Programmierparadigmen

0. Auflage, 5. März 2014

Martin Thoma

Vorwort

Dieses Skript wird/wurde im Wintersemester 2013/2014 von Martin Thoma zur Vorlesung von Prof. Dr. Snelting und Jun.-Prof. Dr. Hummel geschrieben. Dazu wurden die Folien von Prof. Dr. Snelting und Jun.-Prof. Dr. Hummel benutzt, die Struktur sowie einige Beispiele, Definitionen und Sätze übernommen.

Das Ziel dieses Skriptes ist vor allem in der Klausur als Nachschlagewerk zu dienen; es soll jedoch auch vorher schon für die Vorbereitung genutzt werden können und nach der Klausur als Nachschlagewerk dienen.

Ein Link auf das Skript ist unter
`martin-thoma.com/programmierparadigmen`
zu finden.

Anregungen, Verbesserungsvorschläge, Ergänzungen

Noch ist das Skript im Aufbau. Es gibt viele Baustellen und es ist fraglich, ob ich bis zur Klausur alles in guter Qualität bereitstellen kann. Daher freue ich mich über jeden Verbesserungsvorschlag.

Anregungen, Verbesserungsvorschläge und Ergänzungen können per Pull-Request gemacht werden oder mir per Email an info@martin-thoma.de geschickt werden.

Was ist Programmierparadigmen?

TODO

Erforderliche Vorkenntnisse

Grundlegende Kenntnisse vom Programmieren, insbesondere mit Java, wie sie am KIT in „Programmieren“ vermittelt werden, werden vorausgesetzt. Außerdem könnte ein grundlegendes Verständnis für das $\mathcal{O}$-Kalkül aus „Grundbegriffe der Informatik“ hilfreich sein.

Die Unifikation wird wohl auch in „Formale Systeme“ erklärt; das könnte also hier von Vorteil sein.

Inhaltsverzeichnis

1 Programmiersprachen

Im folgenden werden einige Begriffe definiert anhand derer Programmiersprachen unterschieden werden können.

Definition 1

Eine **Programmiersprache** ist eine formale Sprache, die durch eine Spezifikation definiert wird und mit der Algorithmen beschrieben werden können. Elemente dieser Sprache heißen **Programme**.

Ein Beispiel für eine Sprachspezifikation ist die *Java Language Specification*.[1] Obwohl es kein guter Stil ist, ist auch eine Referenzimplementierung eine Form der Spezifikation.

Im Folgenden wird darauf eingegangen, anhand welcher Kriterien man Programmiersprachen unterscheiden kann.

1.1 Abstraktion

Wie nah an den physikalischen Prozessen im Computer ist die Sprache? Wie nah ist sie an einer mathematisch / algorithmischen Beschreibung?

Definition 2

Eine **Maschinensprache** beinhaltet ausschließlich Instruktionen, die direkt von einer CPU ausgeführt werden können. Die Menge dieser Instruktionen sowie deren Syntax wird **Befehlssatz** genannt.

[1] Zu finden unter `http://docs.oracle.com/javase/specs/`

Beispiel 1 (Maschinensprachen)
1) x86:

2) SPARC:

Definition 3
Assembler TODO

Beispiel 2 (Assembler)
TODO

1.2 Paradigmen

Die grundlegendste Art, wie man Programmiersprachen unterscheiden kann ist das sog. „Programmierparadigma“, also die Art wie man Probleme löst.

Definition 4 (Imperatives Paradigma)
In der imperativen Programmierung betrachtet man Programme als eine folge von Anweisungen, die vorgibt auf welche Art etwas Schritt für Schritt gemacht werden soll.

Definition 5 (Prozedurales Paradigma)
Die prozeduralen Programmierung ist eine Erweiterung des imperativen Programmierparadigmas, bei dem man versucht die Probleme in kleinere Teilprobleme zu zerlegen.

Definition 6 (Funktionales Paradigma)
In der funktionalen Programmierung baut man auf Funktionen und ggf. Funktionen höherer Ordnung, die eine Aufgabe ohne Nebeneffekte lösen.

Haskell ist eine funktionale Programmiersprache, C ist eine nichtfunktionale Programmiersprache.

Wichtige Vorteile von funktionalen Programmiersprachen sind:

- Sie sind weitgehend (jedoch nicht vollständig) frei von Seiteneffekten.

- Der Code ist häufig sehr kompakt und manche Probleme lassen sich sehr elegant formulieren.

Definition 7 (Logisches Paradigma)
In der logischen Programmierung baut auf der Unifikation auf.

genauer

1.3 Typisierung

Eine weitere Art, Programmiersprachen zu unterscheiden ist die stärke ihrer Typisierung.

Definition 8 (Dynamische Typisierung)
Bei dynamisch typisierten Sprachen kann eine Variable ihren Typ ändern.

Beispiele sind Python und PHP.

Definition 9 (Statische Typisierung)
Bei statisch typisierten Sprachen kann eine niemals ihren Typ ändern.

Beispiele sind C, Haskell und Java.

1.4 Kompilierte und interpretierte Sprachen

Sprachen werden überlicherweise entweder interpretiert oder kompiliert, obwohl es Programmiersprachen gibt, die beides unterstützen.

C und Java werden kompiliert, Python und TCL interpretiert.

1.5 Dies und das

Definition 10 (Seiteneffekt)
Seiteneffekte sind Veränderungen des Zustandes.

Das geht besser

Manchmal werden Seiteneffekte auch als Nebeneffekt oder Wirkung bezeichnet.

Definition 11 (Unifikation)

Die Unifikation ist eine Operation in der Logik und dient zur Vereinfachung prädikatenlogischer Ausdrücke.

Das ist keine formale Definition!

Beispiel 3 (Unifikation)

2 Programmiertechniken

2.1 Rekursion

Definition 12 (rekursive Funktion)
Eine Funktion $f : X \to X$ heißt rekursiv definiert, wenn in der Definition der Funktion die Funktion selbst wieder steht.

Beispiel 4 (rekursive Funktionen)

1) Fibonacci-Funktion:

$$fib : \mathbb{N}_0 \to \mathbb{N}_0$$
$$fib(n) = \begin{cases} n & \text{falls } n \leq 1 \\ fib(n-1) + fib(n-2) & \text{sonst} \end{cases}$$

Erzeugt die Zahlen $0, 1, 1, 2, 3, 5, 8, 13, \ldots$

2) Fakultät:

$$! : \mathbb{N}_0 \to \mathbb{N}_0$$
$$n! = \begin{cases} 1 & \text{falls } n \leq 1 \\ n \cdot (n-1)! & \text{sonst} \end{cases}$$

3) Binomialkoeffizient:

$$\binom{\cdot}{\cdot} : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$$
$$\binom{n}{k} = \begin{cases} 1 & \text{falls } k = 0 \vee k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{sonst} \end{cases}$$

Ein Problem von rekursiven Funktionen in Computerprogrammen ist der Speicherbedarf. Für jeden rekursiven Aufruf müssen alle Umgebungsvariablen der aufrufenden Funktion („stack frame“) gespeichert bleiben, bis der rekursive Aufruf beendet ist. Im Fall der Fibonacci-Funktion sieht ist der Call-Stack in Abb. 2.1 abgebildet.

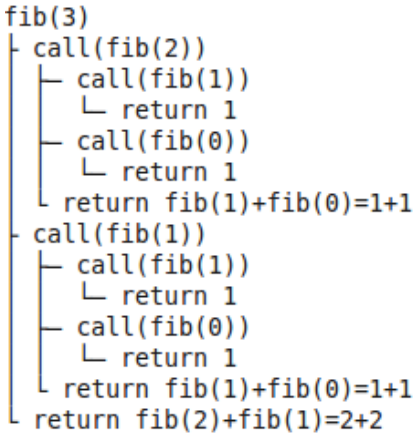

Abbildung 2.1: Call-Stack der Fibonacci-Funktion

Bemerkung 1

Die Anzahl der rekursiven Aufrufe der Fibonacci-Funktion f_C ist:

$$f_C(n) = \begin{cases} 1 & \text{falls } n = 0 \\ 2 \cdot fib(n) - 1 & \text{falls } n \geq 1 \end{cases}$$

Beweis:

- Offensichtlich gilt $f_C(0) = 1$
- Offensichtlich gilt $f_C(1) = 1 = 2 \cdot fib(1) - 1$
- Offensichtlich gilt $f_C(2) = 3 = 2 \cdot fib(2) - 1$
- Für $n \geq 3$:

$$\begin{aligned} f_C(n) &= 1 + f_C(n-1) + f_C(n-2) \\ &= 1 + (2 \cdot fib(n-1) - 1) + (2 \cdot fib(n-2) - 1) \end{aligned}$$

$$= 2 \cdot (fib(n-1) + fib(n-2)) - 1$$
$$= 2 \cdot fib(n) - 1$$

Mit Hilfe der Formel von Moivre-Binet folgt:

$$f_C \in \mathcal{O}\left(\frac{\varphi^n - \psi^n}{\varphi - \psi}\right) \text{ mit } \varphi := \frac{1+\sqrt{5}}{2} \text{ und } \psi := 1 - \varphi$$

Dabei ist der Speicherbedarf $\mathcal{O}(n)$. Dieser kann durch das Benutzen eines Akkumulators signifikant reduziert werden. TODO

Definition 13 (linear rekursive Funktion)
Eine Funktion heißt linear rekursiv, wenn in jedem Definitionszweig der Funktion höchstens ein rekursiver Aufruf vorkommt.

Definition 14 (endrekursive Funktion)
Eine Funktion heißt endrekursiv, wenn in jedem Definitionszweig der Rekursive aufruf am Ende des Ausdrucks steht. Der rekursive Aufruf darf also insbesondere nicht in einen anderen Ausdruck eingebettet sein.

Auf Englisch heißen endrekursive Funktionen *tail recursive*.

Beispiel 5 (Linear- und endrekursive Funktionen)

1) `fak n = if (n==0) then 1 else (n * fak (n-1))`
ist eine linear rekursive Funkion, aber nicht endrekursiv, da nach der Rückgabe von `fak (n-1)` noch die Multiplikation ausgewertet werden muss.

2) `fakAcc n acc = if (n==0) then acc else fakAcc (n-1) (n*acc)`
ist eine endrekursive Funktion.

3) `fib n = n <= 1 ? n : fib(n-1) + fib (n-2)`
ist weder linear- noch endrekursiv.

Wenn eine rekursive Funktion nicht terminiert oder wenn

2.2 Backtracking

2.3 Funktionen höherer Ordnung

Funktionen höherer Ordnung sind Funktionen, die auf Funktionen arbeiten. Bekannte Beispiele sind:

- `map(function, list)`
  `map` wendet `function` auf jedes einzelne Element aus `list` an.
- `filter(function, list)`
  `filter` gibt eine Liste aus Elementen zurück, für die `function` mit `true` evaluiert.
- `reduce(function, list)`
  `function` ist für zwei Elemente aus `list` definiert und gibt ein Element des gleichen Typs zurück. Nun steckt `reduce` zuerst zwei Elemente aus `list` in `function`, merkt sich dann das Ergebnis und nimmt so lange weitere Elemente aus `list`, bis jedes Element genommen wurde.
  Bei `reduce` ist die Assoziativität wichtig (vgl. Seite 43)

3 Logik

3.1 Prädikatenlogik erster Stufe

Folgendes ist von `http://de.wikipedia.org/wiki/Pr%C3%A4dikatenlogik_erster_Stufe`

Die Prädikatenlogik erster Stufe ist ein Teilgebiet der mathematischen Logik. Sie befasst sich mit der Struktur gewisser mathematischer Ausdrücke und dem logischen Schließen, mit dem man von derartigen Ausdrücken zu anderen gelangt. Dabei gelingt es, sowohl die Sprache als auch das Schließen rein syntaktisch, das heißt ohne Bezug zu mathematischen Bedeutungen, zu definieren. [...]

Wir beschreiben hier die verwendete Sprache auf rein syntaktische Weise, das heißt wir legen die betrachteten Zeichenketten, die wir Ausdrücke der Sprache nennen wollen, ohne Bezug auf ihre Bedeutung fest.

3.1.1 Symbole

Eine Sprache erster Stufe wird aus folgenden Symbolen aufgebaut:

- $\forall, \exists, \wedge, \vee, \rightarrow, \leftrightarrow, \neg, (,), \equiv$
- sogenannte Variablensymbole $v_0, v_1, v_2, \ldots,$
- eine (möglicherweise leere) Menge $\mathcal{C}$ von Konstantensymbolen,
- eine (möglicherweise leere) Menge $\mathcal{F}$ von Funktionssymbolen,

- eine (möglicherweise leere) Menge $\mathcal{R}$ von Relationssymbolen.

Das Komma wird hier nur als Trennzeichen für die Aufzählung der Symbole benutzt, es ist nicht Symbol der Sprache.

3.1.2 Terme

Die nach folgenden Regeln aufgebauten Zeichenketten heißen Terme:

- Ist v ein Variablensymbol, so ist v ein Term.
- Ist c ein Konstantensymbol, so ist c ein Term.
- Ist f ein 1-stelliges Funktionssymbol und ist t_1 ein Term, so ist ft_1 ein Term.
- Ist f ein 2-stelliges Funktionssymbol und sind t_1, t_2 Terme, so ist ft_1t_2 ein Term.
- Ist f ein 3-stelliges Funktionssymbol und sind t_1, t_2, t_3 Terme, so ist $ft_1t_2t_3$ ein Term.
- und so weiter für 4,5,6,...-stellige Funktionssymbole.

Ist zum Beispiel c eine Konstante und sind f und g 1- bzw. 2-stellige Funktionssymbole, so ist fgv_2fc ein Term, da er sich durch Anwendung obiger Regeln erstellen lässt: c ist ein Term, daher auch fc; fc und v_2 sind Terme, daher auch gv_2fc und damit schließlich auch fgv_2fc.

Wir verzichten hier auf Klammern und Kommata als Trennzeichen, das heißt wir schreiben fgv_2fc und nicht $f(g(v_2, f(c)))$. Wir setzen damit implizit voraus, dass unsere Symbole derart beschaffen sind, dass eine eindeutige Lesbarkeit gewährleistet ist.

Die Regeln für die Funktionssymbole fasst man oft so zusammen:

- Ist f ein n-stelliges Funktionssymbol und sind $t_1, \ldots, t_n$ Terme, so ist $ft_1 \ldots t_n$ ein Term.

Damit ist nichts anderes als die oben angedeutete unendliche Folge von Regeln gemeint, denn die drei Punkte ... gehören nicht zu den vereinbarten Symbolen. Dennoch wird manchmal von dieser Schreibweise Gebrauch gemacht.

Über den Aufbau der Terme lassen sich weitere Eigenschaften definieren. So definieren wir offenbar durch die folgenden drei Regeln rekursiv, welche Variablen in einem Term vorkommen:

- Ist v ein Variablensymbol, so sei $\mathrm{var}(v) = \{v\}$.
- Ist c ein Konstantensymbol, so sei $\mathrm{var}(c) = \emptyset$.
- Ist f ein n-stelliges Funktionssymbol und sind $t_1, \ldots, t_n$ Terme, so sei $\mathrm{var}(ft_1 \ldots t_n) = \mathrm{var}(t_1) \cup \ldots \cup \mathrm{var}(t_n)$.

3.1.3 Ausdrücke

Wir erklären nun durch Bildungsgesetze, welche Zeichenketten wir als Ausdrücke der Sprache ansehen wollen.

Atomare Ausdrücke

- Sind t_1 und t_2 Terme, so ist $t_1 \equiv t_2$ ein Ausdruck.
- Ist R ein 1-stelliges Relationssymbol und ist t_1 ein Term, so ist Rt_1 ein Ausdruck.
- Ist R ein 2-stelliges Relationssymbol und sind t_1, t_2 Terme, so ist Rt_1t_2 ein Ausdruck.
- und so weiter für 3,4,5,...-stellige Relationssymbole.

Dabei gelten die oben zur Schreibweise bei Termen gemachten Bemerkungen.

Zusammengesetzte Ausdrücke

Wir beschreiben hier, wie sich aus Ausdrücken weitere gewinnen lassen.

- Ist φ ein Ausdruck, so ist auch $\neg\varphi$ ein Ausdruck.
- Sind φ und ψ Ausdrücke, so sind auch $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\varphi \to \psi)$ und $(\varphi \leftrightarrow \psi)$ Ausdrücke.
- Ist φ ein Ausdruck und ist x eine Variable, so sind auch $\forall x\varphi$ und $\exists x\varphi$ Ausdrücke.

Damit sind alle Ausdrücke unserer Sprache festgelegt. Ist zum Beispiel f ein 1-stelliges Funktionssymbol und R ein 2-stelliges Relationssymbol, so ist : $\forall v_0((Rv_0v_1 \vee v_0 \equiv fv_1) \to \exists v_2 \neg Rv_0v_2)$ ein Ausdruck, da er sich durch Anwendung obiger Regeln aufbauen lässt. Es sei noch einmal darauf hingewiesen, dass wir die Ausdrücke mittels der genannten Regeln rein mechanisch erstellen, ohne dass die Ausdrücke zwangsläufig irgendetwas bezeichnen müssten.

3.1.4 1. Stufe

Unterschiedliche Sprachen erster Stufe unterscheiden sich lediglich in den Mengen $\mathcal{C}$, $\mathcal{F}$ und $\mathcal{R}$, die man üblicherweise zur Symbolmenge S zusammenfasst und auch die *Signatur* der Sprache nennt. Man spricht dann auch genauer von S-Termen bzw. S-Ausdrücken. Die Sprache, das heißt die Gesamtheit aller nach obigen Regeln gebildeten Ausdrücke, wird mit $L(S)$, L^S oder L^S_I bezeichnet. Bei letzterem steht die römische I für die 1-te Stufe. Dies bezieht sich auf den Umstand, dass gemäß letzter Erzeugungsregel nur über Variable quantifiziert werden kann. L^S_I sieht nicht vor, über alle Teilmengen einer Menge oder über alle Funktionen zu quantifizieren. So lassen sich die üblichen [[Peano-Axiome]] nicht in L^S_I ausdrücken, da das Induktionsaxiom eine Aussage über alle Teilmengen der natürlichen Zahlen macht. Das kann als Schwäche dieser Sprache angesehen werden, allerdings sind die Axiome

der [[Zermelo-Fraenkel-Mengenlehre]] sämtlich in der ersten Stufe mit dem einzigen Symbol $\in$ formulierbar, so dass die erste Stufe prinzipiell für die Mathematik ausreicht.

3.1.5 Freie Variablen

Weitere Eigenschaften von Ausdrücken der Sprache L_I^S lassen sich ebenfalls rein syntaktisch definieren. Gemäß dem oben beschriebenen Aufbau durch Bildungsregeln definieren wir die Menge $\text{frei}(\varphi)$ der im Ausdruck φ frei vorkommenden Variablen wie folgt:

- $\text{frei}(t_1 \equiv t_2) = \text{var}(t_1) \cup \text{var}(t_2)$
- $\text{frei}(Rt_1 \dots t_n) = \text{var}(t_1) \cup \dots \cup \text{var}(t_n)$
- $\text{frei}(\neg\varphi) = \text{frei}(\varphi)$
- $\text{frei}(\varphi \wedge \psi) = \text{frei}(\varphi) \cup \text{frei}(\psi)$ und genauso für $\vee, \rightarrow, \leftrightarrow$
- $\text{frei}(\forall x\varphi) = \text{frei}(\varphi) \setminus \{x\}$
- $\text{frei}(\exists x\varphi) = \text{frei}(\varphi) \setminus \{x\}$

Nicht-freie Variable heißen *gebundene Variable*. Ausdrücke φ ohne freie Variable, das heißt solche mit $\text{frei}(\varphi) = \emptyset$, nennt man *Sätze*. Sämtliche in obigem motivierenden Beispiel angegebenen Axiome der geordneten abelschen Gruppen sind bei entsprechender Übersetzung in die Sprache $L_I^{\{0,+,-,\leq\}}$ Sätze, so zum Beispiel $\forall v_0 \forall v_1 + v_0 v_1 \equiv + v_1 v_0$ für das Kommutativgesetz.

3.1.6 Metasprachliche Ausdrücke

Das gerade gegebene Beispiel $\forall v_0 \forall v_1 + v_0 v_1 \equiv + v_1 v_0$ als Symbolisierung des Kommutativgesetzes in der Sprache $L_I^{\{0,+,-,\leq\}}$ zeigt, dass die entstehenden Ausdrücke oft schwer lesbar sind. Daher kehrt der Mathematiker, und oft auch der Logiker, gern zur klassischen Schreibweise $\forall x, y : x + y = y + x$ zurück. Letzteres ist aber

kein Ausdruck der Sprache $L_I^{\{0,+,-,\leq\}}$ sondern nur eine Mitteilung eines solchen Ausdrucks unter Verwendung anderer Symbole einer anderen Sprache, hier der sogenannten [[Metasprache]], das heißt derjenigen Sprache, in der man über $L_I^{\{0,+,-,\leq\}}$ spricht. Aus Gründen der besseren Lesbarkeit lässt man auch gern überflüssige Klammern fort. Das führt nicht zu Problemen, solange klar bleibt, dass man die leichter lesbaren Zeichenketten jederzeit zurückübersetzen könnte.

3.1.7 Substitutionen

Häufig werden in der Mathematik Variablen durch Terme ersetzt. Auch das lässt sich hier rein syntaktisch auf Basis unserer Symbole erklären. Durch folgende Regeln legen wir fest, was es bedeuten soll, den Term t für eine Variable x einzusetzen. Wir folgen dabei wieder dem regelhaften Aufbau von Termen und Ausdrücken. Die Ersetzung wird als $[\,]\frac{t}{x}$ notiert, wobei die eckigen Klammern weggelassen werden dürfen.

Für Terme s wird die Einsetzung $s\frac{t}{x}$ wie folgt definiert:

- Ist v ein Variablensymbol, so ist $v\frac{t}{x}$ gleich t falls $v = x$ und v sonst.
- Ist c ein Konstantensymbol, so ist $c\frac{t}{x} := c$.
- Sind f ein n-stelliges Funktionssymbol und $t_1, \ldots, t_n$ Terme, so ist $[ft_1 \ldots t_n]\frac{t}{x} := ft_1\frac{t}{x} \ldots t_n\frac{t}{x}$.

Für Ausdrücke schreiben wir eckige Klammern um den Ausdruck, in dem die Substitution vorgenommen werden soll. Wir legen fest:

- $[t_1 \equiv t_2]\frac{t}{x} := t_1\frac{t}{x} \equiv t_2\frac{t}{x}$
- $[Rt_1 \ldots t_n]\frac{t}{x} := Rt_1\frac{t}{x} \ldots t_n\frac{t}{x}$
- $[\neg\varphi]\frac{t}{x} := \neg[\varphi]\frac{t}{x}$
- $[(\varphi \vee \psi)]\frac{t}{x} := ([\varphi]\frac{t}{x} \vee [\psi]\frac{t}{x})$ und genauso für $\wedge, \rightarrow, \leftrightarrow$

- $[\exists x\varphi]\frac{t}{x} := \exists x\varphi$; analog für den Quantor $\forall$
- $[\exists y\varphi]\frac{t}{x} := \exists y[\varphi]\frac{t}{x}$ falls $x \neq y$ und $y \notin \text{var}(t)$; analog für den Quantor $\forall$
- $[\exists y\varphi]\frac{t}{x} := \exists u[\varphi]\frac{u}{y}\frac{t}{x}$ falls $x \neq y$ und $y \in \text{var}(t)$, wobei u eine Variable sei, die nicht in φ oder t vorkommt, zum Beispiel die erste der Variablen $v_0, v_1, v_2, \ldots$, die diese Bedingung erfüllt. Die analoge Festlegung wird für $\forall$ getroffen.

Bei dieser Definition wurde darauf geachtet, dass Variablen nicht unbeabsichtigt in den Einflussbereich eines Quantors geraten. Falls die gebundene Variable x im Term auftritt, so wird diese zuvor durch eine andere ersetzt, um so die Variablenkollision zu vermeiden.

Definition 15 (Freie Variable)
Eine Variable, die nicht gebunden ist, heißt frei.

Beispiel 6 (Freie Variablen[1])
In dem Ausduck $(\lambda x \to xy)$ ist y eine freie Variable.

Definition 16 (Kombinator)
Ein Kombinator ist eine Funktion oder Definition ohne freie Variablen.

Beispiel 7 (Kombinatoren[2])

1) $\lambda a \to a$
2) $\lambda a \to \lambda b \to a$
3) $\lambda f \to \lambda a \to \lambda b \to fba$

[1] Quelle: `http://www.haskell.org/haskellwiki/Free_variable`
[2] Quelle: `http://www.haskell.org/haskellwiki/Combinator`

4 λ-Kalkül

Der Lambda-Kalkül ist eine formale Sprache. In diesem Kalkül gibt es drei Arten von Termen T:

- Variablen: x
- Applikationen: (TT)
- Lambda: $\lambda x.T$

Es ist zu beachten, dass Funktionsapplikation linksassoziativ ist. Es gilt also:

$$a\ b\ c\ d = ((a\ b)\ c)\ d$$

Definition 17 (Freie Variable)
TODO

Definition 18 (Gebundene Variable)
Eine Variable heißt gebunden, wenn sie nicht frei ist.

4.1 Reduktionen

Definition 19 (α-Äquivalenz)
Zwei Terme T_1, T_2 heißen α-Äquivalent, wenn T_1 durch konsistente Umbenennung in T_2 überführt werden kann.

Man schreibt dann: $T_1 \stackrel{\alpha}{=} T_2$.

Beispiel 8 (α-Äquivalenz)

$$\lambda x.x \stackrel{\alpha}{=} \lambda y.y$$
$$\lambda x.xx \stackrel{\alpha}{=} \lambda y.yy$$
$$\lambda x.(\lambda y.z(\lambda x.zy)y) \stackrel{\alpha}{=} \lambda a.(\lambda x.z(\lambda c.zx)x)$$

Definition 20 (β-Äquivalenz)
TODO

Beispiel 9 (β-Äquivalenz)
TODO

Definition 21 (η-Äquivalenz)
Zwei Terme $\lambda x.f\ x$ und f heißen η-Äquivalent, wenn x nicht freie Variable von f ist.

Beispiel 10 (η-Äquivalenz)
TODO

4.2 Auswertungsstrategien

Definition 22 (Normalenreihenfolge)
In der Normalenreihenfolge-Auswertungsstrategie wird der linkeste äußerste Redex ausgewertet.

Definition 23 (Call-By-Name)
In der Call-By-Name Auswertungsreihenfolge wird der linkeste äußerste Redex reduziert, der nicht von einem λ umgeben ist.

Die Call-By-Name Auswertung wird in Funktionen verwendet.

Definition 24 (Call-By-Value)
In der Call-By-Value Auswertung wird der linkeste Redex reduziert, der nicht von einem λ umgeben ist und dessen Argument ein Wert ist.

Die Call-By-Value Auswertungsreihenfolge wird in C und Java verwendet. Auch in Haskell werden arithmetische Ausdrücke in der Call-By-Name Auswertungsreihenfolge reduziert.

4.3 Church-Zahlen

TODO

4.4 Weiteres

Satz 4.1 (Satz von Curch-Rosser)
Wenn zwei unterschiedliche Terme a und b äquivalent sind, d.h. mit Reduktionsschritten beliebiger Richtung ineinander transformiert werden können, dann gibt es einen weiteren Term c, zu dem sowohl a als auch b reduziert werden können.

5 Typinferenz

Definition 25 (Datentyp)
Ein *Datentyp* oder kurz *Typ* ist eine Menge von Werten, mit denen eine Bedeutung verbunden ist.

Beispiel 11 (Datentypen)

- `bool` = { True, False }
- `char` = vgl. Seite 57
- $\texttt{int}_{\text{Haskell}} = [-2^{29}, 2^{29} - 1] \cap \mathbb{N}$
- $\texttt{int}_{\text{C90}} = [-2^{15} - 1, 2^{15} - 1] \cap \mathbb{N}$[1]
- `float` = siehe IEEE 754
- Funktionstypen, z. B. `int` → `int` oder `char` → `int`

Hinweis: Typen sind unabhängig von ihrer Repräsentation. So kann ein `bool` durch ein einzelnes Bit repräsentiert werden oder eine Bitfolge zugrunde liegen.

Auf Typen sind Operationen definiert. So kann man auf numerischen Typen eine Addition (+), eine Subtraktion (-), eine Multiplikation (*) und eine Division (/) definieren.
Ich schreibe hier bewusst „eine“ Multiplikation und nicht „die“ Multiplikation, da es verschiedene Möglichkeiten gibt auf Gleitpunktzahlen Multiplikationen zu definieren. So kann man beispielsweise die Assoziativität unterschiedlich wählen.

Beispiel 12 (Multiplikation ist nicht assoziativ)
In Python 3 ist die Multiplikation linksassoziativ. Also:

[1] siehe ISO/IEC 9899:TC2, Kapitel 7.10: Sizes of integer types <limits.h>

```
>>> 0.1*0.1*0.3
0.0030000000000000005
>>> (0.1*0.1)*0.3
0.0030000000000000005
>>> 0.1*(0.1*0.3)
0.003
```

Definition 26 (Typvariable)
Eine Typvariable repräsentiert einen Typen.

Hinweis: Üblicherweise werden kleine griechische Buchstaben ($\alpha, \beta, \tau_1, \tau_2, \dots$) als Typvariablen gewählt.

Genau wie Typen bestimmte Operationen haben, die auf ihnen definiert sind, kann man sagen, dass Operationen bestimmte Typen, auf die diese Anwendbar sind. So ist

$$\alpha + \beta$$

für numerische α und β wohldefiniert, auch wenn α und β boolesch sind oder beides Strings sind könnte das Sinn machen. Es macht jedoch z. B. keinen Sinn, wenn α ein String ist und β boolesch.

Die Menge aller Operationen, die auf die Variablen angewendet werden, nennt man **Typkontext**. Dieser wird üblicherweise mit Γ bezeichnet.

Das Ableiten einer Typisierung für einen Ausdruck nennt man **Typinferenz**. Man schreibt: $\vdash (\lambda x.2) : \alpha \to \text{int}$.

Bei solchen Ableitungen sind häufig viele Typen möglich. So kann der Ausdruck

$$\lambda x.2$$

Mit folgenderweise typisiert werden:

- $\vdash (\lambda x.2) : \text{bool} \to \mathit{int}$
- $\vdash (\lambda x.2) : \text{int} \to \mathit{int}$

- $\vdash (\lambda x.2) : \text{Char} \to int$
- $\vdash (\lambda x.2) : \alpha \to int$

In der letzten Typisierung stellt α einen beliebigen Typen dar.

Ein Typkontext Γ ordnet jeder freien Variable x einen Typ $\Gamma(x)$ durch folgende Regeln zu:

$$\texttt{CONST}: \frac{c \in \text{Const}}{\Gamma \vdash c : \tau_c}$$

$$\texttt{VAR}: \frac{\Gamma(x) = \tau}{\Gamma \vdash c : \tau}$$

$$\texttt{ABS}: \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2}$$

$$\texttt{APP}: \frac{\Gamma \vdash t_1, \tau_2\tau \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1 t_2 : \tau}$$

Dabei ist der lange Strich kein Bruchstrich, sondern ein Symbol der Logik das als **Schlussstrich** bezeichnet wird. Dabei ist der Zähler als Voraussetzung und der Nenner als Schlussfolgerung zu verstehen.

Definition 27 (Typsubstituition)

Eine *Typsubstituition* ist eine endliche Abbildung von Typvariablen auf Typen.

Für eine Menge von Typsubsitutionen wird überlicherweise σ als Symbol verwendet. Man schreibt also beispielsweise:

$$\sigma = [\alpha_1 \mapsto \texttt{bool}, \alpha_2 \mapsto \alpha_1 \to \alpha_1]$$

Definition 28 (Lösung eines Typkontextes)

Sei t eine beliebige freie Variable, $\tau = \tau(t)$ ein beliebiger Typ σ eine Menge von Typsubstitutionen und Γ ein Typkontext.

(σ, τ) heißt eine Lösung für (Γ, t), falls gilt:

$$\sigma\Gamma \vdash t : \tau$$

6 Parallelität

Systeme mit mehreren Prozessoren sind heutzutage weit verbreitet. Inzwischen sind sowohl in Desktop-PCs als auch Laptops, Tablets und Smartphones „Multicore-CPUs“ verbaut. Daher sollten auch Programmierer in der Lage sein, Programme für mehrere Kerne zu entwickeln.

Parallelverarbeitung kann auf mehreren Ebenen statt finden:

- **Bit-Ebene**: Werden auf 32-Bit Computern `long long`, also 64-Bit Zahlen, addiert, so werden parallel zwei 32-Bit Additionen durchgeführt und das carry-flag benutzt.
- **Anweisungs-Ebene**: Die Ausführung von Anweisungen in der CPU besteht aus mehreren Phasen (Instruction Fetch, Decode, Execution, Write-Back). Besteht zwischen aufeinanderfolgenden Anweisungen keine Abhängigkeit, so kann der Instruction Fetch-Teil einer zweiten Anweisung parallel zum Decode-Teil einer ersten Anweisung geschehen. Das nennt man Pipelining.
- **Datenebene**: Es kommt immer wieder vor, dass man in Schleifen eine Operation für jedes Objekt eines Contaitainers (z. B. einer Liste) durchführen muss. Zwischen den Anweisungen verschiedener Schleifendurchläufe besteht dann eventuell keine Abhängigkeit. Dann können alle Schleifenaufrufe parallel durchgeführt werden.
- **Verarbeitungsebene**: Verschiedene Programme sind unabhängig von einander.

Gerade bei dem letzten Punkt ist zu beachten, dass echt parallele Ausführung nicht mit *verzahnter Ausführung* zu verwechseln ist. Auch bei Systemen mit nur einer CPU und einem Kern kann man gleichzeitig den Browser nutzen und einen Film über eine Multimedia-Anwendung laufen lassen. Dabei wechselt der Scheduler sehr schnell zwischen den verschiedenen Anwendungen, sodass es sich so anfühlt, als würden die Programme echt parallel ausgeführt werden.

Weitere Informationen zu Pipelining gibt es in der Vorlesung „Rechnerorganisation“ bzw. „Digitaltechnik und Entwurfsverfahren“ (zu der auch ein exzellentes Skript angeboten wird). Informationen über Schedulung werden in der Vorlesung „Betriebssysteme“ vermittelt.

6.1 Architekturen

Es gibt zwei Ansätze, wie man Parallelrechner entwickeln kann:

- **Gemeinsamer Speicher**: In diesem Fall kann jeder Prozessor jede Speicherzelle ansprechen. Dies ist bei Multicore-CPUs der Fall.
- **Verteilter Speicher**: Es ist auch möglich, dass jeder Prozessor seinen eigenen Speicher hat, der nur ihm zugänglich ist. In diesem Fall schicken die Prozessoren Nachrichten (engl. *message passing*). Diese Technik wird in Clustern eingesetzt.

Eine weitere Art, wie man Parallelverarbeitung klassifizieren kann, ist anhand der verwendeten Architektur. Der der üblichen, sequentiellen Art der Programmierung, bei der jeder Befehl nach einander ausgeführt wird, liegt die sog. **Von-Neumann-Architektur** zugrunde. Bei der Programmierung von parallel laufenden Anwendungen kann man das **PRAM-Modell** (kurz für *Parallel Random Access Machine*) zugrunde legen. In diesem Modell geht man von ei-

ner beliebigen Anahl an Prozessoren aus, die über lokalen Speicher verfügen und synchronen Zugriff auf einen gemeinsamen Speicher haben.

Anhand der **Flynn'schen Klassifikation** können Rechnerarchitekturen in vier Kategorien unterteilt werden:

	Single Instruction	Multiple Instruction
Single Data	SISD	MISD
Multiple Data	SIMI	MIMD

Dabei wird die Von-Neumann-Architektur als *SISD-Architektur* und die PRAM-Architektur als *SIMD-Architektur* klassifiziert. Es ist so zu verstehen, dass ein einzelner Befehl auf verschiedene Daten angewendet wird.

Bei heutigen Multicore-Rechnern liegt MIMD vor.

MISD ist nicht so richtig sinnvoll.

Definition 29 (Nick's Class)
Nick's Class (in Zeichen: $\mathcal{NC}$) ist die Klasse aller Probleme, die im PRAM-Modell in logarithmischer Laufzeit lösbar sind.

Beispiel 13 (Nick's Class)
Folgende Probleme sind in $\mathcal{NC}$:

1) Die Addition, Multiplikation und Division von Ganzzahlen,

2) Matrixmultiplikation, die Berechnung von Determinanten und Inversen,

3) ausschließlich Probleme aus $\mathcal{P}$, also: $\mathcal{NC} \subseteq \mathcal{P}$

Es ist nicht klar, ob $\mathcal{P} \subseteq \mathcal{NC}$ gilt. Bisher wurde also noch kein Problem $P \in \mathcal{P}$ gefunden mit $P \notin \mathcal{NC}$.

6.2 Prozesskommunikation

Die Prozesskommunikation wird durch einige Probleme erschwert:

Definition 30 (Wettlaufsituation)
Ist das Ergebnis einer Operation vom zeitlichen Ablauf der Einzeloperationen abhängig, so liegt eine Wettlaufsituation vor.

Beispiel 14 (Wettlaufsituation)
Angenommen, man hat ein Bankkonto mit einem Stand von 2000 Euro. Auf dieses Konto wird am Monatsende ein Gehalt von 800 Euro eingezahlt und die Miete von 600 Euro abgehoben. Nun stelle man sich folgende beiden Szenarien vor:

t	Prozess 1: Lohn	Prozess 2: Miete	Kontostand
1	Lade Kontostand	Lade Kontostand	2000
2	Addiere Lohn		2000
3	Speichere Kontostand		2800
4		Subtrahiere Miete	2800
5		Speichere Kontostand	1400

Dieses Problem existiert nicht nur bei echt parallelen Anwendungen, sondern auch bei zeitlich verzahnten Anwendungen.

Definition 31 (Semaphore)
Eine Semaphore $S = (c, r, f, L)$ ist eine Datenstruktur, die aus einer Ganzzahl, den beiden atomaren Operationen $r =$ „reservieren probieren“ und $f =$ „freigeben“ sowie einer Liste L besteht.

r gibt entweder Wahr oder Falsch zurück um zu zeigen, ob das reservieren erfolgreich war. Im Erfolgsfall wird c um 1 verringert. Es wird genau dann Wahr zurück gegeben, wenn c positiv ist. Wenn Wahr zurückgegeben wird, dann wird das aufrufende Objekt der Liste hinzugefügt.

f kann nur von Objekten aufgerufen werden, die in L sind. Wird f von $o \in L$ aufgerufen, wird o aus L entfernt und c um

eins erhöht.

Semaphoren können eingesetzt werden um Wettlaufsituationen zu verhindern.

Definition 32 (Monitor)
Ein Monitor $M = (m, c)$ ist ein Tupel, wobei m ein Mutex und c eine Bedingung ist.

Monitore können mit einer Semaphore, bei der $c = 1$ ist, implementiert werden. Monitore sorgen dafür, dass auf die Methoden der Objekte, die sie repräsentieren, zu jedem Zeitpunkt nur ein mal ausgeführt werden können. Sie sorgen also für *gegenseitigen Ausschluss*.

Beispiel 15 (Monitor)
Folgendes Beispiel von `https://en.wikipedia.org/w/index.php?title=Monitor_(synchronization)&oldid=596007585` verdeutlicht den Nutzen eines Monitors:

```
monitor class Account {
  private int balance := 0
  invariant balance >= 0

  public method boolean withdraw(int amount)
     precondition amount >= 0
  {
    if balance < amount:
        return false
    else:
        balance := balance - amount
        return true
  }

  public method deposit(int amount)
     precondition amount >= 0
  {
    balance := balance + amount
```

```
    }
}
```

6.3 Parallelität in Java

Java unterstützt mit der Klasse `Thread` und dem Interface `Runnable` Parallelität.

Interessante Stichwörder sind noch:

- ThreadPool
- Interface Executor
- Interface Future<V>
- Interface Callable<V>

7 Haskell

Haskell ist eine funktionale Programmiersprache, die von Haskell Brooks Curry entwickelt und 1990 in Version 1.0 veröffentlicht wurde.

Wichtige Konzepte sind:

1. Funktionen höherer Ordnung
2. anonyme Funktionen (sog. Lambda-Funktionen)
3. Pattern Matching
4. Unterversorgung
5. Typinferenz

Haskell kann mit „Glasgow Haskell Compiler“ mittels `ghci` interpretiert und mittels

7.1 Erste Schritte

Haskell kann unter `www.haskell.org/platform/` für alle Plattformen heruntergeladen werden. Unter Debian-Systemen ist das Paket `ghc` bzw. `haskell-platform` relevant.

7.1.1 Hello World

Speichere folgenden Quelltext als `hello-world.hs`:

hello-world.hs
```
1 main = putStrLn "Hello, World!"
```

Kompiliere ihn mit `ghc -o hello hello-world.hs`. Es wird eine ausführbare Datei erzeugt.

Alternativ kann es direkt mit `runghc hello-world.hs` ausgeführt werden.

7.2 Syntax

7.2.1 Klammern und Funktionsdeklaration

Haskell verzichtet an vielen Stellen auf Klammern. So werden im Folgenden die Funktionen $f(x) := \frac{\sin x}{x}$ und $g(x) := x \cdot f(x^2)$ definiert:

```
f :: Floating a => a -> a
f x = sin x / x

g :: Floating a => a -> a
g x = x * (f (x*x))
```

Die Funktionsdeklarationen mit den Typen sind nicht notwendig, da die Typen aus den benutzten Funktionen abgeleitet werden.

Zu lesen ist die Deklaration wie folgt:

```
[Funktionsname] :: [Typendefinitionen] =>
                   Signatur
```

T. Def. Die Funktion `f` benutzt als Parameter bzw. Rückgabewert einen Typen. Diesen Typen nennen wir `a` und er ist vom Typ `Floating`. Auch `b`, `wasweisich` oder etwas ähnliches wäre ok.

Signatur Die Signatur liest man am einfachsten von hinten:

- `f` bildet auf einen Wert vom Typ `a` ab und
- `f` hat genau einen Parameter `a`

Gibt es Funktionsdeklarationen, die bis auf Wechsel des Namens und der Reihenfolge äquivalent sind?

7.2.2 if / else

Das folgende Beispiel definiert den Binomialkoeffizienten (vgl. Beispiel 4.3)

```
binom :: (Eq a, Num a, Num a1) => a -> a -> a1
binom n k =
    if (k==0) || (k==n)
    then 1
    else binom (n-1) (k-1) + binom (n-1) k
```

Das könnte man auch mit sog. Guards machen:

```
binom :: (Eq a, Num a, Num a1) => a -> a -> a1
binom n k
    | (k==0) || (k==n) = 1
    | otherwise        = binom (n-1) (k-1)
                         + binom (n-1) k
```

7.2.3 Rekursion

Die Fakultätsfunktion wurde wie folgt implementiert:

$$fak(n) := \begin{cases} 1 & \text{falls } n = 0 \\ n \cdot fak(n) & \text{sonst} \end{cases}$$

```
fak :: (Eq a, Num a) => a -> a
fak n = if (n==0) then 1 else n * fak (n-1)
```

Diese Implementierung benötigt $\mathcal{O}(n)$ rekursive Aufrufe und hat einen Speicherverbrauch von $\mathcal{O}(n)$. Durch einen **Akkumulator** kann dies verhindert werden:

```
fakAcc :: (Eq a, Num a) => a -> a -> a
fakAcc n acc = if (n==0)
               then acc
               else fakAcc (n-1) (n*acc)

fak :: (Eq a, Num a) => a -> a
fak n = fakAcc n 1
```

7.2.4 Listen

- `[]` erzeugt die leere Liste,
- `[1,2,3]` erzeugt eine Liste mit den Elementen $1, 2, 3$
- `:` wird **cons** genannt und ist der Listenkonstruktor.
- `list !! i` gibt das i-te Element von `list` zurück.
- `head list` gibt den Kopf von `list` zurück, `tail list` den Rest:

```
Prelude> head []
*** Exception: Prelude.head: empty list
Prelude> tail []
*** Exception: Prelude.tail: empty list
Prelude> tail [1]
[]
Prelude> head [1]
1
Prelude> null []
True
Prelude> null [[]]
False
```

- `null list` prüft, ob `list` leer ist.
- `length list` gibt die Anzahl der Elemente in `list` zurück.
- `maximum [1,9,1,3]` gibt 9 zurück (analog: `minimum`).
- `last [1,9,1,3]` gibt 3 zurück.
- `reverse [1,9,1,3]` gibt `[3,1,9,1]` zurück.
- `elem item list` gibt zurück, ob sich `item` in `list` befindet.

Beispiel in der interaktiven Konsole

```
Prelude> let mylist = [1,2,3,4,5,6]
Prelude> head mylist
1
Prelude> tail mylist
[2,3,4,5,6]
Prelude> take 3 mylist
[1,2,3]
Prelude> drop 2 mylist
[3,4,5,6]
Prelude> mylist
[1,2,3,4,5,6]
Prelude> mylist ++ sndList
[1,2,3,4,5,6,9,8,7]
```

List-Comprehensions

List-Comprehensions sind kurzschreibweisen für Listen, die sich an der Mengenschreibweise in der Mathematik orientieren. So entspricht die Menge

$$myList = \{\, 1, 2, 3, 4, 5, 6 \,\}$$

$$test = \{ x \in myList \mid x > 2 \}$$

in etwa folgendem Haskell-Code:

```
Prelude> let mylist = [1,2,3,4,5,6]
Prelude> let test = [x | x <- mylist, x>2]
Prelude> test
[3,4,5,6]
```

7.2.5 Strings

- Strings sind Listen von Zeichen:
  `tail ÄBCDEF"` gibt `"BCDEF"` zurück.

7.3 Typen

7.3.1 Standard-Typen

Haskell kennt einige Basis-Typen:

- **Int**: Ganze Zahlen. Der Zahlenbereich kann je nach Implementierung variieren, aber der Haskell-Standart garantiert, dass das Intervall $[-2^{29}, 2^{29} - 1]$ abgedeckt wird.
- **Integer**: beliebig große ganze Zahlen
- **Float**: Fließkommazahlen
- **Double**: Fließkommazahlen mit doppelter Präzision
- **Bool**: Wahrheitswerte
- **Char**: Unicode-Zeichen

Des weiteren gibt es einige strukturierte Typen:

- Listen: z. B. $[1, 2, 3]$
- Tupel: z. B. $(1, 'a', 2)$

- Brüche (Fractional, RealFrac)
- Summen-Typen: Typen mit mehreren möglichen Repräsentationen

7.3.2 Typinferenz

In Haskell werden Typen aus den Operationen geschlossfolgert. Dieses Schlussfolgern der Typen, die nicht explizit angegeben werden müssen, nennt man **Typinferent**.

Haskell kennt die Typen aus Abb. 7.1.

Ein paar Beispiele zur Typinferenz:

```
Prelude> let x = \x -> x*x
Prelude> :t x
x :: Integer -> Integer
Prelude> x(2)
4
Prelude> x(2.2)
<interactive>:6:3:
    No instance for (Fractional Integer)
      arising from the literal `2.2'
    Possible fix: add an instance declaration for
                        (Fractional Integer)
    In the first argument of `x', namely `(2.2)'
    In the expression: x (2.2)
    In an equation for `it': it = x (2.2)


Prelude> let mult = \x y->x*y
Prelude> mult(2,5)
<interactive>:9:5:
    Couldn't match expected type `Integer' with
                actual type `(t0, t1)'
    In the first argument of `mult', namely `(2, 5)'
```

```
    In the expression: mult (2, 5)
    In an equation for 'it': it = mult (2, 5)
Prelude> mult 2 5
10
Prelude> :t mult
mult :: Integer -> Integer -> Integer

Prelude> let concat = \x y -> x ++ y
Prelude> concat [1,2,3] [3,2,1]
[1,2,3,3,2,1]
Prelude> :t concat
concat :: [a] -> [a] -> [a]
```

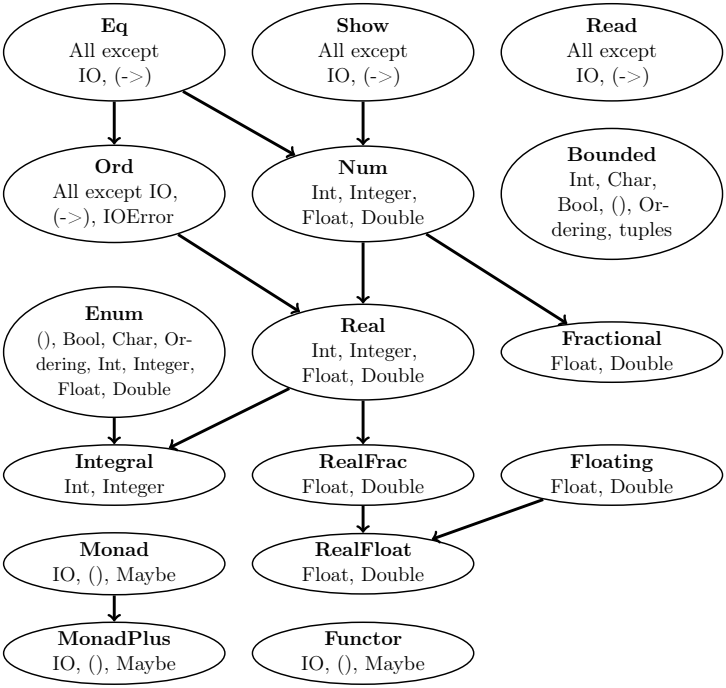

Abbildung 7.1: Hierarchie der Haskell Standardklassen

7.3.3 type

Mit `type` können Typsynonyme erstellt werden:

```
type Prename = String
type Age = Double
type Person = (Prename, Age)
type Friends = [Person]
```

7.3.4 data

Mit dem Schlüsselwort `data` können algebraische Datentypen erzeugt werden:

7.4 Lazy Evaluation

Haskell wertet Ausdrücke nur aus, wenn es nötig ist.

Beispiel 16 (Lazy Evaluation)
Obwohl der folgende Ausdruck einen Teilausdruck hat, der einen Fehler zurückgeben würde, kann er aufgrund der Lazy Evaluation zu 2 evaluiert werden:

lazy-evaluation.hs

```
g a b c
    | c > 0     = b
    | otherwise = a

main = do
    print (g (1/0) 2 3)
```

Ein spezialfall der Lazy-Evaluation ist die sog. *Kurzschlussauswertung*. Das bezeichnet die Lazy-Evaluation von booleschen Ausdrücken.

7.5 Beispiele

7.5.1 Quicksort

qsort.hs

```
1 qsort []     = []
2 qsort (p:ps) = (qsort (filter (\x -> x<=p) ps))
3           ++ p:(qsort (filter (\x -> x> p) ps))
```

- Die leere Liste ergibt sortiert die leere Liste.
- Wähle das erste Element `p` als Pivotelement und teile die restliche Liste `ps` in kleinere und gleiche sowie in größere Elemente mit `filter` auf. Konkateniere diese beiden Listen mit `++`.

Durch das Ausnutzen von Unterversorgung lässt sich das ganze sogar noch kürzer schreiben:

qsort.hs

```
1 qsort []     = []
2 qsort (p:ps) = (qsort (filter (<=p) ps))
3           ++ p:(qsort (filter (> p) ps))
```

7.5.2 Fibonacci

fibonacci.hs

```
1 fib n
2     | (n == 0) = 0
3     | (n == 1) = 1
4     | otherwise = fib (n - 1) + fib (n - 2)
```

fibonacci-akk.hs

```
1 fibAkk n n1 n2
2     | (n == 0) = n1
```

```
3     | (n == 1) = n2
4     | otherwise = fibAkk (n - 1) n2 (n1 + n2)
5 fib n = fibAkk n 0 1
```

fibonacci-zip.hs

```
1 fib = 0 : 1 : zipWith (+) fibs (tail fibs)
```

fibonacci-pattern-matching.hs

```
1 fib 0 = 0
2 fib 1 = 1
3 fib n = fib (n - 1) + fib (n - 2)
```

7.5.3 Quicksort

7.5.4 Funktionen höherer Ordnung

folds.hs

```
1 summer :: [Int] -> Int
2 summer = foldr (-) 0
3
4 summel :: [Int] -> Int
5 summel = foldl (-) 0
6
7 main :: IO ()
8 main = do
9     print (summer [1,2,3])
10    -- 0-(1-(2-3)) = 0-(1-(-1)) = 2
11    print (summel [1,2,3])
12    -- ((0-1)-2)-3 = -6
```

7.6 Weitere Informationen

- `hackage.haskell.org/package/base-4.6.0.1`: Referenz
- `haskell.org/hoogle`: Suchmaschine für das Haskell-Manual
- `wiki.ubuntuusers.de/Haskell`: Hinweise zur Installation von Haskell unter Ubuntu

8 Prolog

Prolog ist eine Programmiersprache, die das logische Programmierparadigma befolgt.

Eine interaktive Prolog-Sitzung startet man mit `swipl`.

In Prolog definiert man Terme.

8.1 Erste Schritte

8.1.1 Hello World

Speichere folgenden Quelltext als `hello-world.pl`:

Kompiliere ihn mit `gplc hello-world.pl`. Es wird eine ausführbare Datei erzeugt.

8.2 Syntax

8.3 Beispiele

8.3.1 Humans

Erstelle folgende Datei:

`human.pro`

```
1 human(bob).
2 human(socrates).
3 human(antonio).
```

Kompiliere diese mit

```
$ swipl -c human.pro
% library(swi_hooks) compiled into pce_swi_hooks
%                 0.00 sec, 2,224 bytes
% human.pro compiled 0.00 sec, 644 bytes
% /usr/lib/swi-prolog/library/listing compiled into
%                 prolog_listing 0.00 sec, 21,648 bytes
```

Dabei wird eine `a.out` Datei erzeugt, die man wie folgt nutzen kann:

```
$ ./a.out
Welcome to SWI-Prolog (Multi-threaded, 32 bits, Version
Copyright (c) 1990-2011 University of Amsterdam, VU Amste
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is fre
software, and you are welcome to redistribute it under ce
conditions. Please visit http://www.swi-prolog.org for de

For help, use ?- help(Topic). or ?- apropos(Word).

?- human(socrates).
true.
```

8.3.2 Zebrarätsel

Folgendes Rätsel wurde von `https://de.wikipedia.org/w/index.php?title=Zebrar%C3%A4tsel&oldid=126585006` entnommen:

1. Es gibt fünf Häuser.
2. Der Engländer wohnt im roten Haus.

3. Der Spanier hat einen Hund.
4. Kaffee wird im grünen Haus getrunken.
5. Der Ukrainer trinkt Tee.
6. Das grüne Haus ist direkt rechts vom weißen Haus.
7. Der Raucher von Altem-Gold-Zigaretten hält Schnecken als Haustiere.
8. Die Zigaretten der Marke Kools werden im gelben Haus geraucht.
9. Milch wird im mittleren Haus getrunken.
10. Der Norweger wohnt im ersten Haus.
11. Der Mann, der Chesterfields raucht, wohnt neben dem Mann mit dem Fuchs.
12. Die Marke Kools wird geraucht im Haus neben dem Haus mit dem Pferd.
13. Der Lucky-Strike-Raucher trinkt am liebsten Orangensaft.
14. Der Japaner raucht Zigaretten der Marke Parliaments.
15. Der Norweger wohnt neben dem blauen Haus.

Wer trinkt Wasser? Wem gehört das Zebra?

zebraraetsel.pro

```
1 Street=[Haus1,Haus2,Haus3],
2 mitglied(haus(rot,_,_),Street),
3 mitglied(haus(blau,_,_),Street),
4 mitglied(haus,(grün,_,_),Street),
5 mitglied(haus(rot,australier,_),Street),
6 mitglied(haus(_,italiener,tiger),Street),
7 sublist(haus(_,_,eidechse),haus(_,chinese,_),Street),
8 sublist(haus(blau,_,_),haus(_,_,eidechse),Street),
9 mitglied(haus(_,N,nilpferd),Street).
```

8.4 Weitere Informationen

- `wiki.ubuntuusers.de/Prolog`: Hinweise zur Installation von Prolog unter Ubuntu

9 Scala

Scala ist eine funktionale Programmiersprache, die auf der JVM aufbaut und in Java Bytecode kompiliert wird. Scala bedeutet scalable language.

Mit sog. „actors“ bietet Scala eine Unterstützung für die Entwicklung prallel ausführender Programme.

Weitere Materialien sind unter `http://www.scala-lang.org/` und `http://www.simplyscala.com/` zu finden.

9.1 Erste Schritte

Scala kann auf Debian-basierten Systemen durch das Paket `scala` installiert werden.

9.1.1 Hello World

Interaktiv

Folgendes Beispiel stammt von `http://wiki.ubuntuusers.de/Scala`.

scala-test.scala

```
1 #!/usr/bin/env scala
2 !#
3 def promptprint (s: String) = {println ("> " + s)}
4
```

```
println ("Hallo ")
args foreach promptprint
```

Es kann mit `./scala-test.scala Scala funktioniert` ausgeführt werden.

Kompiliert

hello-world.scala

```
object HelloWorld {
    def main(args: Array[String]) {
        println("Hello World!")
    }
}
```

Dieses Beispiel kann mit `scalac hello-world.scala` kompiliert und mit `scala HelloWorld` ausgeführt werden.

9.2 Vergleich mit Java

Scala und Java haben einige Gemeinsamkeiten, wie den Java Bytecode, aber auch einige Unterschiede.

Gemeinsamkeiten

- Java Bytecode
- Keine Mehrfachvererbung
- Statische Typisierung
- Scopes

Unterschiede

- Java hat Interfaces, Scala hat traits.
- Java hat primitive Typen, Scala ausschließlich Objekte.
- Scala benötigt kein `;` am Ende von Anweisungen.

Weitere Informationen hat Graham Lea unter `http://grahamhackingscala.blogspot.de/2009/11/scala-under-hood-of-hello-world.html` zur Verfügung gestellt.

9.3 Syntax

In Scala gibt es sog. *values*, die durch das Schlüsselwort `val` angezeigt werden. Diese sind Konstanten. Die Syntax ist der UML-Syntax ähnlich.

```
val name: type = value
```

Variablen werden durch das Schlüsselwort `var` angezeigt:

```
var name: type = value
```

Methoden werden mit dem Schlüsselwort `def` erzeugt:

```
def name(parameter: String): Unit = { code body... }
```

9.4 Beispiele

10 X10

10.1 Syntax

10.2 Beispiele

11 C

C ist eine imperative Programmiersprache. Sie wurde in vielen Standards definiert. Die wichtigsten davon sind:

Wo sind unterschiede?

- C89
- C99
- ANSI C
- C11

11.1 Datentypen

Die grundlegenden C-Datentypen sind

Typ	**Größe**
char	1 Byte
int	4 Bytes
float	4 Bytes
double	8 Bytes
void	0 Bytes

zusätzlich kann man `char` und `int` noch in `signed` und `unsigned` unterscheiden.

Dez.	Z.	Dez.	Z.	Dez.	Z.	Dez.	Z.
0		31		64	@	96	’
1				65	A	97	a
2				66	B	98	b
3					C	99	c
4					D	100	d
5					E		
6					F		
7					G		
8					H		
9					I		
10							
11							
12							
13							
14							
15							
16							
17							
18							
19							
20							
21							
22							
23							
24							
25							
26							
27							
28							
29							
31						127	

11.2 ASCII-Tabelle

11.3 Syntax

11.4 Beispiele

11.4.1 Hello World

Speichere den folgenden Text als `hello-world.c`:

hello-world.c

```
1 #include <stdio.h>
2
3 int main(void)
4 {
5     printf("Hello, World\n");
6     return 0;
7 }
```

Compiliere ihn mit `gcc hello-world.c`. Es wird eine ausführbare Datei namens `a.out` erzeugt.

12 MPI

Message Passing Interface (kurz: MPI) ist ein Standard, der den Nachrichtenaustausch bei parallelen Berechnungen auf verteilten Computersystemen beschreibt.

12.1 Syntax

12.2 Beispiele

13 Compilerbau

Wenn man über Compiler redet, meint man üblicherweise „vollständige Übersetzer“:

Definition 33

Ein **Compiler** ist ein Programm C, das den Quelltext eines Programms A in eine ausführbare Form übersetzen kann.

Jedoch gibt es verschiedene Ebenen der Interpretation bzw. Übersetzung:

1. **Reiner Interpretierer**: TCL, Unix-Shell
2. **Vorübersetzung**: Java-Bytecode, Pascal P-Code, Python[1], Smalltalk-Bytecode
3. **Laufzeitübersetzung**: JavaScript[2]
4. **Vollständige Übersetzung**: C, C++, Fortran

Zu sagen, dass Python eine interpretierte Sprache ist, ist in etwa so korrekt wie zu sagen, dass die Bibel ein Hardcover-Buch ist.[3]

Reine Interpretierer lesen den Quelltext Anweisung für Anweisung und führen diese direkt aus.

Bild

[1] Python hat auch `.pyc`-Dateien, die Python-Bytecode enthalten.

[2] JavaScript wird nicht immer zur Laufzeit übersetzt. Früher war es üblich, dass JavaScript nur interpretiert wurde.

[3] Quelle: stackoverflow.com/a/2998544, danke Alex Martelli für diesen Vergleich.

Bei der *Interpretation nach Vorübersetzung* wird der Quelltext analysiert und in eine für den Interpretierer günstigere Form übersetzt. Das kann z. B. durch

?

- Zuordnung Bezeichnergebrauch - Vereinbarung
- Transformation in Postfixbaum
- Typcheck, wo statisch möglich

geschehen. Diese Vorübersetzung ist nicht unbedingt maschinennah.

Bild

Die *Just-in-time-Compiler* (kurz: JIT-Compiler) betreiben Laufzeitübersetzung. Folgendes sind Vor- bzw. Nachteile von Just-in-time Compilern:

- schneller als reine Interpretierer
- Speichergewinn: Quelle kompakter als Zielprogramm
- Schnellerer Start des Programms
- Langsamer (pro Funktion) als vollständige Übersetzung
- kann dynamisch ermittelte Laufzeiteigenschaften berücksichtigen (dynamische Optimierung)

Was ist hier gemeint?

Moderne virtuelle Maschinen für Java und für .NET nutzen JIT-Compiler.

Bei der *vollständigen Übersetzung* wird der Quelltext vor der ersten Ausführung des Programms A in Maschinencode (z. B. x86, SPARC) übersetzt.

Bild

13.1 Funktionsweise

Üblicherweise führt ein Compiler folgende Schritte aus:

1. Lexikalische Analyse
2. Syntaktische Analyse
3. Semantische Analyse
4. Zwischencodeoptimierung
5. Codegenerierung
6. Assemblieren und Binden

13.2 Lexikalische Analyse

In der lexikalischen Analyse wird der Quelltext als Sequenz von Zeichen betrachtet. Sie soll bedeutungstragende Zeichengruppen, sog. *Tokens*, erkennen und unwichtige Zeichen, wie z. B. Kommentare überspringen. Außerdem sollen Bezeichner identifiziert und in einer *Stringtabelle* zusammengefasst werden.

Beispiel 17

Beispiel erstellen

13.2.1 Reguläre Ausdrücke

Beispiel 18 (Regulärere Ausdrücke)

Folgender regulärer Ausdruck erkennt Float-Konstanten in C nach ISO/IEC 9899:1999 §6.4.4.2:

$((0|\ldots|9)^*.(0|\ldots|9)^+)|((0|\ldots|9)^+.)$

Satz 13.1
Jede reguläre Sprache wird von einem (deterministischen) endlichen Automaten akzeptiert.

TODO: Bild einfügen

Zu jedem regulären Ausdruck im Sinne der theoretischen Informatik kann ein nichtdeterministischer Automat generiert werden. Dieser kann mittels Potenzmengenkonstruktion[4] in einen deterministischen Automaten überführen. Dieser kann dann mittels Äquivalenzklassen minimiert werden.

Alle Schritte beschreiben

13.2.2 Lex

Lex ist ein Programm, das beim Übersetzerbau benutzt wird um Tokenizer für die lexikalische Analyse zu erstellen. Flex ist eine Open-Source Variante davon.

Eine Flex-Datei besteht aus 3 Teilen, die durch `%%` getrennt werden:

```
Definitionen: Definiere Namen
%%
Regeln: Definiere reguläre Ausdrücke und
        zugehörige Aktionen (= Code)
%%
Code: zusätzlicher Code
```

x	Zeichen 'x' erkennen
"xy"	Zeichenkette 'xy' erkennen
\	Zeichen 'x' erkennen (TODO)
$[xyz]$	Zeichen x, y oder z erkennen
$[a-z]$	Alle Kleinbuchstaben erkennen
$[-z]$	Alle Zeichen außer Kleinbuchstaben erkennen
$x\|y$	x oder y erkennen
(x)	x erkennen
x*	0, 1 oder mehrere Vorkommen von x erkennen
x+	1 oder mehrere Vorkommen von x erkennen
x?	0 oder 1 Vorkommen von x erkennen
{Name}	Expansion der Definition Name
\t, \n, \rq	Tabulator, Zeilenumbruch, Wagenrücklauf erkennen

Reguläre Ausdrücke in Flex

13.3 Syntaktische Analyse

In der syntaktischen Analyse wird überprüft, ob die Tokenfolge zur kontextfreien Sprachegehört. Außerdem soll die hierarchische Struktur der Eingabe erkannt werden.

Warum kontextfrei?

Ausgegeben wird ein **abstrakter Syntaxbaum**.

Beispiel 19 (Abstrakter Syntaxbaum)
TODO

Was ist gemeint?

13.4 Semantische Analyse

Die semantische Analyse arbeitet auf einem abstrakten Syntaxbaum und generiert einen attributierten Syntaxbaum.

Sie führt eine kontextsensitive Analyse durch. Dazu gehören:

[4] http://martin-thoma.com/potenzmengenkonstruktion/

- **Namensanalyse**: Beziehung zwischen Deklaration und Verwendung

?

- **Typanalyse**: Bestimme und prüfe Typen von Variablen, Funktionen, ...

?

- **Konsistenzprüfung**: Wurden alle Einschränkungen der Programmiersprache eingehalten?

?

Beispiel 20 (Attributeriter Syntaxbaum)
TODO

13.5 Zwischencodeoptimierung

Hier wird der Code in eine sprach- und zielunabhängige Zwischensprache transformiert. Dabei sind viele Optimierungen vorstellbar. Ein paar davon sind:

- **Konstantenfaltung**: Ersetze z. B. $3 + 5$ durch 8.
- **Kopienfortschaffung**: Setze Werte von Variablen direkt ein
- **Code verschieben**: Führe Befehle vor der Schleife aus, statt in der Schleife

?

- **Gemeinsame Teilausdrücke entfernen**: Es sollen doppelte Berechnungen vermieden werden

Beispiel?

- **Inlining**: Statt Methode aufzurufen, kann der Code der Methode an der Aufrufstelle eingebaut werden.

13.6 Codegenerierung

Der letzte Schritt besteht darin, aus dem generiertem Zwischencode den Maschinencode oder Assembler zu erstellen. Dabei muss folgendes beachtet werden:

- **Konventionen**: Wie werden z. B. im Laufzeitsystem Methoden aufgerufen?
- **Codeauswahl**: Welche Befehle kennt das Zielsystem?
- **Scheduling**: In welcher Reihenfolge sollen die Befehle angeordnet werden?
- **Registerallokation**: Welche Zwischenergebnisse sollen in welchen Prozessorregistern gehalten werden?
- **Nachoptimierung**

Bildquellen

Abb. **??** S^2: Tom Bombadil, tex.stackexchange.com/a/42865

Abkürzungsverzeichnis

Beh. Behauptung

Bew. Beweis

bzgl. bezüglich

bzw. beziehungsweise

ca. circa

d. h. das heißt

DEA Deterministischer Endlicher Automat

etc. et cetera

ggf. gegebenenfalls

sog. sogenannte

Vor. Voraussetzung

vgl. vergleiche

z. B. zum Beispiel

z. z. zu zeigen

Ergänzende Definitionen

Definition 34 (Quantoren)

a) $\forall x \in X : p(x)$: Für alle Elemente x aus der Menge X gilt die Aussage p.

b) $\exists x \in X : p(x)$: Es gibt mindestens ein Element x aus der Menge X, für das die Aussage p gilt.

c) $\exists! x \in X : p(x)$: Es gibt genau ein Element x in der Menge X, sodass die Aussage p gilt.

Definition 35 (Prädikatenlogik)
Eine Prädikatenlogik ist ein formales System, das Variablen und Quantoren nutzt um Aussagen zu formulieren.

Definition 36 (Aussagenlogik)
TODO

Definition 37 (Grammatik)
Eine (formale) **Grammatik** ist ein Tupel (Σ, V, P, S) wobei gilt:

(i) Σ ist eine endliche Menge und heißt **Alphabet**,

(ii) V ist eine endliche Menge mit $V \cap \Sigma = \emptyset$ und heißt **Menge der Nichtterminale**,

(iii) $S \in V$ heißt das **Startsymbol**

(iv) $P = \{ p : I \to r \mid I \in (V \cup \Sigma)^+, r \in (V \cup \Sigma)^* \}$ ist eine endliche Menge aus **Produktionsregeln**

Man schreibt:

- $a \Rightarrow b$: Die Anwendung einer Produktionsregel auf a ergibt b.
- $a \Rightarrow^* b$: Die Anwendung mehrerer (oder keiner) Produktionsregeln auf a ergibt b.
- $a \Rightarrow^+ b$: Die Anwendung mindestens einer Produktionsregel auf a ergibt b.

Beispiel 21 (Formale Grammatik)
Folgende Grammatik $G = (\Sigma, V, P, A)$ erzeugt alle korrekten Klammerausdrücke:

- $\Sigma = \{ (,) \}$
- $V = \{ \alpha \}$
- $s = \alpha$
- $P = \{ \alpha \to () \mid \alpha\alpha \mid (\alpha) \}$

Definition 38 (Kontextfreie Grammatik)
Eine Grammatik (Σ, V, P, S) heißt **kontextfrei**, wenn für jede Produktion $p : I \to r$ gilt: $I \in V$.

Definition 39 (Sprache)
Sei $G = (\Sigma, V, P, S)$ eine Grammatik. Dann ist

$$L(G) := \{ \omega \in \Sigma^* \mid S \Rightarrow^* \omega \}$$

die Menge aller in der Grammatik ableitbaren Wörtern. $L(G)$ heißt Sprache der Grammatik G.

Definition 40
Sei $G = (\Sigma, V, P, S)$ eine Grammatik und $a \in (V \cup \Sigma)^+$.

a) $\Rightarrow_L$ heißt **Linksableitung**, wenn die Produktion auf das linkeste Nichtterminal angewendet wird.

b) $\Rightarrow_R$ heißt **Rechtsableitung**, wenn die Produktion auf das rechteste Nichtterminal angewendet wird.

Beispiel 22 (Links- und Rechtsableitung)

Sie G wie zuvor die Grammatik der korrekten Klammerausdrücke:

$$
\begin{array}{lcl}
\alpha \Rightarrow_L \alpha\alpha & \iff & \alpha \Rightarrow_R \alpha\alpha \\
\Rightarrow_L \alpha\alpha\alpha & & \Rightarrow_R \alpha\alpha\alpha \\
\Rightarrow_L ()\alpha\alpha & & \Rightarrow_R \alpha\alpha() \\
\Rightarrow_L ()(\alpha)\alpha & & \Rightarrow_R \alpha(\alpha)() \\
\Rightarrow_L ()(())\alpha & & \Rightarrow_R \alpha(())() \\
\Rightarrow_L ()(())() & & \Rightarrow_R ()(())()
\end{array}
$$

Definition 41 (LL(k)-Grammatik)

Sei $G = (\Sigma, V, P, S)$ eine kontextfreie Grammatik. G heißt LL(k)-Grammatik für $k \in \mathbb{N}_{\geq 1}$, wenn jeder Ableitungsschritt durch die linkesten k Symbole der Eingabe bestimmt ist.

Was ist die Eingabe einer Grammatik?

Ein LL-Parser ist ein Top-Down-Parser liest die Eingabe von Links nach rechts und versucht eine Linksableitung der Eingabe zu berechnen. Ein LL(k)-Parser kann k Token vorausschauen, wobei k als *Lookahead* bezeichnet wird.

Satz .2

Für linksrekursive, kontextfreie Grammatiken G gilt:

$$\forall k \in \mathbb{N} : G \notin \mathrm{SLL}(k)$$

Symbolverzeichnis

Reguläre Ausdrücke

$\emptyset$ Leere Menge
ϵ Das leere Wort
α, β Reguläre Ausdrücke
$L(\alpha)$ Die durch α beschriebene Sprache
$L(\alpha|\beta) = L(\alpha) \cup L(\beta)$
$L(\alpha \cdot \beta) = L(\alpha) \cdot L(\beta)$
$\alpha^+ = L(\alpha)^+$ TODO: Was ist $L(\alpha)^+$
$\alpha^* = L(\alpha)^*$ TODO: Was ist $L(\alpha)^*$
$\perp$ Bottom

Stichwortverzeichnis