution $G = [a_1/x_1, ..., a_n/x_n]$ (mit Variablen x_i und den Termen a_i), so daß $lG = t$. Es erfolgt also eine Anpassung von l an t.

2. Ersetze t nun durch rG, also ein angepaßtes r.

Ein Beispiel für diese Termersetzung liefert uns die folgende Regel:

```
x * (y + z) → x * z + y * z
```

Sie soll auf den folgenden Term angewendet werden:

```
(a + b) * ((a + d) + e * f) + g
```

Demzufolge ergibt sich G wie folgt:

$G = [$`(a + b)/x, (c + d)/y, e * f/z`$]$

das Ergebnis der Termersetzung ist:

```
(a + b) * (a + d) + (a + b) * e * f + g
```

Das Auswerten von Ausdrücken in funktionalen Sprachen erfolgt durch sukzessives Anwenden von Termersetzungsschritten. Wichtig: Die Regeln für allgemeine Datentypen und Funktionsauswertung müssen (unabhängig von de Reihenfolge) determinierte Ergebnisse liefern.

5.2.3 Die Auswertung von rekursiven Funktionen

Sei die folgende Funktionsdefinition gegeben: $\texttt{f x}_1 \texttt{ ... x}_n \texttt{ = F}$. Wobei diese Definition sowohl rekursive Aufrufe von `f` als auch bewachte Gleichungen enthält. Die Regel für die Auswertung von $\texttt{f a}_1 \texttt{ ... a}_n$ mit den Termen $\texttt{a}_i$ lautet dann wie folgt: $\texttt{f a}_1 \texttt{ ... a}_n \to \texttt{F}[\texttt{a}_1 \texttt{ /x}_1 \texttt{ ... a}_1\texttt{/x}_n] \texttt{ where f x}_1 \texttt{ ... x}_n \texttt{ = F}$.

5.2.4 Erzwingen von Reihenfolge

Durch Termersetzungsregeln mit Bedingungen ist es möglich Reihenfolge bei den Termersetzungsschritten zu definieren. Dies geschieht durch Termersetzungsregeln der Form $l_1 \to r_1 \wedge l_2 \to r_2 \wedge ... \wedge l_n \to r_n \Rightarrow l_{n+1} \to r_{n+1}$. Eine solche Regel bedeutet, wenn es möglich ist die Regeln $l_1 \to r_1$ bis $l_n \to r_n$ anzuwenden, dann wende auch die Regel $l_{n+1} \to r_{n+1}$ an. Ein Beispiel dafür ist:

1. Regel $(x \to y) \Rightarrow (x \vee y \to z \vee y)$
2. Regel $\texttt{True} \vee v \to \texttt{True}$

Die Anwendung auf $a \vee b$ liefert die faule Auswertung:

$$\text{Wenn } \underbrace{a \to \texttt{True} \text{ dann } a \vee b}_{\text{1. Regel}} \to \underbrace{\texttt{True} \vee b \to \texttt{True}}_{\text{2. Regel}}$$

In *Informatik - Eine grundlegende Einführung* von MANFRED BROY findet sich die folgende Semantik für Konstrukte mit bedingten Regeln: :

B → B′ ⇒ **if** B **then** E **else** E′ **fi** → **if** B′ **then** E **else** E′ **fi**
if true **then** E **else** E′ **fi** → E
if False **then** E **else** E′ **fi** → E′

Siehe auch: MANFRED BROY: **Informatik Eine grundlegende Einführung** Band 1; 2.Auflage; 2.3 Algorithmen als Termersetzungssysteme; Seite 61ff

5.3 Funktionsabstraktion und λ-Kalkül

Es ist in GOFER möglich anonyme (namenlose) Funktionen mittels des sogenannten Lambda-Ausdrucks zu definieren. Zur Einleitung eines Lambda-Ausdrucks wird in GOFER ein Backslash als stilisiertes Lambda verwendet. Ein solcher Ausdruck hat die folgende Syntax:

\atomicPatterns → Expression

Dazu ein Beispiel:

```
?(\ x y -> x + y)2 3
5
```

`\ x y -> x + y` beschreibt `f x y = x + y`

Der Übergang von `x + y` zu der anonymen Funktion `( x y -> x + y)` liefert die Funktionsabstraktion oder den λ-Ausdruck. In der theoretischen Informatik schreibt man $(\lambda\, \mathsf{x\ y}\ .\ \mathsf{x} + \mathsf{y})$ statt `(\ x y -> x + y)`.

5.3.1 Funktionsabstraktion

Sei E ein Term der Sorte s, so wird die Schreibweise

$$(s_1x_1, ..., s_nx_n)s : E$$

als **Funktionsabstraktion** bezeichnet. Die $x_1, ..., x_n$ werden als formale Parameter bezeichnet. Die Identifikatoren x_i sind in E gebunden und stehen dort für Ausdrücke der Sorte s_i. Der Ausdruck $(s_1x_1, ..., s_nx_n)s$ heißt Kopfzeile der Funktionsabstraktion, sie gibt deren Funktionalität wieder. E wird als Rumpf (body) bezeichnet. Funktionsabstraktionen sind lokale Vereinbarung, also gilt es Bindungsbereiche, Gültigkeit und Verschattung zu beachten (wie bei `let` oder `where`).

Es existieren verschiedene Auswertungsstrategien für die Auswertung einer -Funktionsabstraktion auf Terme. Seien E_i Terme der Sorte s_i.

$$\star \quad ((s_1x_1, ..., s_nx_n)s : E)(E_1...E_n)$$

nun existieren abhängig davon ob die E_i Grundterme oder Terme mit freien Identifikatoren sind die folgenden beiden Auswertungsstrategien:

1. Wenn die E_i Grundterme sind, ist $\star$ durch $E[E_1/x_1, ..., E_n/x_n]$ ersetzbar. Dabei werden die Funktionsausdrücke nicht selbst ausgewertet sondern, nur die Funktionsaufrufe. Damit werden nur Berechnungsregeln für die Funktionsanwendung mit ausgewerteten Parametern angegeben (Werteaufruf, call by value). Die Grundterme entsprechen den Werten.

2. Auch wenn E_i freie Identifikatoren enthält, ist $\star$ durch $E[E_1/x_1, ..., E_n/x_n]$ ersetzbar. Das ist im allgemeinen kein Grundterm. Diese Methode wird als call by name (Namensaufruf) bezeichnet. Da hier im Gegensatz zu call by value die Ausdrücke in den Rumpf eingesetzt werden ohne sie vorher auszuwerten, müssen sie, falls der entsprechende Parameter mehrfach im Rumpf E vorkommt, auch mehrfach ausgewertet werden. Damit ist diese Methode unter Umständen ineffizienter. Allerdings terminiert sie öfter.

5.3.2 λ-Kalkül

Das λ-Kalkül ist eine allgemeine Theorie, die von ALONZO CHURCH in den Dreißigern entwickelt wurde. Es handelt sich dabei um die einfachste algorithmische Sprache in der man alle Berechnungen durchführen kann. Sämtliche Berechnungen werden auf die Definition von Funktionen zurückgeführt.

Sei $v \in V$ eine abzählbare Menge von Variablen und Λ die im folgenden induktiv definierte Termalgebra:

1. **Variable**: Wenn $v \in V$, so ist $v \in \Lambda$ ein Term.
2. **λ-Abstraktion**: Wenn F ein Term ist und v eine Variable, so ist $\lambda v.F \in \Lambda$ ein Term.
3. **Funktionsanwendung**: Sind F_1, F_2 Terme in Λ so ist auch $(F_1 F_2)$ ein Term in Λ.

Die Terme werden auch als **λ-Ausdrücke** bezeichnet.

Die λ-Abstraktion $\lambda v.t$ definiert eine Funktion mit dem Parameter v und dem Funktionsrumpf t. (FF') bedeutet die Anwendung der durch F gegebenen Funktion auf das Argument F'. F kann dabei sowohl eine λ-Abstraktion $\lambda v.t$ als auch ein beliebiger anderer Term sein.

Funktionen mit mehr als einem Parameter gibt man durch geschachtelte λ- Abstraktionen wieder. So besitzt eine zweistellige Funktion die Form $(\lambda v.(\lambda w.t))$.

In λ-Ausdrücken sind die Parameterbezeichnungen frei wählbar. Das heißt $\lambda x.t'$ ist dieselbe Funktion wie $\lambda y.t$, wenn y in T nicht frei vorkommt und $t' = t[y/x]$. Die konsistente Umbenennung von formalen Parametern wird auch als **α-Konversion** bezeichnet. Es gilt:

$$\lambda x.t \quad \underset{\alpha}{\longrightarrow} \quad \lambda y.t[y/x]$$

falls y nicht frei in t vorkommt (lokale Bindungen sind selbstverständlich erlaubt).

Die Regel der **β-Konversation** definiert die Anwendung eines λ-Ausdruckes auf Daten. Jeder Aufruf $((\lambda v.t)t')$ darf ersetzt werden durch $t[t'/v]$, falls in t' keine freie Variable vorkommt die in t gebunden ist.

$$((\lambda v.t)t') \quad \underset{\beta}{\longrightarrow} \quad t[t'/v]$$

Als Beispiel für die Anwendung der beiden Konversationen dient uns die Auswertung der folgenden Funktionen:

$$(\lambda x.(\lambda y.x + y)y)$$

Auf diese λ-Abstraktion ist die β-Konversation nicht anwendbar, y sowohl frei in der äußeren als auch gebunden in der inneren Funktion vorkommt. Daher muß zuerst die α-Konversion angewendet werden.

$$(\lambda x.(\lambda y.x + y)y) \quad \underset{\alpha}{\longrightarrow} \quad (\lambda x.(\lambda z.x + z)y) \quad \underset{\beta}{\longrightarrow} \quad \lambda z.y + z$$

Siehe auch: MANFRED BROY: **Informatik Eine grundlegende Einführung** Band 1; 2.Auflage; 4.1.6 Funktionsabstraktion; Seite 98ff und: GERHARD GOOS: **Vorlesungen über Informatik** Band 1; 2.Auflage; 5.1. Elementarer λ-Kalkül; Seite 210ff

Kapitel 6

Zuweisungsorientierte Programmierung

6.1 Rechnen als Zustandsveränderung

Im vorhergehenden Kapitel wurde die Berechnung von Funktionen durch Formulare dargestellt. Überträgt man dieses auf die Zuweisungsorientierte Programmierung ist:

- **Rechnen**: eine Eintragung in die "Kästchen" der Formulare, mit Wiederverwendung der Formulare.
- **Zustand**: der Inhalt der "Kästchen"

Die heute gebräuchlichen Rechnerarchitekturen besitzen einen linearen Speicher (Folge von Karos)damit geht die Formularstruktur natürlich verloren. Daher sind komplizierte Programme zur Ablaufsteuerung notwendig.

Um die Speicherplätze eindeutig zu kennzeichnen sind diese numeriert (Adresse). Diese Speicherplätze können gelesen, (gelöscht) und beschrieben werden, was zu einer Zustandsänderung führt. Benötigt wird nun eine Notation für diese Zustände (ohne die konkreten Speicherplätze direkt zu verwenden), die **Zustandsvariable**. Das sind Trippel aus:

- **Referenz (Verweis)**: eindeutige Kennzeichnung (entspricht der Adresse im Speicher)
- **Behälter (Schublade)**: eine Speicherstruktur zum Aufnehmen des jeweiligen "Wertes". Dieser Behälter hat eine feste Größe und dementsprechend gibt es Behälter für verschiedene Typen von Werten (Boolean, Integer, ...).
- **Wert**: die jeweils im Behälter vorhandene Größe oder die Angabe leer bzw. undefiniert (dafür wird auch das Bottomsymbol $\perp$ verwendet).

Die Grundoperationen auf diese Zustandsvariable sind:

- **lesen**: Feststellen des Inhaltes des Behälters ohne ihn zu ändern.
- **schreiben**: Ersetzen des momentanen Wertes durch einen neuen Wert, kurz Wertzuweisung.

Der Zugriff auf eine Variable erfolgt mittels einer **Zugriffsfunktion**, einer Funktion zur Berechnung der Referenz (Speicheradresse). In der Programmiersprache ist der Name der Variable die textuelle Repräsentation der Zugriffsfunktion. In der zuweisungsorientierten Programmierung ist dieser Name ein einfacher Identifikator.

Die Variablendeklaration legt den Typ (Behälter) und den Namen der Variablen fest. Als Beispiel dient die Variablendeklaration in JAVA:

```
int i;
int i = 27;    Initialisierung der Variable mit einem Wert
```

Die Wertzuweisung erfolgt mittels eines Zuweisungsoperators in PASCAL `:=` in JAVA und auch C hingegen bloß `=` (Vorsicht auf keinem Fall mit dem Gleichheitszeichen verwechseln oder gleichsetzen). Der Syntax einer Wertzuweisung unter JAVA ist:

$$\underbrace{\text{Referenzzugriff}}_{\substack{\text{Zugriffsfunktion}\\ \text{mit Behälter}}} \quad = \quad \underbrace{\text{Ausdruck}}_{\substack{\text{Vorschrift zur}\\ \text{Berechnung}\\ \text{des Wertes}}}$$

Ein Beispiel hierfür ist:

```
int i = 1;
i = i + 1;    (erhöhe i um eins)
```

Die Ausführung einer Zuweisung erfolgt in folgenden Schritten:

1. Ausrechnen des Ausdrucks mit dem momentanen Wert der Variable (1+1=2).

2. Löschen des alten Wertes.

3. Schreiben des neuen (in 1. bestimmten) Wertes in den Behälter.

6.2 Typen und Ausdrücke

Im folgenden sind die Standarddatentypen von JAVA kurz aufgeführt. JAVA ist eine streng typisierte Sprache. Das bedeutet das jede Variable einen deklarierten Typ haben muß. Allerdings beherrscht JAVA die automatische Typkonvertierung. So das sich Operationen ohne weiters zwischen verschiedenen Datentypen ausführen lassen.

Typ	**Größe**	**Bereich**
Ganzzahlen		
`int`	4 Byte	-2,147,483,648 bis 2,147,483,647
`byte`	1 Byte	-128 bis 127
`short`	2 Byte	-32768 bis 32767
`long`	8 Byte	-9,223,372,036,854,775,808 bis -9,223,372,036,854,775,807
Gleitkommazahlen		
`float`	4 Byte	etwa $+/-3.4 \times 10^{38}$ (6 bis 7 signifikante Stellen)
`double`	8 Byte	etwa $+/-1,79 \times 10^{308}$ (15 signifikante Stellen)
Zeichentypen		
`char`	2Byte	Unicode - Zeichen
Wahrheitswerte		
`boolean`	1 Byte	`true` oder `false`

Die meisten von JAVA verwendeten Operatoren sind in der nachfolgenden Tabelle aufgeführt. Sie sind zumeist linksassoziativ (bis auf die unären Operatoren und die Zuweisungsoperatoren). Boolesche Ausdrücke werden faul ausgewertet. Die Operatoren sind in der Tabelle nach Gruppen mit gleichem Operatorvorrang geordnet, dabei fällt der Rang innerhalb der Tabelle von oben nach unten. Innerhalb einer Gruppe haben die Operatoren den selben Rang.

Typ	**Assoz.**	**Operator**	**Erklärung**
unäre Operatoren	⟵	`!`	logisches NOT
		`~`	bitweises NOT
		`++`	Inkrement
		`--`	Dekrement
		`+`	unäres Plus
		`-`	unäres Minus (Negation)
		(Type)	Typkonvertierung
		`new`	Speicher für neues Objekt
Multiplikation	⟶	`*`	Multiplikation
		`/`	Division
		`%`	Modulo
Addition	⟶	`+`	binäres Plus
		`-`	binäres Minus
Schieben	⟶	`<<`	arithmetisches links Schieben
		`>>`	arithmetisches rechts Schieben
		`>>>`	logisches rechts Schieben
Vergleich	⟶	`<`	Kleiner
		`<=`	Kleiner gleich
		`>`	Größer
		`>=`	Größer gleich
Gleichheit	⟶	`==`	Gleich
		`!=`	Nicht Gleich
AND	⟶	`&`	bitweises AND
XOR	⟶	`^`	bitweises XOR
OR	⟶	`|`	bitweises OR
Logisches AND	⟶	`&&`	logisches AND
Logisches OR	⟶	`||`	logisches OR
Bedingung	⟵	`?:`	bedingter Ausdruck
Zuweisung	⟵	`=`	einfache Zuweisung
		`*=`	Zuweisung des Produkts
		`/=`	Zuweisung der Division
		`%=`	Zuweisung des Restes
		`+=`	Zuweisung der Summe
		`-=`	Zuweisung der Differenz
		`&=`	Zuweisung nach bitweisem AND
		`^=`	Zuweisung nach bitweisem XOR
		`|=`	Zuweisung nach bitweisem OR
		`<<=`	Zuweisung nach Linksschieben
		`>>=`	Zuweisung nach Rechtsschieben
		`>>>=`	Zuweisung nach Rechtsschieben

6.3 Ablaufsteuerung

6.3.1 Anweisungen

Zuweisungsorientierte Programme setzen sich aus Anweisungen zusammen. Die wichtigsten elementaren Anweisungen sind:

- die leere Anweisung, die nichts tut (in JAVA ein Semikoma)
- die Zuweisung

6.3.2 Zusammsammengesetzte Anweisungen

Sequentielle Komposition

Wie es die Bezeichnung schon andeutet werden hier einfach Anweisungen hintereinander gestellt. Sind *Anweisung1* und *Anweisung2* Anweisungen, so ist:

Anweisung1 ; *Anweisung2*

eine zusammengesetzte Anweisung mit folgender Bedeutung: zuerst wird *Anweisung1* ausgeführt und dann *Anweisung2*.

Bedingte Anweisungen

Seien *Anweisung1* und *Anweisung2* Anweisungen und *Bool* ein Boolescher Ausdruck, dann ist:

```
if Bool
   Anweisung1
else
   Anweisung2
```

eine bedingte Anweisung. Ist der Ausdruck *Bool* wahr so wird *Anweisung1* ausgeführt ist dieser Ausdruck hingegen falsch so wird die *Anweisung2* ausgeführt.

Es existieren im weiteren verschiedene Spielarten der bedingten Anweisung. So die bedingte Anweisung mit nur einem Zweig:

```
if Bool
   Anweisung1
```

Anweisung1 wird nur ausgeführt wenn der Ausdruck *Bool* wahr ist. Ist der Ausdruck *Bool* hingegen falsch so tut diese Anweisung nichts. Sie entspricht somit einer bedingten Anweisung in deren else - Zweig nur eine leere Anweisung steht.

Die Case - Anweisung erlaubt es einen Ausdruck nacheinander auf verschiedene Bedingungen zu testen und dementsprechend zu verzweigen. In JAVA hat man dafür die switch - Anweisung:

```
switch( Test_Ausdruck )
{
  case Konstante1:
    Anweisung1
    break;
  case Konstante2:
    Anweisung2
    break;

    ...

  default:
    Anweisung n
}
```

Die switch - Anweisung wertet zunächst den Ausdruck *Test_Ausdruck* aus und vergleicht ihn dann nacheinander mit den Konstanten. Stimmen die Werte dieses Ausdrucks und irgendeiner Konstante überein, so wird die dahinterstehende Anweisungsfolge ausgeführt. Um die switch Anweisung danach zu beenden, ist es nötig, als letzte Anweisung in dieser Folge ein break zu setzen. Fehlt die break - Anweisung so wird mit den Anweisungen hinter dem nächsten case fortgefahren. Falls der Ausdruck mit keiner der Konstanten übereinstimmt, wird die Anweisung hinter der default - Marke ausgeführt.

Wiederholungsanweisungen

Sei *Anweisung1* eine Anweisungen und *Bool* ein Boolescher Ausdruck, dann ist:

```
while( Bool )
  Anweisung1
```

eine Wiederholungsanweisung (Schleife). Der Ausdruck *Bool* wird als Bedingung bezeichnet und die Anweisung *Anweisung1* bildet den Rumpf der Wiederholungsanweisung. Die Wiederholungsanweisung wird ausgeführt, indem der Rumpf solange wiederholt ausgeführt wird bis die Bedingung den Wert `false` liefert. Liefert die Bedingung schon von Anfang an den Wert `false` so wird die Anweisung *Anweisung1* nicht ausgeführt. Daher spricht man von abweisender Wiederholung.

Eine weitere Spielart der Wiederholungsanweisungen ist die nichtabweisende Wiederholung:

```
do
  Anweisung1
while( Bool )
```

Hier wird der Rumpf stets einmal ausgeführt, bevor die Bedingung *Bool* überprüft wird. Der Rumpf wird solange wiederholt ausgeführt wird bis die Bedingung den Wert `false` liefert.

Die gezählte Wiederholung ist eine besondere Form der Wiederholungsanweisung die über den natürlichen Zahlen arbeitet. Dabei wird der Rumpf der Wiederholungsanweisung sooft ausgeführt wie zuvor durch eine Zähler mit einem festgesetzten Startwert S inkrementiert bzw. dekrementiert werden muß, um einen ebenfalls festgesetzten Zielwert Z zu erreichen. Sowohl S als auch Z müssen dabei aus dem Bereich der natürlichen Zahlen stammen. Der Rumpf wird dabei $S - Z + 1$

mal durchlaufen. JAVA weicht hier allerdings ab:

```
for ( I ; B ; S )
  Anweisung
```

In einer for - Schleife wird zunächst Ausdruck I ausgewertet um die Schleife zu initialisieren. Danach wird Ausdruck B als Testausdruck interpretiert. Der Schleifenrumpf also die Anweisung wird genau dann ausgeführt, wenn Ausdruck B den Wert `true` hat. Nach dem Ende des Schleifenrumpfes wird Ausdruck S ausgewertet und danach wieder mit Auswerten des Testausdrucks Ausdruck B fortgefahren.

Im folgenden sind zwei Beispiele für for - Schleifen unter JAVA abgedruckt:

1.Beispiel:

```
int x=5, y=0
for(int z=0; z<2; z++, y++);
x=x+y
```

$\Rightarrow$ y = 7

2.Beispiel:

```
int x=5, y=0
for(x=0; x<2; x++, y++)
   x=x+y;
```

$\Rightarrow$ y = 3

Im folgenden sei noch kurz aufgezeigt wie man for - Schleifen umformen kann (und unleserlichen Code produziert). Die folgenden Schleifen sind äquivalent:

```
for(I; B; S) A;
{I; for(; B; S) A;}
for(I; B;){ A; S;}
{I; for(; B;){ A; S;}}
{I; while(B){ A; S;}}
```

6.3.3 Blöcke

Zunächst gilt es zu definieren was ein Block ist. Ein Block umfaßt eine beliebige Folge von lokalen Variablendeklarationen und Anweisungen getrennt durch Semikola, die von geschweiften Klammern eingeschlossen sind.

Block: `{` *{ Variablendeklaration* `;` *| Anweisung }** `}`

Da ein Block selbst auch eine Anweisung ist, ist es möglich Blöcke zu verschachteln. Blöcke definieren den Bindungs- und Gültigkeitsbereich von Variablen, in dem die in ihnen lokal vereinbarten Variablen bis Blockende gültig sind. Allerdings ist es nicht möglich, identisch benannte Variablen in ineinander verschachtelten Blöcken neu zu definieren. Das folgende Beispiel läßt sich mit dem JDK nicht kompilieren:

```
{
   int n = 2;
   {
      int n = 3;
   }
}
```

Blöcke dienen der Zusammenfassung mehrerer Anweisungen für Schleifen und bedingte Anweisungen sowie der Strukturierung größerer Programme.

6.4 Funktionen in zuweisungsorientierten Sprachen

Grob gesagt sind Funktionen in zuweisungsorientierten Sprachen Blöcke mit einem Namen und einer Parameterliste. In diesem Block erfolgt die Berechnung des Funktionswertes aus den gegebenen Parametern. Eine Funktionsdefinition sieht (in JAVA) etwas vereinfacht etwa so aus:

Modifier Resultattyp Funktionsname(*Parameterliste*)

```
{
  ...
  return(...);
}
```

Funktionsrumpf

Zunächst wird der Typ des Rückgabewertes (der Resultattyp) der Funktion gefolgt vom Funktionsnamen (Bezeichner) angegeben. In der Parameterliste muß der Datentyp des Parameters gefolgt von seinen Namen eingegeben werden. Sollen mehrere Parameter an die Funktion übergeben werden sind diese durch Kommata zu trennen. Es ist ebenso möglich keine Parameter an die Funktion zu übergeben. Die Sichtbarkeit der Parameter beschränkt sich, wie bei den lokalen Variablen, auf den Funktionsrumpf. Am Schluß der Berechnung muß deren Ergebnis an die return - Anweisung übergeben werden. Return liefert diesen Wert, an die Stelle von der aus die Funktion aufgerufen wurde. Der Code einer Funktion, der hinter einer return - Anweisung steht wird nicht mehr ausgeführt. Es gilt aber zu beachten, daß die Berechnung des Funktionswert eventuell weitere Variablenwerte verändert (Seiteneffekte).

Der Aufruf einer Funktion erfolgt durch die Angabe des Funktionsnamens und der Übergabe der Parameter. Dazu werden in Klammern die gewünschten Parameter angegeben die in Zahl und Typ mit den im Funktionskopf definierten übereinstimmen müssen. Bei der Übergabe mehrerer Parameter sind diese durch Kommata zu trennen. Wichtig für das Verständnis ist es, daß der Name der Funktion in irgend einem Ausdruck stets für den gesamten Funktionsrumpf steht ("textuelle Ersetzung").

6.4.1 Prozeduren

Prozeduren unterscheiden sich von Funktionen darin das sie keinen Wert zurückgeben. Als Resultattyp wird der spezielle Typ `void` angegeben, der keine Werte besitzt. An die return - Anweisung wird dementsprechend kein Wert übergeben. Prozeduren haben dementsprechend nur Seiteneffekte.

In JAVA werden sowohl Funktionen und als auch Prozeduren als Methoden bezeichnet.

6.5 Felder, Arrays

Unter einem Feld bzw. Array versteht man einen zum abspeichern größerer Mengen gleichartiger Elemente (alle Elemente haben den selben Typ) geordneten Speicherbereich, mit direktem Zugriff aus jedes Feldelement (Komponente).

Für jeden Typ *tp* (der nicht Feldtyp ist) ist in JAVA *tp*[] der Typ der Felder mit Elementen aus *tp*. Zum Beispiel:

```
int[]
char[]
```

Ein Feld a vom Typ $tp[]$ ist eine Abbildung fa eines endlichen Anfangsstücks der Folge der natürlichen Zahlen (einschließlich Null) auf Variablen des Typs tp. Es handelt sich also vereinfacht gesagt um eine Art Tabelle.

0	1	...	$n-1$	n
fa_0	fa_1	...	fa_{n-1}	fa_n

In JAVA schreibt man `a[i]` statt $fa(i)$. Die `a[i]` sind Variablen vom Typ tp und fa ist die Referenzzugriffsfunktion für das Feld.

Die Deklaration von Variablen vom Typ $tp[]$ erfolgt wie üblich. Dabei können die eckigen Klammern sowohl hinter dem Typ als auch hinter der Variable stehen. Zum Beispiel:

```
int[] a;
int a[];
```

Nach der Deklaration des Arrays wurde zwar der Typ der Feldelemente festgelegt allerdings wurde noch keine Instanz (Exemplar) dieses Arrays im Speicher erzeugt. Dazu dient die Initialisierung des Arrays. Es existieren zwei Möglichkeiten der Initialisierung. Erstens über die Verwendung der Erzeugungsfunktion `new`:

```
int[] a = new int[3];
```

Mit dieser Anweisung wird ein Feld der Länge 3 erzeugt. Die einzelnen Feldelemente haben den Wert 0. Die zweite Möglichkeit besteht in der impliziten Initialisierung:

```
int[] a = {1,2,3};
```

Auch mit dieser Anweisung wird ein Feld der Länge 3 erzeugt. Die einzelnen Feldelemente bekommen explizit einen Wert zugewiesen.

Nun ein kleines Beispiel:

```
int[] x={1,2,3,4}, y={5,6,7,8}, z;
y[x[0]]=9;
(z=y)[(y=x)[0]+y[1]]=0;
```

1. In der zweiten Zeile unseres Beispiels wird der Variable `y[1]` der Wert 9 zugewiesen.
2. In der dritten Zeile wird zunächst x an y zugewiesen.
3. Damit ist `y[0] = 1` und `y[1] = 2`.
4. Somit ergibt sich `(z=y)[1+2]` und weiter `(z=y)[3]`
5. Nun wird das alte y an z zugewiesen.
6. Das ergibt für `z[3] = 0`.

Regel: beim Feldzugriff werden zuerst die Referenzausdrücke für die Zugriffsfunktion berechnet (von links nach rechts) und erst danach erfolgt die Zuweisung von Werten an das Feld bzw. von Feldern zueinander (ebenfalls von links nach rechts).

6.5.1 Mehrdimensionale Felder

tp[]...[] ist ein n - dimensionales Feld oder genauer ein eindimensionales Feld, dessen Komponenten (n-1) - dimensionale Felder sind. Die Deklaration mehrdimensionaler Felder erfolgt analog zur Initialisierung eindimensionaler Felder. Dabei wird für jede Dimension ein Klammerpaar angegeben. Auch die Initialisierung erfolgt analog. Beispiel:

```
int matrix[][] = new int[5][6];
```

Hier wird ein Feld mit 5 Komponenten erzeugt. Die Komponenten sind vom Typ `int[]` und haben ihrerseits die Länge 6.

Eine Besonderheit die erst bei mehrdimensionalen Feldern zum tragen kommt ist die partielle Initialisierung. Ein Beispiel:

```
int matrix[][] = new int[3][];
for(int i=0; i<3; i++)
  matrix[i] = new int[i+1];
   for(int j=0; j<i+1; j++)
    matrix[i][j]=j;
```

Zunächst wird ein Feld von 3 Komponenten erzeugt. Dann werden die untergeordneten Felder erzeugt. Deren erstes Feld hat die Länge 1, das zweite Feld hat die Länge 2 und das dritte die Länge 3. Bei der partiellen Initialisierung gilt es zu beachten das sie stets von links nach rechts erfolgen muß.

6.5.2 Die length - Funktion

JAVA stellt die spezielle Funktion `length` zur Verfügung, um die Länge eines Feldes bestimmen zu können.

```
a.length
```

Dieser Ausdruck liefert die Länge des Feldes `a`. Zur Veranschaulichung der Arbeitsweise der Funktion soll die folgende Skizze dienen:

e	.	.	.

Der Wert e gibt die Länge des Feldes an und kann nur mit `length` ausgelesen werden.

6.6 Sprungbefehle

Zwar unterstützt JAVA keine goto's, aber es verfügt über eingeschränkte Sprünge aus Schleifen heraus. Der Befehl `break` führt dazu das die Schleife in der er steht verlassen und mit der Ausführung des Codes hinter der Schleife fortgefahren wird.

Da es stellenweise jedoch notwendig ist aus einer verschachtelten Struktur von Schleifen zu springen, verfügt JAVA über das benannte `break`. Mit diesem ist es möglich aus mehrfach verschachtelten Schleifen heraus zu springen. Die Zeilenmarke muß vor der äußersten Schleife stehen, aus der man springen möchte.

Mittels `continue` ist es möglich innerhalb einer Schleife an das Ende des Schleifenrumpfes zu springen.

Siehe auch: Manfred Broy: **Informatik Eine grundlegende Einführung** Band 1; 2.Auflage; 5. Zuweisungsorientierte Programmierung; Seite 155ff

Kapitel 7

Der Zusicherungskalkül

7.1 Einführendes Beispiel

Als Beispiel dient uns eine JAVA Methode zur Berechnung des kleinsten gemeinsamen Vielfachen (kgV):

```
int A = a;
int B = b;
int C = c;
```

a,b,c sind beliebige natürliche Zahlen.

```
while((A!=B) || (B!=C)
{
   if(A<B)
     A=A+a;
   else if(B<C)
     B=B+b;
   else
     C=C+c;
}
```

Die auftretenden Zusicherungen sind Aussagen über den Zustand des Programmes, an einer bestimmten Stelle (an der ich als Leser gerade bin, oder an der der Computer gerade wäre). Die Zusicherung ist an die Stelle zu schreiben, die gerade betrachtet wird.

Der Gesamtzustand des Speichers ist die Menge aller Variablen. Also ist der Zustand eine logische Aussage über die Werte der Programmvariablen. (Eigentlich sind Programme mit Zusicherungen zu Entwickeln und die Zusicherungen als Kommentare einzufügen.)

Zu Begin unseres Beispiels müßte die Zusicherung P_0 stehen:

P_0: $\quad \mathtt{a} \geq 0 \wedge \mathtt{b} \geq 0 \wedge \mathtt{c} \geq 0$

Vor der while - Schleife wäre die Zusicherung P einzufügen:

P: `A,B,C` sind Vielfache von `a,b,c`
und `A,B,C` $\leq$ kgv(`a,b,c`)

Im Rumpf der while - Schleife zu Beginn des Schleifenrumpfes:

$P_0(z) \wedge P(z) \wedge \underbrace{\text{Bedingung}}_{\text{Schleifenbedingung}}$ $\Rightarrow$ wobei z der Zustand des Speichers ist

Für jeden Zustand z in der Schleife gilt:

$P_0(z) \wedge P(z)$ $\Leftarrow$ Schleifeninvariante

Nach Verlassen der Schleife gilt:

$P_0(z) \wedge P(z) \wedge \neg(\text{Bedingung})$ $\Rightarrow$ hieraus folgt: `A == B == C ==` kgv(`a,b,c`)

7.2 Allgemeines zum Zusicherungskalkül

Das Zusicherungskalkül ist eine Hilfe für die Konstruktion beweisbar korrekter Programme und dient der Definition der exakten Bedeutung von Konstrukten (Anweisungen, ...) in Programmiersprachen. Die Grundidee, die dem zugrunde liegt, ist die Beschreibung von Programmzuständen. Im wesentlichen sind das:

- **Speicherzustand**
- **Zustandsraum**: Definiert als Menge aller möglichen Belegungen (Werte aller Variablen).
- **Zustand**: Eine beliebige Belegung einer Variable.

Wird die Anweisung A im Zustand z ausgeführt so ergibt das den Folgezustand z' (Wirkung von A). Im allgemeinen ist es aber nicht nötig detailierte Aussagen über alle Variable (alle Werte) zu machen. Daher sollten nur Aussagen über die relevanten Werte gemacht werden. Dafür bedient man sich der Prädikatenlogik (siehe nächstes Kapitel).

$\neg, \vee, \wedge, ...$	boolesche Operatoren
$\exists x ...$	Es exixtiert (mindestens) ein x mit ...
$\forall x ...$	Für alle x gilt ...

Das Prädikat (logische Aussage) entspricht dabei der Zusicherung. $P(z)$ ist das Prädikat P im Abhängigkeit vom Zustand z.

Fragestellungen beim Zusicherungskalkül:

1. Zustandsverfolgung:
   (a) Gelte im Anfangszustand z_0 die Zusicherung $P(z_0)$. Sei z_n der Zustand nach n Schritten (Ausführung einer elementaren Anweisung). Was für ein Prädikat Q gibt es, für das es ein $Q(z_n)$ gibt?
   (b) Wenn am Ende des Programmablaufes $Q(z_n)$ gelten soll, welches P muß dann zu Begin (in z_0) gelten.
2. Eine Aufgabe sei beschrieben durch das Paar (P, Q) von Zusicherungen (Spezifikation), die besagt, daß P zu Beginn und Q nach der Terminierung gelten soll. Gesucht ist also ein Programm A und ein Beweis, daß wenn $P(z_0)$ gilt, A ausgeführt wird und dann $Q(z_e)$ gilt. Hier muß man allerdings zwischen zwei Fällen unterscheiden:

(a) **partielle Korrektheit**: liegt vor wenn im Zustand z_0 $P(z_0)$ gilt, und nach der Ausführung von A, $Q(z_0)$ gilt: $P\{A\}Q$

(b) **totale Korrektheit**: liegt vor wenn im Zustand z_0 $P(z_0)$ gilt, A terminiert und dann $Q(z_0)$ gilt: $\{P\}A\{Q\}$

Im Fall unseres kgV - Beispiels liegt totale Korrektheit vor, da `A,B,C` stets kleiner oder gleich kgV(`a,b,c`) sind und `A,B,C` nur vergrößert werden.

7.3 Zusicherungskalkül nach Floyd und Hoare

Der Nachweis das bestimmte Zusicherungen für ein Programm gelten wird als Programmverifikation bezeichnet. Sie entspricht einem mathematischen Beweis, einer Ableitung. Für die Durchführung der Beweisschritte werden spezielle Ableitungsregeln verwendet, die Floyd - Hoare - Trippel. Für eine Reihe elementarer Anweisungen sind sie im folgenden angegeben.

1. Leere Anweisung: Da sie nichts ändert gelten nach ihren Ausführen die gleichen Zusicherungen.

   $$\{R\};\{R\}$$

2. Zuweisung: Sei `v` der Bezeichner einer Variable und a der Wert der diesem zugewiesen werden soll.

   $$\{\ \underbrace{\mathsf{zulässig}(\mathtt{v})}_{\text{Referenzzugriff muß berechenbar sein}} \ \land \ \underbrace{\mathsf{zulässig}(a)}_{\text{Der Wert } (a) \text{ muß berechenbar sein}} \ \land \ \underbrace{Q[a/\mathtt{v}]}_{\text{prüfen was passiert wenn } \mathtt{v} \text{ denn Wert } a \text{ zugewiesen bekommt}}\ \}\mathtt{v} = a\{Q\}$$

3. Sequentielle Komposition: Die Nachbedingung der zuerst auszuführenden Anweisung wird zur Vorbedingung der anschließend auszuführenden Anweisung.

   $\{P\}A_1 \bullet A_2\{R\}$ gilt genau dann, wenn:
   $\exists Q(\{P\}A_1\{Q\} \land \{Q\}A_2\{R\})$

4. Bedingte Anweisung: Hier wird sowohl der Fall beachtet das die Bedingung gilt als auch das sie nicht gilt. Die Bedingungen der bedingten Anweisung werden in Zusicherungen umgewandelt.

   Aus $\{b \land P\}A_1\{Q\}$ und $\{\neg b \land P\}A_2\{Q\}$ folgt:
   $\{P\}$ `if`(b) A_1 `else` $A_2\{Q\}$

5. Wiederholungsanweisung: A wird nur solange Ausgeführt wie die Bedingung b gilt, dementsprechend gilt nach der Terminierung der Schleife $\neg b$. Die Zusicherung P gilt nach jedem Durchlauf der Schleife. Daher wird P als Invariante der Wiederholungsanweisung bezeichnet.

   Aus $\{P \land b\}A\{P\}$ folgt:
   P $\{$`while`(b) A $\}P \land \neg b$

Allerdings wird damit nur Ausgesagt das P die Schleifeninvariante ist, die Terminierung muß zusätzlich bewiesen werden. Erst dann gilt:

$$\{P\}\ \texttt{while}(b)\ A\ \{Q\}$$

Siehe auch: Manfred Broy: **Informatik Eine grundlegende Einführung** Band 1; 2.Auflage; 5.7 Programmiertechniken für Zuweisungen; Seite 174ff

Index