Teil VI

Grammatiken

29. Knobelaufgabe #15

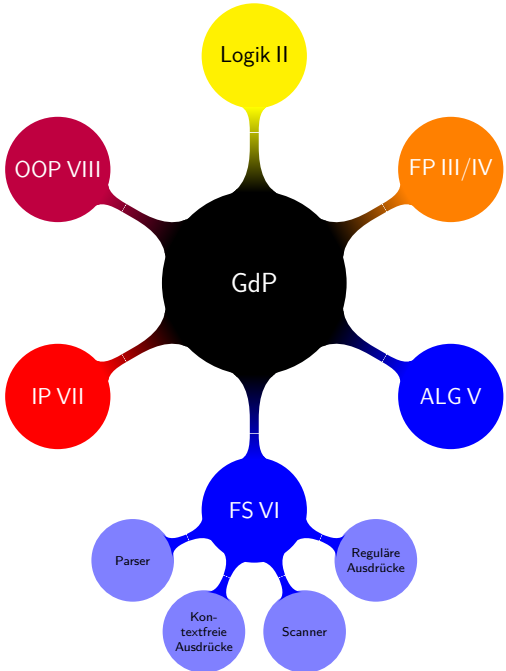
Ist es möglich, ein *nicht-rekursives* Programm zu schreiben, das *nicht terminiert*?

Nicht-rekursiv bedeutet, dass in dem Programm keine rekursiven Funktionsdefinitionen

$$\textbf{\textit{let rec}}\ f\ (x_1 : t_1) : t_2 = \ldots f \ldots$$

verwendet werden dürfen.

29. Gliederung


Logik II
OOP VIII
FP III/IV
GdP
IP VII
ALG V
FS VI
Parser
Reguläre Ausdrücke
Kon-textfreie Ausdrücke
Scanner

29. Lernziele

Nach Durcharbeitung dieses Kapitels sollten Sie

- den Unterschied zwischen konkreter und abstrakter Syntax verstanden haben,
- Syntax und Semantik regulärer Ausdrücke kennen,
- aus einem regulären Ausdruck einen Akzeptor konstruieren können,
- den Unterschied zwischen Interpreter und Übersetzer verstanden haben,
- Syntax und Semantik kontextfreier Ausdrücke kennen,
- (aus einem kontextfreien Ausdruck einen Parser konstruieren können,)
- Möglichkeiten und Grenzen der Sprachfamilien kennen.

29. Überblick

Dieses Kapitel erzählt eine der großen Erfolgsgeschichten der Informatik: die automatische Umsetzung einer deskriptiven Beschreibung („was“) in ein ausführbares Programm („wie“).

Wir führen Formalismen ein, mit denen man die konkrete Syntax einer Programmiersprache beschreiben kann:

- die lexikalische Syntax und
- die kontextfreie Syntax.

29. Wiederholung: Lexikalische Syntax

Betrachten wir ein einfaches Beispielprogramm:

```
4711* (a11 (* speed *) + 815 )
```

Mikroskopisch gesehen besteht das Programm aus einer Folge von Zeichen:

- der Ziffer 4,
- gefolgt von der Ziffer 7,
- gefolgt von der Ziffer 1,
- gefolgt von der Ziffer 1,
- gefolgt von einem Asteriskus *,
- gefolgt von einem Leerzeichen usw.

29. Wiederholung: Lexeme

Als menschlicher Leser sind wir gewohnt — bzw. durch jahrelanges Training geschult — mehrere Zeichen zu einer Einheit zusammenzufassen.

4711	*	(	a11	+	815	)

☞ Nicht alle Zeichen sind für den Rechner gedacht: `(* speed *)` ist ein Kommentar, der sich an den menschlichen Leser richtet.

In der *lexikalischen Syntax* einer Programmiersprache wird festgelegt, wie Zeichen zu größeren Einheiten, sogenannten *Lexemen* (engl. tokens), zusammengefasst werden.

29. Wiederholung: Kontextfreie Syntax

Nicht alle Folgen von Lexemen stellen ein gültiges Programm dar:

```
) * 4711 815 + ( a11
```

umfasst die gleichen Lexeme, ist aber *kein* Mini–F# Programm.

In der *kontextfreien Syntax* einer Programmiersprache wird festgelegt, welche Folgen von Lexemen gültige Programme sind und welche nicht.

Die lexikalische und die kontextfreie Syntax bilden zusammen die *konkrete Syntax* einer Programmiersprache.

30. Lexeme in Mini–F#

- *Numerale:* ein Numeral in Mini–F# besteht aus einer nicht-leeren Folge von Dezimalziffern.
- *Bezeichner:* ein Bezeichner in Mini–F# beginnt mit einem Buchstaben, gefolgt von weiteren Buchstaben, Ziffern und Sonderzeichen, wie einem Unterstrich oder einem Apostroph.

☞ Wie können wir den Aufbau von Numeralen und Bezeichnern präzise beschreiben?

☞ Wir müssen uns eine Sprache ausdenken — eine Sprache, um Sprachen zu beschreiben.

30. Begriffe

- Ein *Alphabet* A ist eine Menge von Zeichen.
  Beispiele:
  - $\{\mathtt{a}, \mathtt{b}\}$,
  - $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, (,), *, +\}$,
  - ASCII-Alphabet,
  - Menge aller Mini–F#-Lexeme.
- Eine *Sprache* ist eine Teilmenge von A^*, der Menge der Sequenzen über A.
- Statt von Sequenzen spricht man auch von *Wörtern*.
- Die Menge aller Sprachen ist $\mathbb{P}(A^*)$.

30. Motivation

Eine Sprache ist eine Menge von Wörtern; auf Wörtern haben wir bereits einige Operationen eingeführt.

- das *leere Wort*: ϵ,
- die *Konkatenation* von Wörtern: $w_1\ w_2$,
- die *n-fache Wiederholung*: w^n.

☞ Verallgemeinern wir w^n zu einer beliebigen Wiederholung w^*, so können wir Numerale beschreiben:

$$\mathit{digit}\ \mathit{digit}^*$$

☞ *digit* steht für die Sprache der Ziffern. *Lies:* ein Numeral ist eine Ziffer gefolgt von einer beliebigen Folge von Ziffern.

30. Motivation

Wie können wir *digit* beschreiben? Eine Ziffer ist entweder 0, oder 1, oder ... Erfinden wir eine Notation für die Alternative, '|', dann ist

0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

die gesuchte Definition von *digit*.

Die lexikalische Syntax von Bezeichnern lässt sich auf ähnliche Art und Weise festlegen:

letter (*letter* | *digit* | _ | `)*

wobei *letter* durch a | ··· | z | A | ··· | Z gegeben ist.

30. Motivation

Jetzt können wir unser Routineprogramm abspulen!

Wir definieren die

- die abstrakte Syntax der Sprachbeschreibungssprache und
- die Semantik der Sprachbeschreibungssprache.

Mir schwirrt der Kopf, eine Sprachbeschreibungssprache! Drehen wir uns da nicht im Kreis? Ich meine, wir wollen doch die konkrete Syntax von Mini–F# festlegen und jetzt müssen wir zuerst die Syntax der Sprachbeschreibungssprache festlegen.

Aber, welche Alternativen gibt es? Willst Du die konkrete Syntax umgangssprachlich festlegen?

Ist mir schon bewusst, dass das keine gute Idee ist.

Wenn Du einen Formalismus verwendest, muss doch klar sein, was erlaubt ist und was gemeint ist.

Die Gründe liegen aber noch tiefer: Im Endeffekt wollen wir die Feststellung, ob ein Stück Text ein gültiges Mini–F# Programm ist, dem Rechner übertragen. Dazu muss eben Syntax und Semantik präzise festgelegt werden.

Mir schwant Böses: ein Mini–F# Programm, das überprüft, ob Mini–F# Programme korrekt sind ...

30. Abstrakte Syntax

$a \in A$	*Alphabet*
$r \in \mathrm{Reg} ::=$	*reguläre Ausdrücke:*
$\mid \epsilon$	das leere Wort
$\mid a$	einzelnes Zeichen $\backslash$ Terminalsymbol
$\mid r_1\ r_2$	Konkatenation $\backslash$ Sequenz
$\mid \emptyset$	die leere Sprache
$\mid r_1 \mid r_2$	Alternative
$\mid r^*$	Wiederholung

☞ Die Symbole ϵ und $\emptyset$ sind *überladen*: je nach Kontext bedeuten sie etwas Verschiedenes — ϵ steht auch für das leere Wort und $\emptyset$ für die leere Menge.

30. Konkrete Syntax

Zwei konkrete Beispiele für konkrete Syntaxen:

	Abstrakte Syntax	`grep`	`egrep`
das leere Wort	ϵ	–	–
einzelnes Zeichen	a	a	a
Konkatenation	$r_1\ r_2$	$r_1\ r_2$	$r_1\ r_2$
die leere Sprache	$\emptyset$	–	–
Alternative	$r_1 \mid r_2$	r_1 `|` r_2	r_1 `|` r_2
Wiederholung	r^*	r`\*`	r`*`
Gruppierung	–	`\(` r `\)`	`(` r `)`
beliebiges Zeichen	$a_1 \mid \cdots \mid a_n$	`.`	`.`
optionales Vorkommen	$\epsilon \mid r$	r`\?`	r`?`
positive Wiederholung	$r\ r^*$	r`\+`	r`+`

☞ `grep` und `egrep` durchsuchen Sequenzen (typischerweise Textdateien) nach Teilsequenzen, die auf ein bestimmtes Muster passen.
☞ `\` ist ein Fluchtsymbol.

30. Semantik

☞ Eine Aufteilung in statische und dynamische Semantik ist nicht notwendig: reguläre Ausdrücke sind beliebig kombinierbar.

Wir definieren zwei verschiedene Semantiken:

- *Denotationelle Semantik:* „Welche Sprache bezeichnet der reguläre Ausdruck?“.
- *Reduktionssemantik:* „Wie kann ich ein einzelnes Wort aus dem regulären Ausdruck ableiten?“.

30. Denotationelle Semantik

- Die Bedeutung von ϵ ist $\{\epsilon\}$.
- Die Bedeutung von a ist $\{a\}$, wobei a das Wort $\{0 \mapsto a\}$ abkürzt.
- *Zur Erinnerung:* Die Bedeutung eines regulären Ausdrucks ist eine Sprache, kein Wort.
- Wir haben die Konkatenation von Wörtern definiert; jetzt müssen wir die Konkatenation auf Mengen von Wörtern fortsetzen.

$$L_1 \cdot L_2 = \{ w_1 \cdot w_2 \mid w_1 \in L_1, w_2 \in L_2 \}$$

☞ Damit ist auch '$\cdot$' *überladen*.

- Beliebige Wiederholung einer Sprache:

$$L^* = \bigcup \{ L^n \mid n \in \mathbb{N} \}$$

mit $L^0 = \{\epsilon\}$ und $L^{n+1} = L \cdot L^n$.

30. Denotationelle Semantik

Die Semantik wird durch eine Bedeutungsfunktion angegeben:

$$
\begin{array}{ll}
[\![a]\!] & = \{\, a \,\} \\
[\![\epsilon]\!] & = \{\, \epsilon \,\} \\
[\![r_1\ r_2]\!] & = [\![r_1]\!] \cdot [\![r_2]\!] \\
[\![\emptyset]\!] & = \emptyset \\
[\![r_1 \mid r_2]\!] & = [\![r_1]\!] \cup [\![r_2]\!] \\
[\![r^*]\!] & = [\![r]\!]^*
\end{array}
$$

☞ Die doppelten Klammern sind die sogenannten *Oxford* oder *Strachey Klammern* — sie umschließen die Syntax.

30. Beispiele

Zugrundeliegendes Alphabet $A = \{\mathtt{a}, \mathtt{b}\}$.

$$
\begin{aligned}
& [\![\mathtt{a}\ \mathtt{b} \mid \mathtt{b}\ \mathtt{a}]\!] \\
= \ & [\![\mathtt{a}\ \mathtt{b}]\!] \cup [\![\mathtt{b}\ \mathtt{a}]\!] \\
= \ & ([\![\mathtt{a}]\!] \cdot [\![\mathtt{b}]\!]) \cup ([\![\mathtt{b}]\!] \cdot [\![\mathtt{a}]\!]) \\
= \ & (\{\mathtt{a}\} \cdot \{\mathtt{b}\}) \cup (\{\mathtt{b}\} \cdot \{\mathtt{a}\}) \\
= \ & \{\mathtt{ab}\} \cup \{\mathtt{ba}\} \\
= \ & \{\mathtt{ab}, \mathtt{ba}\}
\end{aligned}
$$

☞ Der reguläre Ausdruck $[\![\mathtt{a}\ \mathtt{b} \mid \mathtt{b}\ \mathtt{a}]\!]$ bezeichnet die Sprache $\{\mathtt{ab}, \mathtt{ba}\}$.

30. Beispiele

Vertauschen wir Konkatenation und Alternative erhalten wir eine andere Sprache:

$$
\begin{aligned}
& [\![(\mathtt{a} \mid \mathtt{b})\ (\mathtt{b} \mid \mathtt{a})]\!] \\
= \ & [\![\mathtt{a} \mid \mathtt{b}]\!] \cdot [\![\mathtt{b} \mid \mathtt{a}]\!] \\
= \ & ([\![\mathtt{a}]\!] \cup [\![\mathtt{b}]\!]) \cdot ([\![\mathtt{b}]\!] \cup [\![\mathtt{a}]\!]) \\
= \ & (\{\mathtt{a}\} \cup \{\mathtt{b}\}) \cdot (\{\mathtt{b}\} \cup \{\mathtt{a}\}) \\
= \ & \{\mathtt{a}, \mathtt{b}\} \cdot \{\mathtt{b}, \mathtt{a}\} \\
= \ & \{\mathtt{ab}, \mathtt{aa}, \mathtt{bb}, \mathtt{ba}\}
\end{aligned}
$$

☞ Der reguläre Ausdruck $(\mathtt{a} \mid \mathtt{b})\ (\mathtt{b} \mid \mathtt{a})$ bezeichnet die Sprache aller Wörter der Länge 2.

30. Beispiele

$$
\begin{aligned}
& [\![((\mathtt{a} \mid \mathtt{b})\ (\mathtt{b} \mid \mathtt{a}))^*]\!] \\
= \quad & [\![(\mathtt{a} \mid \mathtt{b})\ (\mathtt{b} \mid \mathtt{a})]\!]^* \\
= \quad & \{\mathtt{ab}, \mathtt{aa}, \mathtt{bb}, \mathtt{ba}\}^*
\end{aligned}
$$

☞ Die bezeichnete Sprache umfasst alle Wörter gerader Länge.

30. Reduktionssemantik

Zur Erinnerung:

- *Denotationelle Semantik:* „Welche Sprache bezeichnet der reguläre Ausdruck?“.

  ☞ Globale Sicht.

- *Reduktionssemantik:* „Wie kann ich ein einzelnes Wort aus dem regulären Ausdruck ableiten?“.

  ☞ Lokale Sicht.

30. Reduktionssemantik

☞ Um die Semantik von Mini–F# festzulegen, haben wir Werte eingeführt, das sind besonders einfache Ausdrücke.

☞ In Analogie dazu: für die Reduktionssemantik sind Wörter besonders einfache reguläre Ausdrücke.

$w ::= \epsilon$	das leere Wort
$\mid a$	einzelnes Zeichen ≙ Terminalsymbol
$\mid w_1\ w_2$	Konkatenation ≙ Sequenz

☞ Ein Wort zeichnet sich dadurch aus, dass seine Denotation eine einelementige Menge ist.

30. Reduktionssemantik

Die schrittweise Reduktion $r \longrightarrow r'$ (lies: r kann in einem Schritt zu r' reduziert werden) wird durch ein Beweissystem formalisiert.

Rechenregeln:

$$\frac{}{r\ \epsilon \longrightarrow r} \qquad \frac{}{\epsilon\ r \longrightarrow r}$$

$$\frac{}{r_1 \mid r_2 \longrightarrow r_1} \qquad \frac{}{r_1 \mid r_2 \longrightarrow r_2}$$

$$\frac{}{r^* \longrightarrow \epsilon} \qquad \frac{}{r^* \longrightarrow r\ r^*}$$

☞ Die letzten vier Regeln formalisieren Wahlmöglichkeiten.

30. Beispiele

Für den regulären Ausdruck `a b | b a` existieren zwei mögliche Reduktionsfolgen.

`a b | b a` $\longrightarrow$ `a b`

`a b | b a` $\longrightarrow$ `b a`

☞ Die Sprache, die durch einen regulären Ausdruck beschrieben wird, ist die Menge aller *Worte*, die aus dem Ausdruck ableitbar sind. Für unser Beispiel: $\{\mathtt{ab}, \mathtt{ba}\}$.

30. Reduktionssemantik

Um den Ausdruck (a | b) (b | a) zu vereinfachen, benötigen wir weitere Regeln. Die obigen Beweisregeln erlauben nur den *gesamten* Ausdruck umzuformen; wir brauchen also Regeln, die es uns ermöglichen, einen Ausdruck „mittendrin“ zu manipulieren.

Kongruenz- oder *Kontextregeln*:

$$\frac{r_1 \longrightarrow r_1'}{r_1\ r_2 \longrightarrow r_1'\ r_2} \qquad \frac{r_2 \longrightarrow r_2'}{r_1\ r_2 \longrightarrow r_1\ r_2'}$$

30. Beispiele

Für $(a \mid b)\ (b \mid a)$ gibt es insgesamt vier mögliche Reduktionsfolgen.

$$(a \mid b)\ (b \mid a) \longrightarrow a\ (b \mid a) \longrightarrow a\ b$$
$$(a \mid b)\ (b \mid a) \longrightarrow a\ (b \mid a) \longrightarrow a\ a$$
$$(a \mid b)\ (b \mid a) \longrightarrow b\ (b \mid a) \longrightarrow b\ b$$
$$(a \mid b)\ (b \mid a) \longrightarrow b\ (b \mid a) \longrightarrow b\ a$$

☞ Die bezeichnete Sprache ist somit $\{ab, aa, bb, ba\}$.

30. Beispiele

Wesentlich mehr Möglichkeiten gibt es für $((a \mid b)\ (b \mid a))^*$, nämlich unendlich viele.

$$
\begin{aligned}
((a \mid b)\ (b \mid a))^* &\longrightarrow \epsilon \\
((a \mid b)\ (b \mid a))^* &\longrightarrow (a \mid b)\ (b \mid a)\ ((a \mid b)\ (b \mid a))^* \\
&\longrightarrow a\ (b \mid a)\ ((a \mid b)\ (b \mid a))^* \\
&\longrightarrow a\ b\ ((a \mid b)\ (b \mid a))^* \\
&\longrightarrow a\ b\ \epsilon \\
&\longrightarrow a\ b \\
&\ldots
\end{aligned}
$$

30. Beispiele

Zur Erinnerung: Die Sprache, die durch einen regulären Ausdruck beschrieben wird, ist die Menge aller *Worte*, die aus dem Ausdruck ableitbar sind.

$$\emptyset^* \longrightarrow \emptyset\ \emptyset^* \longrightarrow \emptyset\ \emptyset\ \emptyset^* \longrightarrow \cdots$$

☞ Da es keine Regel gibt, $\emptyset$ zu reduzieren, lässt sich mit dieser Reduktionsfolge kein Wort ableiten.

Die einzige mögliche Reduktionsfolge ist

$$\emptyset^* \longrightarrow \epsilon$$

☞ Somit ist die Semantik von $\emptyset^*$ die Sprache $\{\epsilon\}$. (Auch mit der denotationellen Semantik erhalten wir $[\![\emptyset^*]\!] = \{\epsilon\}$.)

30. Vertiefung

Eine Sprache kann in der Regel auf verschiedene Art und Weisen beschrieben werden.

Beispiel: Sprache aller Wörter mit einer geraden Anzahl von as und beliebig vielen bs.

$$(\mathtt{a}\ \mathtt{b}^*\ \mathtt{a} \mid \mathtt{b})^*$$

Alternative Formulierung:

$$(\mathtt{b} \mid \mathtt{a}\ \mathtt{b}^*\ \mathtt{a})^*$$

Alternative Formulierung ohne '|':

$$\mathtt{b}^*\ (\mathtt{a}\ \mathtt{b}^*\ \mathtt{a}\ \mathtt{b}^*)^*$$

☞ Allgemein gilt: $(r_1 \mid r_2)^* = r_1^*\ (r_2\ r_1^*)^*$.

30. Gleichheit regulärer Ausdrücke

☞ Wie zeigt man, dass zwei reguläre Ausdrücke r_1 und r_2 gleichwertig sind?

Mit Hilfe der Semantik:

- *Denotationelle Semantik:* $[\![r_1]\!] = [\![r_2]\!]$;
- also: Rückgriff auf die (naive) Mengenlehre;
- *Reduktionssemantik:* zu jeder Reduktionsfolge $r_1 \longrightarrow \cdots \longrightarrow w$ gibt es eine korrespondierende Reduktionsfolge $r_2 \longrightarrow \cdots \longrightarrow w$;
- also: Rückgriff auf Beweissysteme (Logik).

30. Vertiefung

Die Sprache aller Wörter, die eine gerade Anzahl von as (bs beliebig) *oder* eine ungerade Anzahl von bs (as beliebig) haben, lässt sich unter Rückgriff auf das vorherige Beispiel (Folie 573) leicht definieren.

$$(\mathtt{a}\ \mathtt{b}^*\ \mathtt{a} \mid \mathtt{b})^* \quad \mid \quad \mathtt{a}^*\ \mathtt{b}\ (\mathtt{a} \mid \mathtt{b}\ \mathtt{a}^*\ \mathtt{b})^*$$

☞ Aber wie beschreibt man die Sprache aller Wörter, die eine gerade Anzahl von as (bs beliebig) *und* eine ungerade Anzahl von bs (as beliebig) haben?

30. Vertiefung

Stände ein Pendant zum Mengendurchschnitt bereit, sagen wir &, könnten wir formulieren:

```
(a b* a | b)* & a* b (a | b a* b)*
```

☞ Es spricht nichts dagegen, den Durchschnitt zu der Sprache der regulären Ausdrücke hinzuzunehmen.
☞ Interessanterweise ist die Erweiterung zwar bequem, aber nicht notwendig: sie erhöht nicht die Ausdruckskraft.

Versuchen wir also die Sprache mit den bisherigen Bordmitteln zu definieren ...

30. Vertiefung

Nützliche Abkürzungen: $g = (\mathtt{aa})^*$ und $u = \mathtt{a}\,(\mathtt{aa})^*$.

1. Schritt: genau ein b und eine gerade Anzahl von as.

$$g\ \mathtt{b}\ g \mid u\ \mathtt{b}\ u$$

☞ Entweder kommen vor und hinter dem b eine gerade Anzahl von as vor oder an beiden Positionen eine ungerade Anzahl.

2. Schritt: wir erhöhen die Anzahl der bs auf zwei:

$$g\ \mathtt{b}\ g\ \mathtt{b}\ g \mid g\ \mathtt{b}\ u\ \mathtt{b}\ u \mid u\ \mathtt{b}\ g\ \mathtt{b}\ u \mid u\ \mathtt{b}\ u\ \mathtt{b}\ g$$

Jetzt werden die as über drei Positionen verteilt; entweder befinden sich an allen Stellen eine gerade Anzahl von as oder an exakt zwei Positionen eine ungerade Anzahl.

30. Vertiefung

3. Schritt: eine ungerade Zahl hat die Form $1 + 2n$, entsprechend definieren wir

$$(g\ \mathsf{b}\ g \mid u\ \mathsf{b}\ u)\ \ (g\ \mathsf{b}\ g\ \mathsf{b}\ g \mid g\ \mathsf{b}\ u\ \mathsf{b}\ u \mid u\ \mathsf{b}\ g\ \mathsf{b}\ u \mid u\ \mathsf{b}\ u\ \mathsf{b}\ g)^*$$

☞ Man sieht, ohne den Durchschnitt müssen wir bei der Formulierung von Sprachen sehr viel mehr Grips investieren.

30. Grenzen regulärer Ausdrücke

- Können wir mit regulären Ausdrücken alle Sprachen beschreiben?
- Nein, es gibt sehr viel mehr Sprachen als Ausdrücke, mit denen Sprachen beschrieben werden können (überabzählbar versus abzählbar).
- Können wir mit regulären Ausdrücken die Syntax von Programmiersprachen festlegen?
- Nein, einfache Hygienevorschriften lassen sich nicht fassen: etwa, dass es in Mini–F# Ausdrücken zu jeder „Klammer auf“ eine korrespondierende „Klammer zu“ geben muss. Die Sprache

$$\{\mathtt{a}^n\ \mathtt{b}^n \mid n \in \mathbb{N}\}$$

ist nicht regulär — lies a als (und b als) .

30. Lösung Knobelaufgabe #4

Endlich!!

Also: das sichselbstausgebende Programm muss mindestens einen String enthalten; ein String alleine reicht aber nicht, es muss noch etwas dazukommen. *Idee:* der String enthält den Programmcode des anderen Teils; damit das Programm sich selbst ausgibt, muss der String verdoppelt werden. Etwa so:

$$s \mathbin{\hat{}} \mathit{show}\ s$$

Warum einmal s und einmal $\mathit{show}\ s$?

s steht für den Programmcode und $\mathit{show}\ s$ für den Text des Programmcodes.

$$(\textbf{\textit{fun}}\ s \to s \mathbin{\hat{}} \mathit{show}\ s)\ \texttt{"(fun s -> s ^ show s)"}$$

30. Lösung Knobelaufgabe #4 — Demo

```
Mini⟩ "Hello, world!"
"Hello, world!"
Mini⟩ show "Hello, world!"
"\"Hello, world!\""
Mini⟩ length "Hello, world!"
13
Mini⟩ length (show "Hello, world!")
15
Mini⟩ (fun s → s ^ show s) "(fun s -> s ^ show s)"
"(fun s -> s ^ show s)\"(fun s -> s ^ show s)\""
Mini⟩ putline it
(fun s → s ^ show s) "(fun s -> s ^ show s)"
```