

3 Terme und Algebren

3.1 Terme

In allen formalen Kalkülen benutzt man **Formeln als Ausdrucksmittel**.
Hier betrachten wir **nur ihre Struktur - nicht ihre Bedeutung**. Wir nennen sie **Terme**.

Terme bestehen aus **Operationen, Operanden, Konstanten und Variablen**:

$a + 5$ blau ? gelb = grün ♥ > ♦

Terme werden nicht „ausgerechnet“.

Operationen, Konstanten und Variablen werden als **Symbole ohne Bedeutung** betrachtet.

Notation von Termen:

Infix-, Postfix-, Präfix- und Baum-Form

Umformung von Termen:

Grundlage für die Anwendung von Rechenregeln, Gesetzen

Für **Variable** in Termen werden Terme **substituiert**:

in $a + a = 2*a$ substituiere a durch $3*b$ $3*b + 3*b = 2*3*b$

Unifikation: Terme durch Substitution von Variablen gleich machen,
z. B. um die Anwendbarkeit von Rechenregeln zu prüfen

Sorten und Signaturen

Terme werden zu einer **Signatur** gebildet.

Sie legt die verwendbaren Symbole und die Strukturierung der Terme fest.

Signatur $\Sigma := (S, F)$, S ist eine Menge von **Sorten**, F ist eine Menge von **Operationen**.

Eine **Sorte** $s \in S$ ist ein **Name für eine Menge von Termen**, z. B. ARITH, BOOL;
verschiedene Namen benennen disjunkte Mengen

Eine **Operation** $f \in F$ ist ein **Operatorsymbol**, beschrieben durch
Anzahl der Operanden (**Stelligkeit**),
Sorten der Operanden und **Sorte des Ergebnisses**

0-stellige Operatoren sind Konstante, z. B. true, 1

Beispiele:

einzelne Operatoren:

Name Operandensorten Ergebnissorte

+	ARITH x ARITH	-> ARITH
<	ARITH x ARITH	-> BOOL
^	BOOL x BOOL	-> BOOL
true:		-> BOOL
1:		-> ARITH

Signatur $\Sigma_{\text{BOOL}} := (S_{\text{BOOL}}, F_{\text{BOOL}})$

$S_{\text{BOOL}} := \{ \text{BOOL} \},$

$F_{\text{BOOL}} :=$

{ true:		-> BOOL,
false:		-> BOOL,
^:	BOOL x BOOL	-> BOOL,
¬:	BOOL	-> BOOL
}		

Korrekte Terme

In **korrekten Termen** muss jeweils die Zahl der Operanden mit der **Stelligkeit** der Operation und die **Sorten** der Operandenterme mit den Operandensorten der Operation übereinstimmen.

Induktive Definition der **Menge τ der korrekten Terme der Sorte s zur Signatur $\Sigma = (S, F)$:**

Sei die Signatur $\Sigma = (S, F)$. Dann ist t ein **korrekter Term der Sorte $s \in S$** , wenn gilt

- $t = v$ und v ist der **Name einer Variablen** der Sorte s , oder
- $t = f(t_1, t_2, \dots, t_n)$, also die **Anwendung einer n -stelligen Operation**

$$f: s_1 \times s_2 \times \dots \times s_n \rightarrow s \in F$$

wobei jedes t_i ein **korrekter Term der Sorte s_i** ist

mit $n \geq 0$ (einschließlich Konstante f bei $n = 0$) und $i \in \{1, \dots, n\}$

$f(t_1, \dots, t_n)$ ist ein **n -stelliger Term**; die t_i sind seine **Unterterme**.

Korrekte Terme, die **keine Variablen** enthalten, heißen **Grundterme**.

Beispiele: korrekte Terme zur Signatur Σ_{BOOL} :

 false \neg true true \wedge x \neg (a \wedge b) x \wedge \neg y

nicht korrekt: a \neg b \neg (\wedge b)

Notationen für Terme

Notation eines n -stelligen Terms mit Operation (Operator) f und Untertermen t_1, t_2, \dots, t_n :

Bezeichnung	Notation	Beispiele
Funktionsform:	Operator vor der geklammerten Folge seiner Operanden $f(t_1, t_2, \dots, t_n)$	$\wedge (< (0, a), \neg (< (a, 10)))$
Präfixform:	Operator vor seinen Operanden $f t_1 t_2 \dots t_n$	$\wedge < 0 a \neg < a 10$
Postfixform:	Operator nach seinen Operanden $t_1 t_2 \dots t_n f$	$0 a < a 10 < \neg \wedge$
Infixform	2-stelliger Operator zwischen seinen (beiden) Operanden $t_1 f t_2$	$0 < a \wedge \neg a < 10$

Die **Reihenfolge der Operanden** ist in allen vier Notationen **gleich**.

Präzedenzen und Klammern für Infixform

Die **Infixform** benötigt **Klammern** oder **Präzedenzen**, um Operanden an ihren Operator zu binden: Ist in $x + 3 * y$ die 3 rechter Operand des + oder linker Operand des * ?

Klammern beeinflussen die Struktur von Termen in der Infixform:

z. B. $(x + 3) * y$ oder $x + (3 * y)$

Redundante Klammern sind zulässig.

Ein Term ist **vollständig geklammert**, wenn er und jeder seiner Unterterme geklammert ist:

z. B. $((x) + ((3) * (y)))$

Für die **Infixform** können den Operatoren unterschiedliche **Bindungsstärken (Präzedenzen)** zugeordnet werden, z. B. bindet * seine Operanden vereinbarungsgemäß stärker an sich als +, d. h. * hat **höhere Präzedenz** als +.

Damit sind $x + 3 * y$ und $x + (3 * y)$ verschiedene Schreibweisen für denselben Term.

Für **aufeinanderfolgende Operatoren gleicher Präzedenz** muss geregelt werden, ob sie ihre Operanden **links-assoziativ** oder **rechts-assoziativ** binden:

links-assoziativ: $x + 3 + y$ steht für $(x + 3) + y$

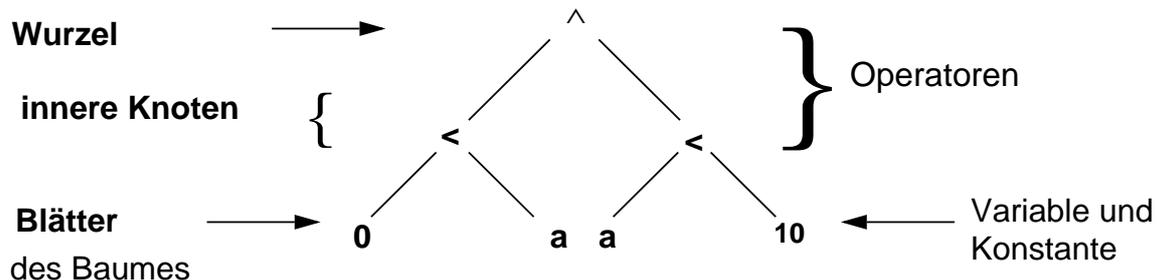
rechts-assoziativ: $x ** 3 ** y$ steht für $x ** (3 ** y)$

Funktionsform, Präfixform, Postfixform benötigen weder Regeln für Präzedenz oder Assoziativität noch zusätzliche Klammern!

Terme als Bäume

Terme kann man als Bäume darstellen (**Kantorowitsch-Bäume**):

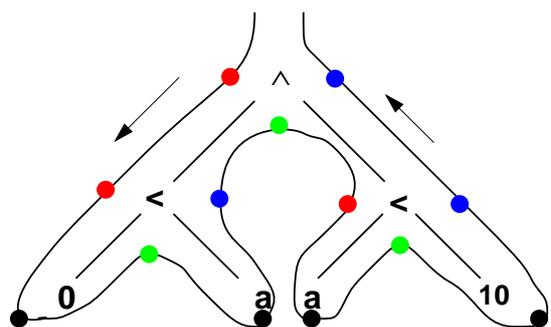
Term: $0 < a \wedge a < 10$



Aus einem Durchlauf des Baumes in Pfeilrichtung erzeugt man

- **Präfixform**, wenn man beim **ersten Besuch**
- **Postfixform**, wenn man beim **letzten Besuch**
- **Infixform**, wenn man beim **vorletzten Besuch** (bei **2-stelligen Operatoren**)

den Operator aufschreibt.



Substitution und Unifikation

Eine **Substitution** beschreibt, wie in einem Term vorkommende **Variablen durch Terme ersetzt** werden.

Eine **einfache Substitution** $\sigma = [v / t]$ ist ein Paar aus einer Variablen v und einem Term t zur Signatur Σ . v und t müssen **dieselbe Sorte** s haben.

Beispiel: $\sigma = [x / 2*b]$

Die **Anwendung einer Substitution** σ auf einen Term u schreibt man $u \sigma$, z. B. $(x+1) [x / 2*b]$.

Die **Anwendung einer einfachen Substitution** $u \sigma$ mit $\sigma = [v / t]$, ist **definiert** durch

- $u [v / t] = t$, falls u die zu ersetzende Variable v ist,
- $u [v / t] = u$, falls $u \neq v$ und u eine Konstante oder eine andere Variable ist,
- $u [v / t] = f(u_1 [v / t], u_2 [v / t], \dots, u_n [v / t])$, falls $u = f(u_1, u_2, \dots, u_n)$

D. h. in u werden **alle Vorkommen der Variablen v gleichzeitig durch den Term t ersetzt**.

Kommt v auch in t vor, so wird es nicht nochmals ersetzt!

Beispiele: $(x + 1) [x / 2*b] = (2*b + 1)$

$$(x - x) [x / 3] = (3 - 3)$$

$$(x + y) [y / y*y] = (x + y*y)$$

Mehrfache Substitution

In einer **mehrfachen Substitution** $\sigma = [v_1 / t_1, \dots, v_n / t_n]$ müssen alle Variablen v_i paarweise verschieden sein. In jedem v_i / t_i müssen v_i und t_i jeweils derselben Sorte s_i angehören.

σ wird dann auf einen Term u wie folgt angewandt:

- $u \sigma = t_i$, falls $u = v_i$ für ein $i \in \{1, \dots, n\}$,
- $u \sigma = u$, falls u eine Konstante ist oder eine Variable, die nicht unter v_i für ein $i \in \{1, \dots, n\}$ vorkommt,
- $u \sigma = f(u_1 \sigma, u_2 \sigma, \dots, u_n \sigma)$, falls $u = f(u_1, u_2, \dots, u_n)$

D. h. σ ist die gleichzeitige Substitution aller Vorkommen jeder Variablen v_i jeweils durch den Term t_i .

Beispiele: $\sigma = [x / 2*b, y / 3]$

$$(x + y) \sigma = (2*b + 3)$$

$$(y + a*y) \sigma = (3 + a*3)$$

$$(x * y) [x / y, y / y*y] = (y * (y * y))$$

Die **leere Substitution** wird $[\]$ notiert. Für alle Terme t gilt $t [\] = t$.

Außerdem gilt $[v / v] = [\]$ für jede Variable v .

Hintereinanderausführung von Substitutionen

Auf einen Term können **mehrere Substitutionen hintereinander** ausgeführt werden,

$$\text{z. B. } u \sigma_1 \sigma_2 \sigma_3 = ((u \sigma_1) \sigma_2) \sigma_3$$

$$(x+y) [x/y*x] [y/3] [x/a] = (y*x+y) [y/3] [x/a] = (3*x+3) [x/a] = (3*a+3)$$

Mehrere **Substitutionen hintereinander** können als **eine Substitution** angesehen werden:

$$\text{z. B. } u \sigma_1 \sigma_2 \sigma_3 = u (\sigma_1 \sigma_2 \sigma_3) = u \sigma$$

Mehrere **einfache Substitutionen hintereinander** kann man **in eine mehrfache Substitution** mit gleicher Wirkung umrechnen:

$$\text{Die Hintereinanderausführung } [x_1 / t_1, \dots, x_n / t_n] [y / r]$$

hat auf jeden Term die gleiche Wirkung wie

$$\text{falls } y \text{ unter den } x_i \text{ vorkommt } [x_1 / (t_1 [y / r]), \dots, x_n / (t_n [y / r])]$$

$$\text{falls } y \text{ nicht unter den } x_i \text{ vorkommt } [x_1 / (t_1 [y / r]), \dots, x_n / (t_n [y / r]), y / r]$$

$$\text{Beispiel: } [x / y*x] [y / 3] [x / a] = [x / 3*x, y / 3] [x / a] = [x / 3*a, y / 3]$$

Umfassende Terme

Rechenregeln werden mit **allgemeineren Termen** formuliert, die auf **speziellere Terme** angewandt werden,

$$\begin{array}{l} \text{z. B. Distributivgesetz:} \\ \text{angewandt auf} \end{array} \quad \begin{array}{l} a * (b + c) \\ 2 * (3 + 4*x) \end{array} \quad = \quad \begin{array}{l} a * b + a * c \\ 2 * 3 + 2 * 4*x \end{array}$$

Ein **Term s umfasst einen Term t**, wenn es eine Substitution σ gibt, die s in t umformt: $s \sigma = t$

s umfasst t, ist eine **Quasiordnung**, d. h. die Relation **umfasst** ist

$$\text{transitiv: } \text{sei } r \sigma_1 = s, s \sigma_2 = t, \text{ dann ist } r (\sigma_1 \sigma_2) = t$$

$$\text{reflexiv: } t [] = t, \text{ mit der leeren Substitution } []$$

Eine **Halbordnung ist umfasst nicht**, weil

$$\begin{array}{l} \text{nicht antisymmetrisch:} \\ \text{Terme, die sich nur in den Variablennamen unterscheiden,} \\ \text{kann man ineinander umformen, z. B.} \\ 2*x [x / y] = 2*y \text{ und } 2*y [y / x] = 2*x \end{array}$$

Deshalb gilt zwar der allgemeinere Term $a * (b + c)$ umfasst den spezielleren $2 * (3 + 4*x)$, aber nicht immer ist ein Term s allgemeiner als ein Term t, wenn s umfasst t: $2*x$ und $2*y$

Unifikation

Die **Unifikation** substituiert zwei Terme, sodass sie gleich werden.

Zwei Terme s und t sind unifizierbar, wenn es eine Substitution σ gibt mit $s \sigma = t \sigma$. σ heißt Unifikator von s und t.

Beispiel: Terme: $s = (x + y)$ $t = (2 + z)$
 Unifikatoren: $\sigma_1 = [x / 2, y / z]$ $\sigma_2 = [x / 2, z / y]$,
 $\sigma_3 = [x / 2, y / 1, z / 1]$ $\sigma_4 = [x / 2, y / 2, z / 2] \dots$

Ist σ ein **Unifikator** von s und t und τ eine **Substitution**, dann ist auch die Hintereinanderausführung $\sigma \tau = \sigma'$ auch ein Unifikator von s und t.

Ein **Unifikator σ heißt allgemeinsten Unifikator** der Terme s und t, wenn es zu allen anderen Unifikatoren σ' eine Substitution τ gibt mit $\sigma \tau = \sigma'$.

Im Beispiel sind σ_1 und σ_2 allgemeinste Unifikatoren, z. B. $\sigma_1 [z / 1] = \sigma_3$

Es kann **mehrere allgemeinste Unifikatoren** geben. Sie können durch **Umbenennen von Variablen** ineinander überführt werden, z. B.

$$\sigma_1 [z / y] = [x / 2, y / z] [z / y] = [x / 2, y / y, z / y] = [x / 2, z / y] = \sigma_2$$

Unifikationsverfahren

Unifikation zweier Terme s und t nach Robinson:

Seien s und t Terme in **Funktionsschreibweise**.

Dann ist das **Abweichungspaar $A(s, t) = (u, v)$** das erste Paar unterschiedlicher, korrespondierender Unterterme u und v, das man beim Lesen von links nach rechts antrifft.

Algorithmus:

1. Setze $\sigma = []$ (leere Substitution)
2. Solange es ein Abweichungspaar $A(s \sigma, t \sigma) = (u, v)$ gibt wiederhole:
 - a. ist **u eine Variable x**, die in v nicht vorkommt, dann ersetze σ durch $\sigma [x / v]$, oder
 - b. ist **v eine Variable x**, die in u nicht vorkommt, dann ersetze σ durch $\sigma [x / u]$,
 - c. **sonst** sind die Terme s und t **nicht unifizierbar; Abbruch** des Algorithmus.
3. Bei Erfolg gilt $s \sigma = t \sigma$ und σ **ist allgemeinsten Unifikator**.

Beachte, dass bei jeder Iteration die bisherige Substitution auf die vollständigen Terme s, t angewandt wird.

Beispiel für Unifikationsverfahren

Unifikation zweier Terme s und t nach Robinson:

$$\begin{aligned} s &= + (* (2, x), 3) \\ t &= + (z, x) \end{aligned}$$

$$\sigma = []$$

Schritt \downarrow Abweichungspaar

$$\begin{array}{ll} 1 & \begin{array}{l} s \sigma = + (* (2, x), 3) \\ t \sigma = + (z, x) \end{array} \quad \text{Fall 2b:} \quad \sigma = [] [z / * (2, x)] \end{array}$$

$$\begin{array}{ll} 2 & \begin{array}{l} s \sigma = + (* (2, x), 3) \\ t \sigma = + (* (2, x), x) \end{array} \quad \text{Fall 2b:} \quad \sigma = [] [z / * (2, x)] [x / 3] \end{array}$$

$$\begin{array}{ll} 3 & \begin{array}{l} s \sigma = + (* (2, 3), 3) \\ t \sigma = + (* (2, 3), 3) \end{array} \quad \text{allgemeinster Unifikator: } \sigma = \begin{array}{l} [z / * (2, x)] [x / 3] = \\ [z / * (2, 3), x / 3] \end{array} \end{array}$$

3.2 Algebren

Eine **Algebra** ist eine **formale Struktur**, definiert durch eine **Trägermenge**, **Operationen** darauf und **Gesetze** zu den Operationen.

In der Modellierung der Informatik spezifiziert man mit Algebren

Eigenschaften veränderlicher Datenstrukturen und dynamische Systeme,
z. B. Datenstruktur *Keller* oder die Bedienung eines Getränkeautomaten.

Wir unterscheiden 2 Ebenen: **abstrakte Algebra** und **konkrete Algebra**:

Eine **abstrakte Algebra** spezifiziert Eigenschaften **abstrakter Operationen**,
definiert nur durch eine **Signatur** - Realisierung durch Funktionen bleibt absichtlich offen

Trägermenge: korrekte Terme zu der Signatur

Gesetze erlauben, Vorkommen von Termen durch andere Terme zu ersetzen
z. B. $\neg \text{false} \rightarrow \text{true}$ $\text{pop}(\text{push}(k, t)) \rightarrow k$

Eine **konkrete Algebra** zu einer abstrakten Algebra

definiert **konkrete Funktionen** zu den Operationen der Signatur,
so dass die Gesetze in **Gleichungen zwischen den Funktionstermen** übergehen.

Sie beschreibt so eine **Implementierung** der spezifizierten Datenstruktur,
bzw. des Systems

Abstrakte Algebra

Eine **abstrakte Algebra** $A = (\tau, \Sigma, Q)$ ist definiert durch die Menge korrekter Terme τ zur **Signatur** Σ und eine **Menge von Axiomen (Gesetzen) Q**.

Axiome haben die Form $t_1 \rightarrow t_2$, wobei t_1, t_2 , **korrekte Terme gleicher Sorte** sind, die **Variablen** enthalten können. Die Algebra definiert, wie man Terme **mit den Axiomen in andere Terme umformen** kann.

Mit Axiomen umformen heißt: Unter Anwenden eines Axioms $t_1 \rightarrow t_2$ kann man einen Term s_1 in einen Term s_2 umformen. Wir schreiben $s_1 \rightarrow s_2$, wenn gilt:

- s_1 und s_2 stimmen in ihren „äußeren“ Strukturen überein und unterscheiden sich nur durch die Unterterme r_1 und r_2 an entsprechenden Positionen in s_1 und s_2 , und
- es gibt eine Substitution σ , sodass gilt $t_1 \sigma = r_1$ und $t_2 \sigma = r_2$

$$\begin{array}{ccccccc}
 \text{Terme} & s_1 = & \dots\dots r_1 \dots\dots & \rightarrow & \dots\dots r_2 \dots\dots & = & s_2 \\
 & & \parallel & & \parallel & & \\
 & & t_1 \sigma & & t_2 \sigma & & \\
 \text{Axiom} & & t_1 & \rightarrow & t_2 & &
 \end{array}$$

s ist in t umformbar, wenn es eine endliche Folge von Termen $s = s_0, s_1, \dots, s_n = t$ mit $s_{i-1} \rightarrow s_i$ gibt; wir schreiben dann $s \rightarrow t$.

„ \rightarrow “ ist **transitiv**. Wenn es auch **irreflexiv** ist (so sollten die Axiome gewählt werden), ist es eine **strenge Halbordnung**.

Beispiel: abstrakte Algebra Bool

Signatur $\Sigma = (\{\text{BOOL}\}, F)$

Operationen F:

true: \rightarrow BOOL
 false: \rightarrow BOOL
 \wedge : BOOL x BOOL \rightarrow BOOL
 \vee : BOOL x BOOL \rightarrow BOOL
 \neg : BOOL \rightarrow BOOL

Axiome Q: für alle x, y der Sorte BOOL gilt

Q₁: $\neg \text{true} \rightarrow \text{false}$
 Q₂: $\neg \text{false} \rightarrow \text{true}$
 Q₃: $\text{true} \wedge x \rightarrow x$
 Q₄: $\text{false} \wedge x \rightarrow \text{false}$
 Q₅: $x \vee y \rightarrow \neg (\neg x \wedge \neg y)$

Die Axiome sind geeignet, alle korrekten Terme ohne Variablen in in einen der beiden Terme **true** oder **false** umzuformen.

true und **false** heißen **Normalformen** (siehe Folie 3.20).

Konkrete Algebra

Zu einer abstrakten Algebra $A_a = (\tau, (S, F), Q)$, kann man

konkrete Algebren wie $A_k = (W_k, F_k, Q)$

angeben, wobei

W_k eine **Menge von Wertebereichen** ist, je einer **für jede Sorte** aus S ,

F_k eine **Menge von Funktionen** ist, je eine **für jede Operation** aus F .

Die Definitions- und Bildbereiche der Funktionen müssen konsistent den Sorten der Operationen zugeordnet werden.

Den **Axiomen Q** müssen **Gleichungen zwischen den Funktionstermen** in den Wertebereichen entsprechen.

Es können in der konkreten Algebra noch weitere Gleichungen gelten.

Eine konkrete Algebra heißt auch **Modell der abstrakten Algebra**.

Beispiel für eine konkrete Algebra

Beispiel: eine konkrete Algebra FSet zur abstrakten Algebra Bool:

konkrete Algebra FSet

abstrakte Algebra Bool

$W_k:$ $\{\emptyset, \{1\}\}$

Sorte BOOL

$F_k:$ $\{1\}$

true

\emptyset

false

Mengendurchschnitt \cap

\wedge

Mengenvereinigung \cup

\vee

Mengenkomplement bezüglich $\{1\}$

\neg

Axiome Q:

Man kann zeigen, dass die Axiome Gleichungen zwischen den Termen in W_k entsprechen:

z. B. $\emptyset \cap x = \emptyset$ entspricht

$\text{false} \wedge x \rightarrow \text{false}$

Die boolesche Algebra mit den üblichen logischen Funktionen ist natürlich auch eine konkrete Algebra zur abstrakten Algebra Bool.

Beispiel 2.2: Datenstruktur Keller

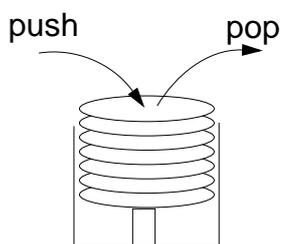
Die Eigenschaften einer **Datenstruktur Keller** beschreiben wir zunächst informell. Folgende **Operationen** kann man mit einem Keller ausführen:

create Stack:	liefert einen leeren Keller
push:	fügt ein Element in den Keller ein
pop:	entfernt das zuletzt eingefügte Element
top:	liefert das zuletzt eingefügte und nicht wieder entfernte Element
empty:	gibt an, ob der Keller leer ist.

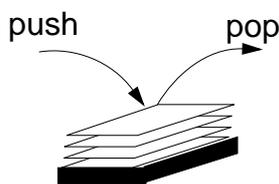
Die Eigenschaften der Datenstruktur Keller sollen präzise durch eine abstrakte Algebra spezifiziert werden.

Beispiele

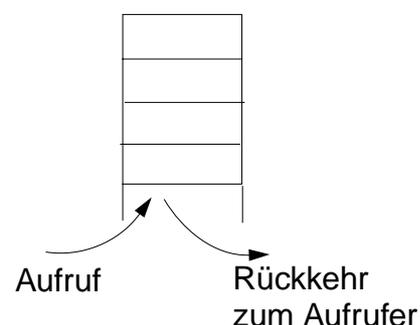
Tellerstapel



Aktenstapel



Laufzeitkeller



Beispiel: Abstrakte Algebra spezifiziert Keller

Abstrakte Algebra Keller:

Signatur $\Sigma = (S, F)$,
 Sorten $S = \{\text{Keller, Element, BOOL}\}$,
 Operationen F :

createStack:		-> Keller
push:	Keller x Element	-> Keller
pop:	Keller	-> Keller
top:	Keller	-> Element
empty:	Keller	-> BOOL

Axiome Q: für beliebige Terme t der Sorte Element und k der Sorte Keller gilt:

K1:	empty (createStack)	-> true
K2:	empty (push (k, t))	-> false
K3:	pop (push (k, t))	-> k
K4:	top (push (k, t))	-> t

Keller ist die Sorte, deren Terme Kellerinhalte modellieren.
 Element und BOOL sind **Hilfssorten** der Algebra.

Implementierungen der abstrakten Algebra Keller können durch **konkrete Algebren** dazu beschrieben werden.

Klassifikation von Operationen

Die Operationen einer Algebra werden in 3 disjunkte Mengen eingeteilt:

- Konstruktoren:** Ergebnissorte ist die definierte Sorte
- Hilfskonstruktoren:** Ergebnissorte ist die definierte Sorte und sie können durch Axiome aus Termen entfernt werden
- Projektionen:** andere Ergebnissorte

z. B. in der Keller-Algebra: definierte Sorte ist Keller

createStack:		-> Keller	Konstruktor
push:	Keller x Element	-> Keller	Konstruktor
pop:	Keller	-> Keller	Hilfskonstruktor (K3 entfernt ihn)
top:	Keller	-> Element	Projektion
empty:	Keller	-> BOOL	Projektion

Normalform

Terme ohne Variable der definierten Sorte sind in **Normalform**, wenn sie nur **Konstruktoren** enthalten **kein Axiom anwendbar** ist.

Normalform-Terme der Algebra Bool sind: true false

Normalform-Terme der Keller-Algebra haben die Form:
 $\text{push} (\dots \text{push} (\text{createStack}, n_1), \dots), n_m)$, mit $m \geq 0$

Die **Terme in Normalform** sind die minimalen Elemente bzgl. der strengen Halbordnung \rightarrow .

Terme s, t , die in **dieselbe Normalform** umformbar sind, heißen **gleichbedeutend**, $s \equiv t$.

Undefinierte Terme:

Terme der definierten Sorte, die man **nicht in eine Normalform** umformen kann, werden als **undefiniert** angesehen. Sie modellieren eine **Fehlersituation**, z. B. $\text{pop} (\text{createStack})$

Für manche **Projektionen** gibt es nicht zu jedem Term in Normalform ein anwendbares Axiom; dies modelliert auch **Fehlersituationen**, z. B. $\text{top} (\text{createStack})$

Anwendungen algebraischer Spezifikationen: Eigenschaften aus den Axiomen erkennen

Beispiel: Keller

1. K3: $\text{pop}(\text{push}(k, t)) \rightarrow k$

Keller-Prinzip: zuletzt eingefügtes Element wird als erstes wieder entfernt (last-in-first-out, LIFO)

2. $\text{top}: \text{Keller} \rightarrow \text{Element}$
K4: $\text{top}(\text{push}(k, t)) \rightarrow t$

top ist die einzige Operation, die Keller-Elemente liefert:

Nur auf das zuletzt eingefügte, nicht wieder entfernte Element kann **zugegriffen** werden.

3. $\text{push}(\dots \text{push}(\text{createStack}, n_1), \dots, n_m)$, mit $m \geq 0$
K3: $\text{pop}(\text{push}(k, t)) \rightarrow k$

Zählt man in einem Term von innen nach außen die push -Operationen positiv und die pop -Operationen negativ, und ist der Wert immer nicht-negativ, so ergibt sich die **Anzahl der Elemente im Keller**, andernfalls ist der Term undefiniert.

Begründung: Rückführung auf Normalform, eine push -Operation für jedes Element im Keller.

Spezifikation um Operationen erweitern

Erweitere die Keller-Spezifikation um eine **Operation size**.

Sie soll die **Anzahl der Elemente im Keller** liefern.

1. Operation **size** in die **Signatur** einfügen:

$\text{size}: \text{Keller} \rightarrow \text{NAT}$

2. Ergebnis-Sorte **NAT** zu den **Sorten** zufügen:

$S = \{\text{Keller}, \text{Element}, \text{BOOL}, \text{NAT}\}$

3. **Axiome** zufügen, so dass size für jeden Keller-Wert definiert ist:

K7: $\text{size}(\text{createStack}) \rightarrow \text{null}$

K8: $\text{size}(\text{push}(k, t)) \rightarrow \text{succ}(\text{size}(k))$

4. Weil in der **Normalform** nur createStack und push vorkommen, braucht size nur für solche Terme definiert zu werden.

Dabei wird vorausgesetzt, dass folgende Algebra bekannt ist:

Sorten: $S = \{\text{NAT}\}$

Operationen: $\text{null}: \rightarrow \text{NAT}$, $\text{succ}: \text{NAT} \rightarrow \text{NAT}$

($\text{succ}(n)$ modelliert den Nachfolger von n , also $n + 1$.)

Realisierung der Spezifikation durch eine konkrete Algebra

Beispiel: eine Realisierung von Kellern durch **Funktionen auf Folgen** von natürlichen Zahlen:

Zuordnung der Sorten:	konkret	abstrakt
	Bool	BOOL
	\mathbb{N}_0	Element
	N-Folge = \mathbb{N}^*	Keller

Signatur und Zuordnung von Funktionen

konkret

newFolge:	-> N-Folge
append: N-Folge x \mathbb{N}_0	-> N-Folge
remove: N-Folge	-> N-Folge
last: N-Folge	-> \mathbb{N}
noElem: N-Folge	-> Bool

abstrakt

createStack
push
pop
top
empty

Definition der Funktionen

newFolge()	-> ()
append ((a_1, \dots, a_n), x)	-> (a_1, \dots, a_n, x)
remove ((a_1, \dots, a_{n-1}, a_n))	-> (a_1, \dots, a_{n-1})
last ((a_1, \dots, a_n))	-> a_n
noElem (f)	-> $f = ()$

Gültigkeit der Axiome zeigen

Keller in Algorithmen einsetzen

Aufgabe: Terme aus **Infixform in Postfixform** umwandeln

gegeben: Term t in Infixform, mit 2-stelligen Operatoren unterschiedlicher Präzedenz; (zunächst) ohne Klammern

gesucht: Term t in Postfixform

Eigenschaften der Aufgabe und der Lösung:

- Reihenfolge der Variablen und Konstanten bleibt unverändert**
- Variablen und Konstanten werden vor ihrem Operator ausgegeben**, also sofort
- In der Infixform aufeinander folgende **Operatoren echt steigender Präzedenz** stehen in der Postfixform **in umgekehrter Reihenfolge**; also kellern.
- Operatorkeller enthält Operatoren echt steigender Präzedenz.**
Es gilt die **Kellerinvariante KI**:
Sei $\text{push}(\dots \text{push}(\text{CreateStack}, \text{opr}_1), \text{opr}_2), \dots)$ dann gilt
Präzedenz (opr_i) < Präzedenz (opr_{i+1})

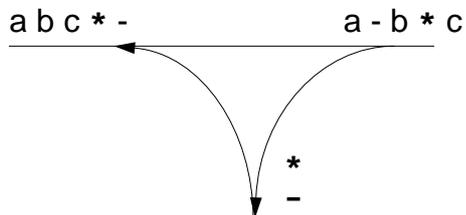
Algorithmus: Infix- in Postfixform wandeln

Die Eingabe enthält einen Term in Infixform;
die Ausgabe soll den Term in Postfixform enthalten

Variable: keller \in Keller; symbol \in Operator \cup ElementarOperand

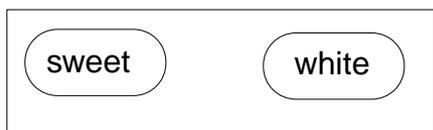
```

keller = createStack();
solange Eingabe nicht leer wiederhole           {KI}
    lies symbol
    falls symbol  $\in$  ElementarOperand
        gib symbol aus
    falls symbol  $\in$  Operator                   {KI}
        solange not empty (keller)  $\wedge$ 
            Präzedenz (top (keller))  $\geq$  Präzedenz (symbol)
            wiederhole                           {KI}
                gib top (keller) aus;
                keller = pop (keller);
            keller = push(keller, symbol);       {KI}
solange not empty (keller) wiederhole
    gib top(keller) aus;
    keller = pop(keller);
    
```



An den Stellen {KI} gilt die Kellerinvariante.

Abstrakte Algebra für Teilaspekt des Getränkeautomaten



Knöpfe des Getränkeautomaten
zur Auswahl von Zutaten

Die Sorte **Choice** modelliert die Auswahl;
Add ist eine Hilfssorte

Signatur $\Sigma = (S, F)$;

Sorten $S := \{\text{Add, Choice}\}$

Operationen F :

- sweet: \rightarrow Add
- white: \rightarrow Add
- noChoice: \rightarrow Choice
- press: Add x Choice \rightarrow Choice

Bedeutung der Axiome:

Q_1 : Knopf nocheinmal drücken
macht Auswahl rückgängig.

Q_2 : Es ist egal, in welcher
Reihenfolge die Knöpfe
gedrückt werden.

Axiome Q : für alle a der Sorte Add und
für alle c der Sorte Choice gilt:

- Q_1 : $\text{press}(a, \text{press}(a, c)) \rightarrow c$
- Q_2 : $\text{press}(\text{sweet}, \text{press}(\text{white}, c)) \rightarrow$
 $\text{press}(\text{white}, \text{press}(\text{sweet}, c))$

Beispiel-Terme: $\text{press}(\text{white}, \text{noChoice})$
 $\text{press}(\text{sweet}, \text{press}(\text{white}, \text{press}(\text{sweet}, \text{noChoice})))$