

32. Knobelaufgabe #17

Welche Sprache bezeichnet der folgende reguläre Ausdruck?

$$riddle = (a a \mid b b)^* ((a b \mid b a) (a a \mid b b)^* (a b \mid b a) (a a \mid b b)^*)^*$$

Und welche Sprachen bezeichnen $riddle / a$ und $riddle / b$?

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation

Abstrakte Syntax

Reduktionsemantik

Denotationelle
Semantik

Vertiefung

Parser »

622

32. Motivation

- ▶ Reguläre Ausdrücke sind wenig ausdrucksstark — die jeweiligen Akzeptoren haben nur ein *endliches* Gedächtnis.
- ▶ Hygienevorschriften wie „zu jeder offenen Klammer muss es eine schließende Klammer geben“ lassen sich nicht formulieren.

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

- ▶ Die Sprache der wohlgeformten Klammerausdrücke lässt sich aber schrittweise (im Fachjargon: induktiv) definieren:
 - ▶ ϵ ist wohlgeformt;
 - ▶ wenn w wohlgeformt ist, dann auch $a w b$.
- ▶ Wir benötigen *Rekursion!*
- ▶ Wir erweitern reguläre Ausdrücke um rekursiv definierte Sprachen:

$$\mathit{rec} x \rightarrow \epsilon \mid a x b$$

Lies: wenn x ein Element der Sprache ist, dann auch ϵ und $a x b$.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation

Abstrakte Syntax

Reduktionsemantik

Denotationelle
Semantik

Vertiefung

Parser »

623

32. Motivation

- ▶ Die Erweiterung bekommt einen neuen Namen: wir sprechen von *kontextfreien Ausdrücken* oder *kontextfreien Grammatiken*.
- ▶ Es gibt auch *kontextsensitive Sprachen*, mehr zu diesem Thema später aus der Abteilung der Theoretischen Informatik.
- ▶ Zum Begriff „kontextfrei“:

$$\mathit{rec} \text{ expr} \rightarrow 0 \mid \text{false} \mid \text{expr} + \text{expr}$$

Die Grammatik erlaubt einen Booleschen Ausdruck, etwa *false*, in einem Kontext zu verwenden, in dem ein arithmetischer Ausdruck erwartet wird: *false + 0*.

- ▶ Die Einschränkung auf wohlgetypte Ausdrücke können wir *nicht* mit der kontextfreien Syntax ausdrücken.
- ▶ Um diese Dinge kümmert sich bei uns die statische Semantik.
- ▶ Durch diese klare Trennung von Zuständigkeiten wird die Sprachdefinition von Mini-F# ungemein erleichtert.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation

Abstrakte Syntax

Reduktionsemantik

Denotationelle
Semantik

Vertiefung

Parser »

624

32. Abstrakte Syntax

$x \in \text{Id}$

$c \in \text{CF} ::=$

	a
	x
	ϵ
	$c_1 c_2$
	\emptyset
	$c_1 \mid c_2$
	$\mathit{rec} x \rightarrow c$

kontextfreie Ausdrücke:

Terminalsymbol
Bezeichner \ Nichtterminalsymbol
das leere Wort
Konkatenation \ Sequenz
die leere Sprache
Alternative
Rekursion

☞ *Zusätzliche Konstrukte:* Bezeichner und Rekursion.

☞ Die Wiederholung ist kein primitives Konzept mehr; sie kann mit Hilfe der Rekursion ausgedrückt werden: c^* durch $\mathit{rec} x \rightarrow \epsilon \mid c x$ oder $\mathit{rec} x \rightarrow \epsilon \mid x c$.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation

Abstrakte Syntax

Reduktionsemantik

Denotationelle
Semantik

Vertiefung

Parser »

625

32. Reduktionssemantik

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation

Abstrakte Syntax

Reduktionssemantik

Denotationelle
Semantik

Vertiefung

Parser*

Teaser: Was ist die Bedeutung von

- ▶ $rec\ x \rightarrow x,$
- ▶ $rec\ x \rightarrow a\ x\ b,$
- ▶ $rec\ x \rightarrow x^*,$
- ▶ $rec\ x \rightarrow x^*\ a,$
- ▶ $rec\ x \rightarrow rec\ y \rightarrow \epsilon \mid a \mid x\ y?$

☞ Semantik ist insbesondere dazu da, die Bedeutung von Randfällen oder Extremfällen zu klären.

626

32. Reduktionssemantik — Rechenregel

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation

Abstrakte Syntax

Reduktionssemantik

Denotationelle
Semantik

Vertiefung

Parser*

Rechenregel:

$$\overline{(rec\ x \rightarrow c) \rightarrow c\{x \mapsto rec\ x \rightarrow c\}}$$

☞ Ein rekursiver Ausdruck wird einmal „aufgefaltet“: der Bezeichner x wird im Rumpf c durch den gesamten Ausdruck $rec\ x \rightarrow c$ ersetzt.

627

32. Reduktionssemantik — Beispiel

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation

Abstrakte Syntax

Reduktionssemantik

Denotationelle
Semantik

Vertiefung

Parser*

Mit Hilfe der Rechenregel können wir zum Beispiel $aabb$ aus dem Ausdruck $rec\ x \rightarrow \epsilon \mid a\ x\ b$ ableiten.

$$\begin{aligned} & rec\ x \rightarrow \epsilon \mid a\ x\ b \\ \rightarrow & \epsilon \mid a\ (rec\ x \rightarrow \epsilon \mid a\ x\ b)\ b \\ \rightarrow & a\ (rec\ x \rightarrow \epsilon \mid a\ x\ b)\ b \\ \rightarrow & a\ (\epsilon \mid a\ (rec\ x \rightarrow \epsilon \mid a\ x\ b)\ b)\ b \\ \rightarrow & a\ (a\ (rec\ x \rightarrow \epsilon \mid a\ x\ b)\ b)\ b \\ \rightarrow & a\ (a\ (\epsilon \mid a\ (rec\ x \rightarrow \epsilon \mid a\ x\ b)\ b)\ b)\ b \\ \rightarrow & a\ (a\ \epsilon\ b)\ b \\ \rightarrow & a\ a\ b\ b \end{aligned}$$

628

32. Reduktionssemantik — Substitution

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation

Abstrakte Syntax

Reduktionssemantik

Denotationelle
Semantik

Vertiefung

Parser*

Die Ersetzung von Bezeichnern durch Ausdrücke nennt man *Substitution*.

☞ In c werden die Bezeichner aus $dom\ \sigma$ mit $\sigma \in Id \rightarrow_{fin} CF$ durch die zugeordneten Ausdrücke ersetzt.

Beispiel:

$$\begin{aligned} & (g\ b\ g\ \mid\ u\ b\ u)\{g \mapsto (aa)^*, u \mapsto a\ (aa)^*\} \\ & = (aa)^*\ b\ (aa)^*\ \mid\ a\ (aa)^*\ b\ a\ (aa)^* \end{aligned}$$

629

32. Reduktionssemantik — Substitution

Substitution von Bezeichnern in Ausdrücken:

$$\begin{aligned}
 a\sigma &= a \\
 x\sigma &= \begin{cases} \sigma(x), & \text{if } x \in \text{dom}(\sigma) \\ x, & \text{otherwise} \end{cases} \\
 \epsilon\sigma &= \epsilon \\
 (c_1 c_2)\sigma &= (c_1\sigma) (c_2\sigma) \\
 \emptyset\sigma &= \emptyset \\
 (c_1 \mid c_2)\sigma &= (c_1\sigma) \mid (c_2\sigma) \\
 (\mathit{rec } x \rightarrow c)\sigma &= \mathit{rec } x \rightarrow c(\sigma \setminus \{x\})
 \end{aligned}$$

☞ $\sigma \setminus \{x\}$ ist die *Einschränkung* von σ auf $\text{dom } \sigma \setminus \{x\}$. Wir müssen sicherstellen, dass gebundene Variablen, z.B. x in $\mathit{rec } x \rightarrow c$, *nicht* ersetzt werden.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation
Abstrakte Syntax

Reduktionssemantik

Denotationelle
Semantik

Vertiefung

Parser»

630

32. Reduktionssemantik — Teaser

Wie sieht es mit dem Ausdruck $\mathit{rec } x \rightarrow x$ aus?

$$\begin{aligned}
 &\mathit{rec } x \rightarrow x \\
 \rightarrow &\mathit{rec } x \rightarrow x \\
 &\vdots
 \end{aligned}$$

☞ Wir machen keinen Fortschritt!

☞ Wir können kein *Wort* ableiten; $\mathit{rec } x \rightarrow x$ bezeichnet die leere Sprache.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation
Abstrakte Syntax

Reduktionssemantik

Denotationelle
Semantik

Vertiefung

Parser»

631

32. Reduktionssemantik — Teaser

Wie sieht es mit dem Ausdruck $\mathit{rec } x \rightarrow a x b$ aus?

$$\begin{aligned}
 &\mathit{rec } x \rightarrow a x b \\
 \rightarrow &a (\mathit{rec } x \rightarrow a x b) b \\
 \rightarrow &a a (\mathit{rec } x \rightarrow a x b) b b \\
 &\vdots
 \end{aligned}$$

☞ Der Ausdruck wird immer größer!

☞ Wir können wiederum kein *Wort* ableiten; auch $\mathit{rec } x \rightarrow a x b$ bezeichnet die leere Sprache.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation
Abstrakte Syntax

Reduktionssemantik

Denotationelle
Semantik

Vertiefung

Parser»

632

32. Reduktionssemantik — Teaser

Wie sieht es mit dem Ausdruck $\mathit{rec } x \rightarrow x^*$ aus?

$$\begin{aligned}
 &\mathit{rec } x \rightarrow x^* \\
 \rightarrow &(\mathit{rec } x \rightarrow x^*)^* \\
 \rightarrow &\epsilon
 \end{aligned}$$

☞ Ein anderes *Wort* lässt sich nicht ableiten, also steht $\mathit{rec } x \rightarrow x^*$ für die Sprache $\{\epsilon\}$.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation
Abstrakte Syntax

Reduktionssemantik

Denotationelle
Semantik

Vertiefung

Parser»

633

32. Denotationelle Semantik

Bei der Motivation von $rec\ x \rightarrow \epsilon \mid a\ x\ b$ haben wir überlegt, wie sich $a^n\ b^n$ schrittweise bilden lässt: aus ϵ wird ab , daraus wird $aabb$ usw.

☞ Dieser Bildungsprozess lässt sich auch auf die *Sprache als Ganzes* übertragen.

Aus dem Ausdruck $rec\ x \rightarrow \epsilon \mid a\ x\ b$ leiten wir zunächst eine Funktion auf Sprachen ab.

$$F(L) = \{\epsilon\} \cup \{a\} \cdot L \cdot \{b\}$$

☞ Die mathematische Funktion F hat den Typ $\mathbb{P}(A^*) \rightarrow \mathbb{P}(A^*)$.

☞ F beschreibt einen Schritt des Bildungsprozesses.

32. Denotationelle Semantik

Jetzt können wir ausgehend von der leeren Menge durch wiederholte Anwendung von F uns der Semantik von $rec\ x \rightarrow \epsilon \mid a\ x\ b$ nähern.

$$\begin{aligned} \emptyset &= \emptyset \\ F(\emptyset) &= \{\epsilon\} \cup \{a\} \cdot \emptyset \cdot \{b\} = \{\epsilon\} \\ F(F(\emptyset)) &= \{\epsilon\} \cup \{a\} \cdot \{\epsilon\} \cdot \{b\} = \{\epsilon, ab\} \\ F(F(F(\emptyset))) &= \{\epsilon\} \cup \{a\} \cdot \{\epsilon, ab\} \cdot \{b\} = \{\epsilon, ab, aabb\} \\ F(F(F(F(\emptyset)))) &= \{\epsilon\} \cup \{a\} \cdot \{\epsilon, ab, aabb\} \cdot \{b\} \\ &= \{\epsilon, ab, aabb, aaabbb\} \end{aligned}$$

☞ Wir nähern uns der Semantik, da wir in endlich vielen Schritten niemals die unendliche Menge $\{a^n\ b^n \mid n \in \mathbb{N}\}$ konstruieren können.

32. Denotationelle Semantik — Fixpunkte

Die Bedeutung eines rekursiven Ausdrucks ergibt sich als Vereinigung aller endlichen Approximationen.

$$\bigcup \{F^n(\emptyset) \mid n \in \mathbb{N}\}$$

☞ $F^n(\emptyset)$ ist die n -fache Anwendung der Funktion F auf \emptyset : $F^0(X) = X$ und $F^{n+1}(X) = F(F^n(X))$.

Eigenschaften von $S = \bigcup \{F^n(\emptyset) \mid n \in \mathbb{N}\}$:

- ▶ Die Sprache S ist ein *Fixpunkt* von F : es gilt $F(S) = S$.
- ▶ Die Sprache ist die *kleinste* Menge mit dieser Eigenschaft; S ist der *kleinste Fixpunkt*.
- ▶ Kurz: Alles notwendige ist drin, mehr aber nicht.

32. Denotationelle Semantik — Bezeichner

☞ Bevor wir die semantischen Gleichungen formulieren, müssen wir uns noch um die Bedeutung von Bezeichnern kümmern.

Deren Bedeutung halten wir in einer *Umgebung* fest.

$$\varrho \in \text{Id} \rightarrow_{\text{fin}} \mathbb{P}(A^*)$$

☞ Den Begriff Umgebung haben wir schon bei der Auswertung von Mini-F# Ausdrücken verwendet.

- ▶ Dort: Abbildung von Bezeichnern auf Werte.
- ▶ Hier: Abbildung von Bezeichnern auf Sprachen.

32. Denotationelle Semantik — semantische Gleichungen

Die Semantikfunktion bildet einen kontextfreien Ausdruck und eine Umgebung auf eine Sprache ab.

$$\begin{aligned} \llbracket a \rrbracket \varrho &= \{a\} \\ \llbracket x \rrbracket \varrho &= \varrho(x) \\ \llbracket \epsilon \rrbracket \varrho &= \{\epsilon\} \\ \llbracket c_1 c_2 \rrbracket \varrho &= \llbracket c_1 \rrbracket \varrho \cdot \llbracket c_2 \rrbracket \varrho \\ \llbracket () \rrbracket \varrho &= \emptyset \\ \llbracket c_1 \mid c_2 \rrbracket \varrho &= \llbracket c_1 \rrbracket \varrho \cup \llbracket c_2 \rrbracket \varrho \\ \llbracket \text{rec } x \rightarrow c \rrbracket \varrho &= \bigcup \{F^n(\emptyset) \mid n \in \mathbb{N}\} \text{ mit } F(X) = \llbracket c \rrbracket (\varrho, \{x \mapsto X\}) \end{aligned}$$

☞ $\text{rec } x \rightarrow c$ wird auf den kleinsten Fixpunkt der zugehörigen Funktion $F : \mathbb{P}(A^*) \rightarrow \mathbb{P}(A^*)$ abgebildet.

☞ Die Umgebung wird verwendet, um die Bedeutung von x zu klären; beim rec -Konstrukt wird die Umgebung erweitert: $'$ ist der bekannte und beliebte Kommaoperator.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation

Abstrakte Syntax

Reduktionsemantik

Denotationelle
Semantik

Vertiefung

Parser*

638

32. Denotationelle Semantik — Teaser

Die dem Ausdruck $\text{rec } x \rightarrow x$ zugeordnete Funktion ist

$$F(L) = L$$

Fixpunktiteration:

$$\begin{aligned} \emptyset \\ F(\emptyset) = \emptyset \end{aligned}$$

☞ Die Bedeutung von $\text{rec } x \rightarrow x$ ist die leere Sprache.

☞ Die Funktion F hat unendlich viele Fixpunkte — jede Sprache ist Fixpunkt dieser Funktion. Der kleinste Fixpunkt ist die leere Sprache.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation

Abstrakte Syntax

Reduktionsemantik

Denotationelle
Semantik

Vertiefung

Parser*

639

32. Denotationelle Semantik — Teaser

Die dem Ausdruck $\text{rec } x \rightarrow a \mid x \mid b$ zugeordnete Funktion ist

$$G(L) = \{a\} \cdot L \cdot \{b\}$$

Fixpunktiteration:

$$\begin{aligned} \emptyset \\ G(\emptyset) = \{a\} \cdot \emptyset \cdot \{b\} = \emptyset \end{aligned}$$

☞ Die Bedeutung von $\text{rec } x \rightarrow a \mid x \mid b$ ist ebenfalls die leere Sprache.

☞ Zum Knobeln: Wieviele Fixpunkte hat G ?

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation

Abstrakte Syntax

Reduktionsemantik

Denotationelle
Semantik

Vertiefung

Parser*

640

32. Denotationelle Semantik — Teaser

Die dem Ausdruck $\text{rec } x \rightarrow x^* \mid \epsilon$ zugeordnete Funktion ist

$$H(L) = L^*$$

Fixpunktiteration:

$$\begin{aligned} \emptyset \\ H(\emptyset) &= \emptyset^* = \{\epsilon\} \\ H(H(\emptyset)) &= \{\epsilon\}^* = \{\epsilon\} \end{aligned}$$

☞ Die Bedeutung von $\text{rec } x \rightarrow x^* \mid \epsilon$ ist $\{\epsilon\}$.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation

Abstrakte Syntax

Reduktionsemantik

Denotationelle
Semantik

Vertiefung

Parser*

641

32. Vertiefung

Im Folgenden spezifizieren wir schrittweise die kontextfreie Syntax von Mini-F#, eingeschränkt auf die in Teil III eingeführten Konstrukte.

Das zugrundeliegende Alphabet ist die Menge aller Mini-F# Lexeme. Wir verwenden

- ▶ *num* als Bezeichner für die Sprache aller Numerale,
- ▶ *id* steht für kleine Bezeichner und
- ▶ *Id* entsprechend für große Bezeichner.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation

Abstrakte Syntax

Reduktionsemantik

Denotationelle
Semantik

Vertiefung

Parser»

642

32. Vertiefung

Versuchen wir uns an der kontextfreien Syntax einfacher arithmetischer Ausdrücke.

$rec\ expr \rightarrow num \mid expr + expr \mid expr * expr$

☞ Der kontextfreie Ausdruck ist an die Baumsprache für Ausdrücke angelehnt: ein arithmetischer Ausdruck ist

- ▶ entweder ein Numeral,
- ▶ oder ein Ausdruck gefolgt von dem Symbol + gefolgt von einem weiteren Ausdruck,
- ▶ oder ein Ausdruck gefolgt von dem Symbol * gefolgt von einem weiteren Ausdruck.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation

Abstrakte Syntax

Reduktionsemantik

Denotationelle
Semantik

Vertiefung

Parser»

643

32. Vertiefung

Aus dem kontextfreien Ausdruck lässt sich das Wort $4711 + 815 * 2765$ ableiten — wir kürzen den Ausdruck mit *E* ab und führen der Übersichtlichkeit halber nicht alle Reduktionsschritte auf:

$E \rightarrow E + E$
 $\rightarrow num + E$
 $\rightarrow 4711 + E$
 $\rightarrow 4711 + E * E$
 $\rightarrow 4711 + num * E$
 $\rightarrow 4711 + 815 * E$
 $\rightarrow 4711 + 815 * num$
 $\rightarrow 4711 + 815 * 2765$

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation

Abstrakte Syntax

Reduktionsemantik

Denotationelle
Semantik

Vertiefung

Parser»

644

32. Vertiefung

Es gibt aber noch eine zweite mögliche Reduktionsfolge:

$E \rightarrow E * E$
 $\rightarrow E + E * E$
 $\rightarrow num + E * E$
 $\rightarrow 4711 + E * E$
 $\rightarrow 4711 + num * E$
 $\rightarrow 4711 + 815 * E$
 $\rightarrow 4711 + 815 * num$
 $\rightarrow 4711 + 815 * 2765$

☞ *Problem:* der obige kontextfreie Ausdruck ist *mehrdeutig*; ein Wort kann auf verschiedene Weisen abgeleitet werden.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation

Abstrakte Syntax

Reduktionsemantik

Denotationelle
Semantik

Vertiefung

Parser»

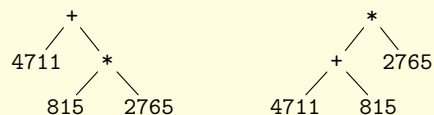
645

32. Vertiefung — mehrdeutige Grammatiken

☞ Warum ist das ein Problem?

Die kontextfreie Syntax dient einzig und allein dem Zweck, aus der linearen Folge von Lexemen die hierarchische Struktur eines Programms zu rekonstruieren.

Die beiden unterschiedlichen Reduktionsfolgen legen aber einen unterschiedlichen hierarchischen Aufbau nahe:



Die Semantik ordnet den beiden Syntaxbäumen eine unterschiedliche Bedeutung zu.

☞ Der obige kontextfreie Ausdruck ist ungeeignet zur Beschreibung arithmetischer Ausdrücke, da er nicht jedem Ausdruck einen eindeutigen abstrakten Syntaxbaum zuordnet.

VI Grammatiken

Ralf Hinze

Reguläre Ausdrücke

Scanner

Kontextfreie Ausdrücke

Motivation

Abstrakte Syntax

Reduktionssemantik

Denotationelle Semantik

Vertiefung

Parser »

646

32. Operatoren — Präfix- und Postfixnotation

Was ist zu tun?

Wir können die Syntax von Mini-F# Ausdrücken überdenken und arithmetische Operatoren nicht zwischen die Operatoren schreiben, sondern davor oder dahinter.

▶ *Präfixnotation* (wie in *Scheme*):

```
rec expr → num | + expr expr | * expr expr
```

▶ *Postfixnotation* (wie in *PostScript*):

```
rec expr → num | expr expr + | expr expr *
```

☞ Beide Syntaxen sind *eindeutig*. (Warum?)

VI Grammatiken

Ralf Hinze

Reguläre Ausdrücke

Scanner

Kontextfreie Ausdrücke

Motivation

Abstrakte Syntax

Reduktionssemantik

Denotationelle Semantik

Vertiefung

Parser »

647

32. Operatoren — Infixnotation

Die allermeisten Sprachen notieren aber — wie auch Mini-F# — Operatoren infix. Warum?

☞ Wahrscheinlich, weil sie der mathematischen Tradition folgen. Das ist natürlich nur eine halbwegs befriedigende Antwort, klärt sie doch nicht, warum die Notation tatsächlich sinnvoll ist.

Um den Gründen auf die Spur zu kommen, betrachten wir eine Summe aus drei Zahlen.

Infixnotation:

```
4711 + 815 + 2765
```

▶ *Präfixnotation:* + 4711 + 815 2765 und + + 4711 815 2765.

▶ *Postfixnotation:* 4711 815 + 2765 + und 4711 815 2765 + +.

☞ Semantisch sind beide Varianten gleich, da die Addition *assoziativ* ist. Präfix- und Postfixnotation machen einen Unterschied, wo es keinen gibt. Allein die Infix-Notation stellt uns nicht vor die Wahl.

VI Grammatiken

Ralf Hinze

Reguläre Ausdrücke

Scanner

Kontextfreie Ausdrücke

Motivation

Abstrakte Syntax

Reduktionssemantik

Denotationelle Semantik

Vertiefung

Parser »

648

32. Operatoren — Assoziativität

Assoziative Operatoren und Funktionen:

- ▶ Disjunktion: `||`,
- ▶ Konjunktion: `&&`,
- ▶ Addition: `+`,
- ▶ Multiplikation: `*`,
- ▶ Konkatenation von Strings: `~`,
- ▶ Minimum: `min`,
- ▶ Maximum: `max`,
- ▶ Konkatenation von Listen: `@`,
- ▶ der Kommaoperator: `','`.

☞ Nicht alle assoziativen Funktionen notieren wir tatsächlich infix, bei allen würde es sich aber anbieten.

VI Grammatiken

Ralf Hinze

Reguläre Ausdrücke

Scanner

Kontextfreie Ausdrücke

Motivation

Abstrakte Syntax

Reduktionssemantik

Denotationelle Semantik

Vertiefung

Parser »

649

32. Operatoren — Assoziativität

☞ Wo Licht ist, ist auch Schatten.

Es hat sich eingebürgert, auch *nicht assoziative* Funktionen infix zu notieren:

- ▶ Subtraktion:

$4711 - 815 - 2765$

☞ $- 4711 - 815 2765$ und $-- 4711 815 2765$ sind unterschiedlich. Die Infixschreibweise klärt nicht, welche Variante gemeint ist.

- ▶ Potenzfunktion (*power* (x, n) wird zu $x ** n$):

$4711 ** 815 ** 2765$

☞ Das gleiche Problem.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation

Abstrakte Syntax

Reduktionsemantik

Denotationelle
Semantik

Vertiefung

Parser»

650

32. Operatoren — Links- und Rechtsassoziierung

Es bedarf einer Festlegung:

- ▶ Die Subtraktion wird allgemein als *linksassoziierend* festgelegt:

$a - b - c$ steht für $-- a b c$

- ▶ Die Potenzfunktion wird allgemein als *rechtsassoziierend* festgelegt:

$a ** b ** c$ steht für $** a ** b c$

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation

Abstrakte Syntax

Reduktionsemantik

Denotationelle
Semantik

Vertiefung

Parser»

651

32. Operatoren — Klammern

Was machen wir, wenn wir die andere Variante benötigen, die wir infix nicht ausdrücken können?

In diesem Fall behilft man sich mit *Klammern*, die die Gruppierung explizit machen:

- ▶ Subtraktion:

$4711 - (815 - 2765)$

- ▶ Potenzfunktion:

$(4711 ** 815) ** 2765$

☞ Klammern sind ein Hilfsmittel der konkreten Syntax.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation

Abstrakte Syntax

Reduktionsemantik

Denotationelle
Semantik

Vertiefung

Parser»

652

32. Operatoren — Klammern

Mit Hilfe von Klammern lassen sich die Begriffe links- und rechtsassoziierend noch einmal verdeutlichen:

- ▶ Ist der Operator ' \oplus ' linksassoziierend, dann steht

$a \oplus b \oplus c$ für $(a \oplus b) \oplus c$

- ▶ ist der Operator rechtsassoziierend, dann steht

$a \oplus b \oplus c$ für $a \oplus (b \oplus c)$

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation

Abstrakte Syntax

Reduktionsemantik

Denotationelle
Semantik

Vertiefung

Parser»

653

32. Operatoren — Assoziativität

Zwischenfazit: die Infixschreibweise ist *semantisch motiviert*:

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

Bei nicht assoziativen Funktionen bedarf es einer *syntaktischen Festlegung*:

- ▶ linksassoziierend: $(a \oplus b) \oplus c$,
- ▶ rechtsassoziierend: $a \oplus (b \oplus c)$.

☞ Eine solche Festlegung kann auch für assoziative Funktionen sinnvoll sein: $xs_1 @ (xs_2 @ xs_3)$ und $(xs_1 @ xs_2) @ xs_3$ sind zwar gleichwertig, aber der erste Ausdruck ist schneller ausgerechnet. (Warum?)

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation

Abstrakte Syntax

Reduktionsemantik

Denotationelle
Semantik

Vertiefung

Parser »

654

32. Operatoren — Bindungsstärke

Bietet eine Sprache mehrere Operatoren an, dann muss man klären, was passiert, wenn zwei Operatoren aufeinandertreffen.

$$4711 + 815 * 2765$$

Ist damit

- ▶ $(4711 + 815) * 2765$ oder
- ▶ $4711 + (815 * 2765)$ gemeint?

☞ Gängige Konvention — Punkt- vor Strichrechnung — gibt der zweiten Alternative den Vorzug.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation

Abstrakte Syntax

Reduktionsemantik

Denotationelle
Semantik

Vertiefung

Parser »

655

32. Operatoren — Bindungsstärke

Hat man zwei beliebige Operatoren vor sich, \oplus und \otimes , so lässt sich der Konflikt mit Hilfe der sogenannten *Bindungsstärke* lösen.

$$a \oplus b \otimes c$$

Man stellt sich vor, dass \oplus und \otimes um den Operanden b streiten; der Operator mit der höheren Bindungsstärke zieht ihn an sich.

- ▶ Hat \oplus die höhere Bindungsstärke, dann ist gemeint

$$(a \oplus b) \otimes c$$

- ▶ Hat \otimes die höhere Bindungsstärke, dann entsprechend

$$a \oplus (b \otimes c)$$

☞ Die Bindungsstärke wird oft mit Hilfe natürlicher Zahlen spezifiziert, etwa '+' hat die Bindungsstärke 0 und '*' hat die Bindungsstärke 1.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation

Abstrakte Syntax

Reduktionsemantik

Denotationelle
Semantik

Vertiefung

Parser »

656

32. Operatoren — eindeutige Grammatiken

Zurück zu unserer Aufgabe, der Aufstellung einer Syntax für einfache arithmetische Ausdrücke.

Wir können die Grammatik eindeutig machen, indem wir Assoziierung und Bindungsstärke der Operatoren in die Beschreibung „hineinprogrammieren“.

```
rec expr0 → expr1 | expr1 + expr0
and expr1 → expr2 | expr2 * expr1
and expr2 → num | ( expr0 )
```

☞ Die Grammatik ist *verschränkt rekursiv*: die Bezeichner $expr_0$, $expr_1$ und $expr_2$ sind in allen rechten Seiten sichtbar.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation

Abstrakte Syntax

Reduktionsemantik

Denotationelle
Semantik

Vertiefung

Parser »

657

32. Operatoren — eindeutige Grammatiken

rec $expr_0 \rightarrow expr_1 \mid expr_1 + expr_0$
and $expr_1 \rightarrow expr_2 \mid expr_2 * expr_1$
and $expr_2 \rightarrow num \mid (expr_0)$

Idee: E_i umfasst nur Ausdrücke, deren oberster Operator eine Bindungsstärke von i oder mehr hat.

- ▶ E_0 umfasst alle Ausdrücke,
- ▶ E_1 umfasst nur Produkte und
- ▶ E_2 umfasst nur atomare oder geklammerte Ausdrücke.

☞ Die Grammatik legt + und * als rechtsassoziiierend fest. (Warum?)

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation

Abstrakte Syntax

Reduktionsemantik

Denotationelle
Semantik

Vertiefung

Parser »

658

32. Operatoren — eindeutige Grammatiken

Die Grammatik ist *eindeutig*: jeder arithmetische Ausdruck lässt sich auf genau eine Art und Weise ableiten.

Zum Beispiel:

$E_0 \rightarrow E_1 + E_0$
 $\rightarrow E_2 + E_0$
 $\rightarrow num + E_0$
 $\rightarrow 4711 + E_0$
 $\rightarrow 4711 + E_1$
 $\rightarrow 4711 + E_2 * E_1$
 $\rightarrow 4711 + num * E_1$
 $\rightarrow 4711 + 815 * E_1$
 $\rightarrow 4711 + 815 * E_2$
 $\rightarrow 4711 + 815 * num$
 $\rightarrow 4711 + 815 * 2765$

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation

Abstrakte Syntax

Reduktionsemantik

Denotationelle
Semantik

Vertiefung

Parser »

659

32. Operatoren — eindeutige Grammatiken

Fazit: die naheliegende Syntax für arithmetische Ausdrücke ist mehrdeutig.

Um die Syntax eindeutig zu machen, muss man Vereinbarungen über die Assoziierung (links- oder rechtsassoziiierend) und die Bindungsstärke (0 . . .) treffen.

Lässt man die Vereinbarungen in die Sprachbeschreibung einfließen,

- ▶ nimmt diese an Umfang zu und
- ▶ an Leserlichkeit ab.

☞ In der Praxis belässt man es oft bei der mehrdeutigen Syntax und führt die zusätzlichen Vereinbarungen getrennt davon auf. So werden wir es auch halten.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation

Abstrakte Syntax

Reduktionsemantik

Denotationelle
Semantik

Vertiefung

Parser »

660

32. Syntax von Mini-F#

Die Erweiterung von kontextfreien Ausdrücken um verschränkte Rekursion ist auch notwendig für die Beschreibung von *in*-Ausdrücken. Diese involvieren eine zweite syntaktische Kategorie: Definitionen.

rec $expr \rightarrow id \mid num \mid expr + expr \mid expr * expr \mid decl \text{ in } expr$
and $decl \rightarrow \text{let } id = expr$

Ausdrücke und Definitionen sind verschränkt rekursiv: Ausdrücke beinhalten Definitionen und umgekehrt.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Motivation

Abstrakte Syntax

Reduktionsemantik

Denotationelle
Semantik

Vertiefung

Parser »

661

32. Syntax von Mini-F# — Mehrdeutigkeiten

Auch diese Grammatik ist mehrdeutig: der **in**-Ausdruck

let $n = 4711$ **in** $n + n$

hat zwei mögliche Interpretationen:

- ▶ **(let** $n = 4711$ **in** n) $+ n$
- ▶ **let** $n = 4711$ **in** ($n + n$)

☞ Der Unterschied ist groß, meint doch das zweite Vorkommen von n in beiden Ausdrücken etwas anderes!

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke
Scanner
Kontextfreie
Ausdrücke
Motivation
Abstrakte Syntax
Reduktionsemantik
Denotationelle
Semantik
Vertiefung
Parser »

662

32. Syntax von Mini-F# — so weit nach rechts wie möglich

Metaregel: wir vereinbaren, dass sich ein **in**-Ausdruck *so weit nach rechts wie möglich erstreckt*.

let $n = 4711$ **in** $n + n$ steht für **let** $n = 4711$ **in** ($n + n$)

Genau wie die Vereinbarungen über Assoziierung und Bindungsstärke kann man auch diese Metaregel

- ▶ in die Syntax hineinprogrammieren oder
- ▶ als separate Bemerkung zur Sprachbeschreibung hinzufügen.

☞ Alternativ kann man das Ende des Sichtbarkeitsbereiches explizit markieren:

let $n = 4711$ **in** $n + n$ **end**.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke
Scanner
Kontextfreie
Ausdrücke
Motivation
Abstrakte Syntax
Reduktionsemantik
Denotationelle
Semantik
Vertiefung
Parser »

663

32. Syntax von Mini-F# — so weit nach rechts wie möglich

Die Vereinbarung „so weit nach rechts wie möglich“ wird in Mini-F# für zwei andere Konstrukte verwendet:

- ▶ Alternativen: **if** e_1 **then** e_2 **else** e_3 .
- ▶ Funktionsausdrücke (anonyme Funktionen): **fun** $x \rightarrow e$.

☞ Der Ausdruck **fun** $x \rightarrow x + x$ ist mehrdeutig: (**fun** $x \rightarrow x$) $+ x$ und **fun** $x \rightarrow (x + x)$ stehen als Interpretationen zur Wahl.

Der erste Ausdruck ist nicht typkorrekt, deswegen wird der zweiten Variante der Vorzug gegeben.

☞ Da die Metaregel „so weit nach rechts wie möglich“ *in der Regel* die sinnvolle Variante auswählt, wird das Ende von Alternativen und Funktionsausdrücken wie bei **in**-Ausdrücken nicht explizit markiert.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke
Scanner
Kontextfreie
Ausdrücke
Motivation
Abstrakte Syntax
Reduktionsemantik
Denotationelle
Semantik
Vertiefung
Parser »

664

32. Syntax von Mini-F# — Notation

Für die Syntaxbeschreibung von Mini-F# verwenden wir mehrere abkürzende Notationen:

- ▶ c^+ steht für eine mindestens einmalige Wiederholung von c : **rec** $x \rightarrow c \mid c x$;
- ▶ c_s^+ für eine mindestens einmalige Wiederholung, bei der die c Elemente durch s Elemente getrennt werden: **rec** $x \rightarrow c \mid c s x$;
- ▶ c_s^* für eine beliebige Wiederholung, bei der die c Elemente durch s Elemente getrennt werden: $\epsilon \mid c_s^+$.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke
Scanner
Kontextfreie
Ausdrücke
Motivation
Abstrakte Syntax
Reduktionsemantik
Denotationelle
Semantik
Vertiefung
Parser »

665

32. Syntax von Mini-F# — Grammatik

VI Grammatiken

Ralf Hinze

Reguläre Ausdrücke
Scanner
Kontextfreie Ausdrücke
Motivation
Abstrakte Syntax
Reduktionsemantik
Denotationelle Semantik
Vertiefung
Parser»

```
// Ausdrücke
rec expr → aexpr          atomarer Ausdruck
| if expr then expr else expr Alternative
| let decl* in expr        lokale Deklaration
| expr || expr             Disjunktion
| ...
| expr + expr             Addition
| ...
| fun apat+ -> expr        Funktionsabstraktion
| aexpr aexpr+            Funktionsapplikation
| expr : type              Typangabe

and aexpr → num           Numeral
| id                       Bezeichner
| ( expr )                 Gruppierung
```

Atomare Ausdrücke müssen niemals geklammert werden.

666

32. Syntax von Mini-F# — Grammatik

VI Grammatiken

Ralf Hinze

Reguläre Ausdrücke
Scanner
Kontextfreie Ausdrücke
Motivation
Abstrakte Syntax
Reduktionsemantik
Denotationelle Semantik
Vertiefung
Parser»

```
// Deklarationen
and decl → pat = expr      Wertedefinition
| funcdecl                 Funktionsdefinition
| rec fundecl+            rekursive Funktionsdefinitionen
and fundecl → id apat+ : type = expr Funktionsdefinition

// Muster
and pat → apat             atomares Muster
| pat : type               Typangabe

and apat → id              Bezeichner
| ( pat* )                 Tupelmuster oder Gruppierung

// Typausdrücke
and type → atype           atomarer Typ
| type * type              Tupeltyp
| type -> type             Funktionstyp
and atype → id             Typbezeichner
| ( type )                 Gruppierung
```

667

32. Syntax von Mini-F# — Operatoren

VI Grammatiken

Ralf Hinze

Reguläre Ausdrücke
Scanner
Kontextfreie Ausdrücke
Motivation
Abstrakte Syntax
Reduktionsemantik
Denotationelle Semantik
Vertiefung
Parser»

Ausdrücke	
;	rechtsassoziierend
:=	rechtsassoziierend
,	nicht assoziierend
	linksassoziierend
&&	linksassoziierend
< =< = <> =>	linksassoziierend
~	rechtsassoziierend
:: @	rechtsassoziierend
+ -	linksassoziierend (infix)
* / %	linksassoziierend
**	rechtsassoziierend
Funktionsapplikation	linksassoziierend
+ - !	linksassoziierend (prefix)
.	linksassoziierend
Muster	
	rechtsassoziierend
&	rechtsassoziierend
Typen	
->	rechtsassoziierend

668

33. Akzeptoren

VI Grammatiken

Ralf Hinze

Reguläre Ausdrücke
Scanner
Kontextfreie Ausdrücke
Parser»

- *Zur Erinnerung:* für reguläre Ausdrücke haben wir Akzeptoren generiert, indem wir systematisch alle Rechtsfaktoren berechnet haben.
- Das Verfahren lässt sich nicht auf kontextfreie Ausdrücke übertragen, da diese im Allgemeinen unendlich viele Rechtsfaktoren besitzen.
- *Beispiel:* die „Klammersprache“ $E = \text{rec } x \rightarrow \epsilon \mid a \ x \ b$ hat die Rechtsfaktoren

$$E / a = (\epsilon \mid a \ E \ b) / a = E \ b$$

$$E / b = (\epsilon \mid a \ E \ b) / b = \epsilon$$

Wenn wir bereits ein a gesehen haben, erwarten wir als Rest einen korrekten Klammerausdruck gefolgt von einem b.

$$E \ b / a = (\epsilon \mid a \ E \ b) \ b / a = E \ b^2$$

$$E \ b / b = (\epsilon \mid a \ E \ b) \ b / b = \epsilon$$

Sehen wir ein weiteres a, dann müssen nach dem Klammerausdruck zwei bs kommen. Und so weiter ...

669

33. Akzeptoren — Idee

- ▶ *Beobachtung:* E tritt im Rechtsfaktor wieder auf, allerdings gefolgt von unterschiedlichen Ausdrücken (b , b^2 , ...).

$$\begin{aligned} E / a &= (\epsilon \mid a E b) / a = E b \\ E b / a &= (\epsilon \mid a E b) b / a = E b^2 \end{aligned}$$

- ▶ *Idee:* der „Akzeptor“ für E kann durch eine rekursive Funktion implementiert werden, wenn wir ihn mit dem Akzeptor für den Folgeausdruck parametrisieren.

type $Follow = List \langle Alphabet \rangle \rightarrow Bool$

let $accept-E (follow : Follow) : Follow$

- 👉 Der Parameter $follow$ legt fest, was *nach* E erwartet wird.
- 👉 Die Typdefinition führt ein sogenanntes *Typsynonym* ein, eine Abkürzung für den Typ auf der rechten Seite.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Parser*

670

33. Akzeptoren — Klammersprache

Mit diesem Ansatz sieht der Akzeptor für den kontextfreien Ausdruck b — ein einzelnes Terminalsymbol — wie folgt aus.

```
let  $accept-b (follow : Follow) : Follow = fun\ input \rightarrow$   
  match  $input\ with$   
  |  $B :: rest \rightarrow follow\ rest$   
  |  $\_ \rightarrow false$ 
```

👉 Fängt die Eingabe mit einem b an, wird die Überprüfung der restlichen Eingabe an $follow$ delegiert.

👉 Anderenfalls ist die Eingabe nicht in der Sprache enthalten — für *keinen* Folgeausdruck.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Parser*

671

33. Akzeptoren — Klammersprache

Der Akzeptor für die Klammersprache:

```
let rec  $accept-E (follow : Follow) : Follow = fun\ input \rightarrow$   
  match  $input\ with$   
  |  $A :: rest \rightarrow accept-E (accept-b\ follow)\ rest$   
  |  $\_ \rightarrow follow\ input$ 
```

👉 Für jedes gelesene a wird der Folgeakzeptor um $accept-b$ erweitert; nach dem ersten b wird der akkumulierte Zopf von $accept-bs$ abgearbeitet.

👉 Wir können zählen, ohne die natürlichen Zahlen bemühen zu müssen ;-).

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Parser*

672

33. Akzeptoren — Klammersprache

Den gewünschten Akzeptor für E erhalten wir, indem wir $accept-E$ mit dem Akzeptor für ϵ aufrufen: $accept-E\ end-of-input$ wobei $end-of-input$ wie folgt definiert ist.

```
let  $end-of-input = fun\ input \rightarrow$   
  match  $input\ with$   
  |  $[] \rightarrow true$   
  |  $\_ :: \_ \rightarrow false$ 
```

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Parser*

673

33. Akzeptoren

Zum Vergleich:

- ▶ Akzeptor für einen regulären Ausdruck:

```
accept-r00 : List (Alphabet) → Bool
```

Ein Akzeptor für einen „isolierten“ regulären Ausdruck.

- ▶ „Akzeptor“ für einen kontextfreien Ausdruck:

```
type Follow = List (Alphabet) → Bool
```

```
accept-E : Follow → Follow
```

Ein Akzeptor für einen kontextfreien Ausdruck gefolgt von einem beliebigen kontextfreien Ausdruck.

- ☞ Ein weiteres Beispiel für die Programmieretechnik der *Verallgemeinerung*.

33. Akzeptoren mit System

Mit Hilfe dieses Ansatzes lässt sich zu jedem kontextfreien Ausdruck systematisch ein korrespondierender Mini-F# Ausdruck, ein Akzeptor des Typs *Follow* → *Follow*, konstruieren.

Terminalsymbol *a*:

```
fun follow →  
  fun input →  
    match input with  
    | A :: rest → follow rest  
    | _        → false
```

- ☞ *A* ist das zu *a* korrespondierende Element des Alphabets.

33. Akzeptoren mit System

Bezeichner *x*:

```
accept-x
```

Das leere Wort ϵ :

```
fun follow →  
  fun input →  
    follow input
```

- ☞ Wir nutzen $\epsilon c = c$ aus.
Kürzer: **fun** follow → follow. Wie heißt diese Funktion?

33. Akzeptoren mit System

Die Sequenz $c_1 c_2$:
ist *accept*₁; die Implementierung von *c*₁, dann wird die Sequenz $c_1 c_2$ wie folgt implementiert.

```
fun follow →  
  fun input →  
    accept1 (accept2 follow) input
```

- ☞ Wir nutzen $(c_1 c_2) c = c_1 (c_2 c)$ aus.
Kürzer: **fun** follow → accept₁ (accept₂ follow). Wie heißt diese Funktion?

33. Akzeptoren mit System

Die leere Sprache \emptyset :

```
fun follow →  
  fun input →  
    false
```

☞ Wir nutzen $\emptyset c = \emptyset$ aus.

Die Alternative $c_1 \mid c_2$:
ist $accept_i$ die Implementierung von c_i , dann wird die Alternative $c_1 \mid c_2$ wie folgt implementiert.

```
fun follow →  
  fun input →  
    accept1 follow input || accept2 follow input
```

☞ Wir nutzen das Distributivgesetz $(c_1 \mid c_2) c = c_1 c \mid c_2 c$ aus.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Parser*

678

33. Akzeptoren mit System

Der rekursive Ausdruck $rec\ x \rightarrow c$:
ist $accept$ die Implementierung von c , dann wird der rekursive Ausdruck $rec\ x \rightarrow c$ wie folgt implementiert.

```
let rec accept-x (follow) =  
  fun input →  
    accept follow input  
in accept-x
```

☞ Tritt x in c auf, so kommt entsprechend $accept-x$ in $accept$ vor.

Kürzer: $let\ rec\ accept-x (follow) = accept\ follow\ in\ accept-x$.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Parser*

679

33. Akzeptoren mit System

Der verschränkt rekursive Ausdruck $rec\ x_1 \rightarrow c_1\ and\ \dots\ and\ x_n \rightarrow c_n$ wird entsprechend auf einen verschränkt rekursiven Mini-F# Ausdruck abgebildet.

```
let rec accept-x1 (follow) = accept1 follow  
  and ...  
  and accept-xn (follow) = acceptn follow  
in accept-x1
```

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Parser*

680

33. Akzeptoren mit System — Beispiel

Setzen wir die Bausteine entsprechend zusammen, ergibt sich für die Klammersprache

```
rec x → ε | a x b
```

der folgende kompakte Mini-F# Ausdruck.

```
let rec accept-x (follow : Follow) : Follow =  
  fun input →  
    follow input || accept-a (accept-x (accept-b follow)) input  
in accept-x
```

☞ Die Struktur des Mini-F# Ausdrucks spiegelt die Struktur des kontextfreien Ausdrucks wider.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Parser*

681

33. Akzeptoren mit noch mehr System

Zu jedem kontextfreien Ausdruck korrespondiert ein Mini-F# Ausdruck.

Wir können diese Korrespondenz auch explizit machen, indem wir den einzelnen Bausteinen einen Namen geben und so eine Bibliothek für die Konstruktion von Akzeptoren erstellen.

```
type Follow = List <Alphabet> → Bool
type Acceptor = Follow → Follow
let symbol (a : Alphabet) : Acceptor =
  fun (follow : Follow) →
    fun input →
      match input with
      | [] → false
      | b :: rest → a = b && follow rest
```

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Parser*

682

33. Akzeptoren mit noch mehr System

```
let eps : Acceptor =
  fun (follow : Follow) → follow
let seq (accept1 : Acceptor, accept2 : Acceptor) : Acceptor =
  fun (follow : Follow) → accept1 (accept2 follow)
let empty : Acceptor =
  fun (follow : Follow) →
    fun input → false
let alt (accept1 : Acceptor, accept2 : Acceptor) : Acceptor =
  fun (follow : Follow) →
    fun input → accept1 follow input || accept2 follow input
```

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Parser*

683

33. Akzeptoren mit noch mehr System — Beispiel

Mit Hilfe dieser Bibliothek kann der Akzeptor für die Klammersprache

```
rec x → ε | a x b
```

noch etwas kompakter definiert werden.

```
let rec accept-x (follow : Follow) : Follow =
  alt (eps, seq (seq (symbol A, accept-x), symbol B)) follow
in accept-x
```

☞ Die Struktur des Mini-F# Ausdrucks spiegelt exakt die Struktur des kontextfreien Ausdrucks wider.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Parser*

684

33. Beispiel — einfache arithmetische Ausdrücke

Probieren wir die Technik an einem weiteren Beispiel aus, den einfachen arithmetischen Ausdrücken.

```
rec expr0 → expr1 | expr1 + expr0
and expr1 → expr2 | expr2 * expr1
and expr2 → num | ( expr0 )
```

Das diesem Ausdruck zugrundeliegende Alphabet ist *Token*:

```
type Token = | Num of Nat | LParen | RParen | Asterisk | Plus
```

☞ Die offene Klammer (wird durch den Konstruktor *LParen* repräsentiert; das Numeral 4711 durch den Wert *Num 4711*.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Parser*

685

33. Beispiel — einfache arithmetische Ausdrücke

Die Umsetzung von

```
rec expr0 → expr1 | expr1 + expr0
and expr1 → expr2 | expr2 * expr1
and expr2 → num | ( expr0 )
```

geht mechanisch vonstatten:

```
let rec accept-expr0 (follow : Follow) : Follow =
  alt (accept-expr1,
       seq (seq (accept-expr1, symbol Plus), accept-expr0)) follow
and accept-expr1 (follow : Follow) : Follow =
  alt (accept-expr2,
       seq (seq (accept-expr2, symbol Asterisk), accept-expr1)) follow
and accept-expr2 (follow : Follow) : Follow =
  alt (accept_num,
       seq (seq (symbol LParen, accept-expr0), symbol RParen)) follow
```

Die Struktur des Mini-F# Ausdrucks (**let** ... **in** *accept-expr₀*) spiegelt exakt die Struktur des kontextfreien Ausdrucks wider.

VI Grammatiken

Ralf Hinze

Reguläre Ausdrücke

Scanner

Kontextfreie Ausdrücke

Parser*

686

33. Beispiel — einfache arithmetische Ausdrücke

Zum Vergleich: der gleiche Akzeptor ohne Verwendung der Bausteine.

```
let rec accept-expr0 (follow : Follow) : Follow =
  fun input →
    accept-expr1 follow input
  || accept-expr1 (symbol Plus (accept-expr0 follow)) input
and accept-expr1 (follow : Follow) : Follow =
  fun input →
    accept-expr2 follow input
  || accept-expr2 (symbol Asterisk (accept-expr1 follow)) input
and accept-expr2 (follow : Follow) : Follow =
  fun input →
    accept_num follow input
  || symbol LParen (accept-expr0 (symbol RParen follow)) input
```

VI Grammatiken

Ralf Hinze

Reguläre Ausdrücke

Scanner

Kontextfreie Ausdrücke

Parser*

687

33. Linksrekursion

Jeder kontextfreie Ausdruck lässt sich systematisch in einen Mini-F# Ausdruck überführen. Ist damit der Fall abgeschlossen?

Nein! Die Umsetzung rekursiver Sprachen ist *nicht perfekt*: der kontextfreie Ausdruck $rec\ x \rightarrow x$ wird auf den Mini-F# Ausdruck

```
let rec accept-x (follow) →
  fun input →
    accept-x follow input
in accept-x
```

abgebildet, eine *nichtterminierende* Funktion.

Die Bedeutung von $rec\ x \rightarrow x$ ist aber die leere Sprache. Deren korrekte Implementierung lautet

```
fun follow →
  fun input →
    false
```

eine stets *terminierende* Funktion.

VI Grammatiken

Ralf Hinze

Reguläre Ausdrücke

Scanner

Kontextfreie Ausdrücke

Parser*

688

33. Linksrekursion

Das Problem der Nichtterminierung tritt immer dann auf, wenn der kontextfreie Ausdruck *linksrekursiv* ist. Dann erfolgt der rekursive Aufruf, ohne dass die Liste von Tokens verkleinert wurde.

Beispiel: Klammergebirge.

```
rec x → ε | x a x b
```

Der zugehörige Akzeptor terminiert für die Eingabe *abab* nicht.

Die äquivalente, rechtsrekursive Formulierung bereitet keine Probleme.

```
rec x → ε | a x b x
```

Beim rekursiven Aufruf des Akzeptors ist sichergestellt, dass die Eingabe verkleinert wurde: der zu *a* korrespondierende Akzeptor hat vorher einen Buchstaben konsumiert.

VI Grammatiken

Ralf Hinze

Reguläre Ausdrücke

Scanner

Kontextfreie Ausdrücke

Parser*

689

33. Linksrekursion

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Parser*

Fazit:

- ▶ Bei handgeschriebenen Akzeptoren muss man Sorge tragen, dass die rekursiven Aufrufe auf kleineren Eingaben arbeiten.
- ▶ Linksrekursion ist *um jeden Preis* zu vermeiden.
- ▶ *Zum Vergleich:* Bei *regulären Ausdrücken* wird durch das Konstruktionsverfahren sichergestellt, dass die Eingabe stets kleiner wird.

690

33. Parser

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Parser*

- ▶ Ein Akzeptor beantwortet die Frage „Ist das gegebene Wort in der von dem kontextfreien Ausdruck bezeichneten Sprache enthalten?“.
- ▶ Ein Parser beantwortet die gleiche Frage, gibt aber im positiven Fall zusätzlich einen *semantischen Wert* zurück.
- ▶ Im Fall arithmetischer Ausdrücke kann der semantische Wert
 - ▶ der Wert des Ausdrucks oder
 - ▶ der abstrakte Syntaxbaum des Ausdrucks sein.
 - ▶ Der letztere Ansatz ist allgemeiner.

691

33. Abstrakte Syntax arithmetischer Ausdrücke

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Parser*

Die abstrakte Syntax arithmetischer Ausdrücke kann mit einer rekursiven Variantentypdefinition eingefangen werden.

```
type Expr =  
  | Const of Nat  
  | Add of Expr * Expr  
  | Mul of Expr * Expr
```

☞ Rekursive Variantentypen sind das Mini-F# Pendant zu Baumsprachen, so dass wir die Baumsprache für arithmetische Ausdrücke im Wesentlichen übernehmen können.

692

33. Ein Auswerter für arithmetische Ausdrücke

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Parser*

Aufgabe: *evaluate* soll einen arithmetischen Ausdruck auswerten.

Mit dem Struktur Entwurfsmuster für *Expr* erhalten wir:

```
let rec evaluate (expr : Expr) : Nat =  
  match expr with  
  | Const nat          → ...  
  | Add (expr1, expr2) → ... evaluate expr1 ... evaluate expr2 ...  
  | Mul (expr1, expr2) → ... evaluate expr1 ... evaluate expr2 ...
```

693

33. Ein Auswerter für arithmetische Ausdrücke

- ▶ *Rekursionsbasis*: Fall $expr = Const\ nat$.

```
let rec evaluate (expr : Expr) : Nat =
  match expr with
  | Const nat      → nat
  | Add (expr1, expr2) → ... evaluate expr1 ... evaluate expr2 ...
  | Mul (expr1, expr2) → ... evaluate expr1 ... evaluate expr2 ...
```

- ▶ *Rekursionsschritt*: Fall $expr = Add\ (expr_1, expr_2)$.

```
let rec evaluate (expr : Expr) : Nat =
  match expr with
  | Const nat      → nat
  | Add (expr1, expr2) → evaluate expr1 + evaluate expr2
  | Mul (expr1, expr2) → ... evaluate expr1 ... evaluate expr2 ...
```

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Parser*

694

33. Ein Auswerter für arithmetische Ausdrücke

- ▶ *Rekursionsschritt*: Fall $expr = Mul\ (expr_1, expr_2)$.

```
let rec evaluate (expr : Expr) : Nat =
  match expr with
  | Const nat      → nat
  | Add (expr1, expr2) → evaluate expr1 + evaluate expr2
  | Mul (expr1, expr2) → evaluate expr1 * evaluate expr2
```

☞ Der Auswerter ist ein waschechter Interpreter.

Der Auswerter internalisiert die Auswertungsregeln $e \Downarrow v$.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Parser*

695

33. Parser — semantische Werte

Ein *Folgeakzeptor* liefert einen Booleschen Wert als Ergebnis:

```
type Follow = List <Token> → Bool
```

Ein *Folgeparser* muss im Erfolgsfall einen semantischen Wert zurückgeben. Aus dem Typ *Bool* wird der Typ *Option*:

```
type Follow ('v) = List <Token> → Option ('v)
```

☞ Der Typparameter 'v spezifiziert den Typ des resultierendes Wertes, für unser laufendes Beispiel ist der Typ zum Beispiel *Nat* (Semantik) oder *Expr* (abstrakter Syntaxbaum).

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Parser*

696

33. Parser — semantische Werte

Welche Änderungen ergeben sich für die Programme?

(„Kennen Sie den Weg zum Audimax?“ Statt schroff mit „Ja!“ oder „Nein!“ zu antworten, sagt man im positiven Fall „Ja! Betreten Sie das Gebäude durch den Haupteingang, dann halten Sie sich links ...“)

Allgemein: wenn bei einer Programmtransformation aus dem Typ *Bool* der Typ *Option* (*t*) wird, dann ändert sich

- ▶ *false* zu *None*,
- ▶ *true* zu *Some e* mit $e : t$, und
- ▶ $e_1 \parallel e_2$ bzw. *if* e_1 *then true* *else* e_2 wird zu

```
match e1 with
| Some a → Some a
| None   → e2
```

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Parser*

697

33. Parser — semantische Werte

Damit erhalten wir für *end-of-input*:

```
let end-of-input : Follow ⟨Expr⟩ =  
  fun input →  
    match input with  
    | [] → Some ?  
    | _ :: _ → None
```

☞ Woher nehmen wir das Argument für *Some*?
Wir müssen *end-of-input* den Wert mit auf den Weg geben.

```
let end-of-input (e : Expr) : Follow ⟨Expr⟩ =  
  fun input →  
    match input with  
    | [] → Some e  
    | _ :: _ → None
```

☞ Aus einer Funktion des Typs *Follow* ist eine Funktion des Typs $Expr \rightarrow Follow \langle Expr \rangle$ geworden.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Parser*

698

33. Parser — semantische Werte

Idee: Jeder Parser übergibt seinen semantischen Wert an den Folgeparser. Die Funktion *end-of-input* ist der initiale Folgeparser.

Ein Akzeptor hat den Typ

```
type Acceptor = Follow → Follow
```

Ein Parser hat den Typ

```
type Parser ⟨a⟩ = (a → Follow ⟨Expr⟩) → Follow ⟨Expr⟩
```

☞ *Expr* ist der Typ des semantischen Wertes, den der Folgeparser als *endgültiges* Ergebnis zurückgibt.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Parser*

699

33. Parser — semantische Werte

Welche Änderungen ergeben sich für die Programme?

► Aus

```
accept-expr0
```

wird zum Beispiel

```
parse-expr0 (fun e → ...)
```

☞ *parse-expr₀* selbst übergibt der bereitgestellten Funktion den semantischen Wert.

► Aus

```
follow
```

wird zum Beispiel

```
follow (Add (e1, e2))
```

☞ Dem Folgeparser wird der semantische Wert übergeben.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Parser*

700

33. Parser — einfache arithmetische Ausdrücke

Änderungen an dem Parser für einfache arithmetische Ausdrücke:

```
let rec parse-expr0 (follow : Expr → Follow ⟨Expr⟩) : Follow ⟨Expr⟩ =  
  fun input →  
    match parse-expr1 (fun e → follow e) input with  
    | None →  
      parse-expr1 (fun e1 →  
        symbol Plus (  
          parse-expr0 (fun e2 →  
            follow (Add (e1, e2)))) input  
        | Some e → Some e
```

☞ *symbol* muss ebenfalls angepasst werden — zur Übung.

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Parser*

701

33. Parser — einfache arithmetische Ausdrücke

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Parser*

Fortsetzung:

```
and parse-expr1 (follow : Expr → Follow ⟨Expr⟩) : Follow ⟨Expr⟩ =  
  fun input →  
    match parse-expr2 (fun e → follow e) input with  
    | None →  
      parse-expr2 (fun e1 →  
        symbol Asterisk (  
          parse-expr1 (fun e2 →  
            follow (Mul (e1, e2)))))) input  
    | Some e → Some e
```

702

33. Parser — einfache arithmetische Ausdrücke

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Parser*

Fortsetzung:

```
and parse-expr2 (follow : Expr → Follow ⟨Expr⟩) : Follow ⟨Expr⟩ =  
  fun input →  
    match parse-num (fun n → follow (Const n)) input with  
    | None →  
      symbol LParen (  
        parse-expr0 (fun e →  
          symbol RParen (  
            follow e))) input  
    | Some e → Some e
```

☞ Insgesamt ergibt sich das Bild einer Kette: es wird ein Korb herumgereicht; jeder fügt ein paar Münzen hinzu und reicht den Korb an den Nachbarn (den Folgeparser) weiter. Am Ende der Kette wird der Korb geleert (*end-of-input*).

703

33. Parser — einfache arithmetische Ausdrücke

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Parser*

Jetzt wird es Zeit, den Parser in Aktion zu erleben.

Zu diesem Zweck kombinieren wir den Scanner *lex* mit dem Parser *parse-expr₀*.

```
let abstract-syntax-tree (input : String) : Option ⟨Expr⟩ =  
  parse-expr0 end-of-input (lex (explode input))  
function abstract-syntax-tree (input : String) : Option ⟨Expr⟩ =  
  parse-expr0 end-of-input (lex (explode input))
```

☞ *explode* überführt einen String in eine Liste von Zeichen; *lex* überführt die Liste von Zeichen in eine Liste von Tokens; *parse-expr₀ end-of-input* überführt die Liste von Tokens in einen abstrakten Syntaxbaum.

704

33. Parser — Demo

VI Grammatiken

Ralf Hinze

Reguläre
Ausdrücke

Scanner

Kontextfreie
Ausdrücke

Parser*

```
Mini) abstract-syntax-tree "4711"  
Some (Const 4711)  
Mini) abstract-syntax-tree "4711+815*2765"  
Some (Add (Const 4711, Mul (Const 815, Const 2765)))  
Mini) abstract-syntax-tree "1+2+3+4"  
Some (Add (Const 1, Add (Const 2, Add (Const 3, Const 4))))  
Mini) abstract-syntax-tree "(1+2)+(3+4)"  
Some (Add (Add (Const 1, Const 2), Add (Const 3, Const 4)))  
Mini) match it with Some expr → evaluate expr  
10  
Mini) abstract-syntax-tree "(1+2+3)*(4+5+6)"  
Some (Mul (Add (Const 1, Add (Const 2, Const 3)),  
  Add (Const 4, Add (Const 5, Const 6))))  
Mini) match it with Some expr → evaluate expr  
90
```

705

33. Zusammenfassung

Wir haben

- ▶ Syntax und Semantik regulärer Ausdrücke definiert,
- ▶ gesehen, wie man aus einem regulären Ausdruck systematisch mit Hilfe der Rechtsfaktoren einen Akzeptor herleitet,
- ▶ die Begriffe Interpreter und Übersetzer eingeführt,
- ▶ Möglichkeiten und Grenzen regulärer Ausdrücke kennengelernt,
- ▶ reguläre Ausdrücke um Rekursion erweitert: zu kontextfreien Ausdrücken,
- ▶ gesehen, wie man aus einem kontextfreien Ausdruck systematisch einen Akzeptor bzw. einen Parser herleitet,
- ▶ die Gefahr der Nichtterminierung bei linkrekursiven Grammatiken besprochen.