32. Knobelaufgabe #17

Welche Sprache bezeichnet der folgende reguläre Ausdruck?

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

Und welche Sprachen bezeichnen $\mathit{riddle} \,/\, \mathtt{a}$ und $\mathit{riddle} \,/\, \mathtt{b}$?

32. Motivation

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

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

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

$$\textbf{\textit{rec}}\ x \to \epsilon \mid \mathtt{a}\ x\ \mathtt{b}$$

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

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“:

  rec *expr* → 0 | `false` | *expr* + *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.

32. Abstrakte Syntax

$x \in \mathsf{Id}$

$c \in \mathsf{CF} ::=$		*kontextfreie Ausdrücke:*
	a	Terminalsymbol
$\mid$	x	Bezeichner$\setminus$Nichtterminalsymbol
$\mid$	ϵ	das leere Wort
$\mid$	$c_1\ c_2$	Konkatenation$\setminus$Sequenz
$\mid$	$\emptyset$	die leere Sprache
$\mid$	$c_1 \mid c_2$	Alternative
$\mid$	$\boldsymbol{rec}\ x \to c$	Rekursion

☞ *Zusätzliche Konstrukte:* Bezeichner und Rekursion.

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

32. Reduktionssemantik

Teaser: Was ist die Bedeutung von

- ***rec*** $x \rightarrow x$,
- ***rec*** $x \rightarrow \mathrm{a}\ x\ \mathrm{b}$,
- ***rec*** $x \rightarrow x^*$,
- ***rec*** $x \rightarrow x^*\ \mathrm{a}$,
- ***rec*** $x \rightarrow$ ***rec*** $y \rightarrow \epsilon \mid \mathrm{a} \mid x\ y$?

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

32. Reduktionssemantik — Rechenregel

Rechenregel:

$$\overline{(\textbf{\textit{rec}}\ x \to c) \longrightarrow c\{x \mapsto \textbf{\textit{rec}}\ x \to c\}}$$

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

32. Reduktionssemantik — Beispiel

Mit Hilfe der Rechenregel können wir zum Beispiel aabb aus dem Ausdruck ***rec*** $x \to \epsilon \mid \mathtt{a}\ x\ \mathtt{b}$ ableiten.

$$
\begin{aligned}
& \textbf{\textit{rec}}\ x \to \epsilon \mid \mathtt{a}\ x\ \mathtt{b} \\
\longrightarrow\ & \epsilon \mid \mathtt{a}\ (\textbf{\textit{rec}}\ x \to \epsilon \mid \mathtt{a}\ x\ \mathtt{b})\ \mathtt{b} \\
\longrightarrow\ & \mathtt{a}\ (\textbf{\textit{rec}}\ x \to \epsilon \mid \mathtt{a}\ x\ \mathtt{b})\ \mathtt{b} \\
\longrightarrow\ & \mathtt{a}\ (\epsilon \mid \mathtt{a}\ (\textbf{\textit{rec}}\ x \to \epsilon \mid \mathtt{a}\ x\ \mathtt{b})\ \mathtt{b})\ \mathtt{b} \\
\longrightarrow\ & \mathtt{a}\ (\mathtt{a}\ (\textbf{\textit{rec}}\ x \to \epsilon \mid \mathtt{a}\ x\ \mathtt{b})\ \mathtt{b})\ \mathtt{b} \\
\longrightarrow\ & \mathtt{a}\ (\mathtt{a}\ (\epsilon \mid \mathtt{a}\ (\textbf{\textit{rec}}\ x \to \epsilon \mid \mathtt{a}\ x\ \mathtt{b})\ \mathtt{b})\ \mathtt{b})\ \mathtt{b} \\
\longrightarrow\ & \mathtt{a}\ (\mathtt{a}\ \epsilon\ \mathtt{b})\ \mathtt{b} \\
\longrightarrow\ & \mathtt{a}\ \mathtt{a}\ \mathtt{b}\ \mathtt{b}
\end{aligned}
$$

32. Reduktionssemantik — Substitution

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

$$c\sigma$$

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

Beispiel:

$$(g\ \mathtt{b}\ g \mid u\ \mathtt{b}\ u)\{g \mapsto (\mathtt{aa})^*, u \mapsto \mathtt{a}\ (\mathtt{aa})^*\}$$
$$= (\mathtt{aa})^*\ \mathtt{b}\ (\mathtt{aa})^* \mid \mathtt{a}\ (\mathtt{aa})^*\ \mathtt{b}\ \mathtt{a}\ (\mathtt{aa})^*$$

32. Reduktionssemantik — Substitution

Substitution von Bezeichnern in Ausdrücken:

$$
\begin{array}{lcl}
a\sigma & = & a \\
x\sigma & = & \begin{cases} \sigma(x), & \textbf{if } x \in \mathit{dom}\,(\sigma) \\ x, & \textbf{otherwise} \end{cases} \\
\epsilon\sigma & = & \epsilon \\
(c_1\ c_2)\sigma & = & (c_1\sigma)\,(c_2\sigma) \\
\emptyset\sigma & = & \emptyset \\
(c_1 \mid c_2)\sigma & = & (c_1\sigma) \mid (c_2\sigma) \\
(\textbf{\textit{rec}}\ x \to c)\sigma & = & \textbf{\textit{rec}}\ x \to c(\sigma \setminus \{x\})
\end{array}
$$

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

32. Reduktionssemantik — Teaser

Wie sieht es mit dem Ausdruck ***rec*** $x \to x$ aus?

$$\begin{aligned} & \textbf{\textit{rec}}\ x \to x \\ \longrightarrow\quad & \textbf{\textit{rec}}\ x \to x \\ & \vdots \end{aligned}$$

☞ Wir machen keinen Fortschritt!

☞ Wir können kein *Wort* ableiten; ***rec*** $x \to x$ bezeichnet die leere Sprache.

32. Reduktionssemantik — Teaser

Wie sieht es mit dem Ausdruck $\textbf{\textit{rec}}\ x \to \mathtt{a}\ x\ \mathtt{b}$ aus?

$$
\begin{aligned}
& \textbf{\textit{rec}}\ x \to \mathtt{a}\ x\ \mathtt{b} \\
\longrightarrow\ & \mathtt{a}\ (\textbf{\textit{rec}}\ x \to \mathtt{a}\ x\ \mathtt{b})\ \mathtt{b} \\
\longrightarrow\ & \mathtt{a}\ \mathtt{a}\ (\textbf{\textit{rec}}\ x \to \mathtt{a}\ x\ \mathtt{b})\ \mathtt{b}\ \mathtt{b} \\
& \vdots
\end{aligned}
$$

☞ Der Ausdruck wird immer größer!

☞ Wir können wiederum kein Wort ableiten; auch $\textbf{\textit{rec}}\ x \to \mathtt{a}\ x\ \mathtt{b}$ bezeichnet die leere Sprache.

32. Reduktionssemantik — Teaser

Wie sieht es mit dem Ausdruck ***rec*** $x \to x^*$ aus?

$$\begin{aligned} & \textbf{\textit{rec}}\ x \to x^* \\ \longrightarrow\quad & (\textbf{\textit{rec}}\ x \to x^*)^* \\ \longrightarrow\quad & \epsilon \end{aligned}$$

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

32. Denotationelle Semantik

Bei der Motivation von ***rec*** $x \to \epsilon \mid \mathtt{a}\ x\ \mathtt{b}$ haben wir überlegt, wie sich $\mathtt{a}^n\ \mathtt{b}^n$ *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*** $x \to \epsilon \mid \mathtt{a}\ x\ \mathtt{b}$ leiten wir zunächst eine Funktion auf Sprachen ab.

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

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

☞ 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*** $x \to \epsilon \mid \mathtt{a}\ x\ \mathtt{b}$ nähern.

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

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

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

☞ 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.

$$
\begin{aligned}
[\![a]\!]\varrho &= \{a\} \\
[\![x]\!]\varrho &= \varrho(x) \\
[\![\epsilon]\!]\varrho &= \{\epsilon\} \\
[\![c_1\ c_2]\!]\varrho &= [\![c_1]\!]\varrho \cdot [\![c_2]\!]\varrho \\
[\![\emptyset]\!]\varrho &= \emptyset \\
[\![c_1 \shortmid c_2]\!]\varrho &= [\![c_1]\!]\varrho \cup [\![c_2]\!]\varrho \\
[\![\textbf{\textit{rec}}\ x \to c]\!]\varrho &= \bigcup\{F^n(\emptyset) \mid n \in \mathbb{N}\} \text{ mit } F(X) = [\![c]\!](\varrho, \{x \mapsto X\})
\end{aligned}
$$

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

32. Denotationelle Semantik — Teaser

Die dem Ausdruck ***rec*** $x \to x$ zugeordnete Funktion ist

$$F(L) = L$$

Fixpunktiteration:

$$\emptyset$$
$$F(\emptyset) = \emptyset$$

☞ Die Bedeutung von ***rec*** $x \to x$ ist die leere Sprache.

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

32. Denotationelle Semantik — Teaser

Die dem Ausdruck ***rec*** $x \to \mathtt{a}\ x\ \mathtt{b}$ zugeordnete Funktion ist

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

Fixpunktiteration:

$$\emptyset$$
$$G(\emptyset) = \{\mathtt{a}\} \cdot \emptyset \cdot \{\mathtt{b}\} = \emptyset$$

☞ Die Bedeutung von ***rec*** $x \to \mathtt{a}\ x\ \mathtt{b}$ ist ebenfalls die leere Sprache.

☞ *Zum Knobeln:* Wieviele Fixpunkte hat G?

32. Denotationelle Semantik — Teaser

Die dem Ausdruck ***rec*** $x \to x^*$ zugeordnete Funktion ist

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

Fixpunktiteration:

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

☞ Die Bedeutung von ***rec*** $x \to x^*$ ist $\{\epsilon\}$.

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.

32. Vertiefung

Versuchen wir uns an der kontextfreien Syntax einfacher arithmetischer Ausdrücke.

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

☞ 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.

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:

$$
\begin{aligned}
E &\longrightarrow E + E \\
&\longrightarrow num + E \\
&\longrightarrow 4711 + E \\
&\longrightarrow 4711 + E * E \\
&\longrightarrow 4711 + num * E \\
&\longrightarrow 4711 + 815 * E \\
&\longrightarrow 4711 + 815 * num \\
&\longrightarrow 4711 + 815 * 2765
\end{aligned}
$$

32. Vertiefung

Es gibt aber noch eine zweite mögliche Reduktionsfolge:

$$
\begin{aligned}
E &\longrightarrow E * E \\
&\longrightarrow E + E * E \\
&\longrightarrow \mathit{num} + E * E \\
&\longrightarrow 4711 + E * E \\
&\longrightarrow 4711 + \mathit{num} * E \\
&\longrightarrow 4711 + 815 * E \\
&\longrightarrow 4711 + 815 * \mathit{num} \\
&\longrightarrow 4711 + 815 * 2765
\end{aligned}
$$

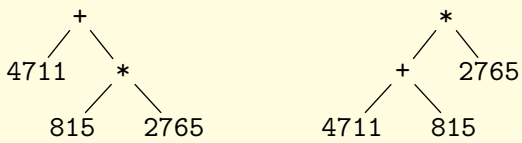
☞ *Problem*: der obige kontextfreie Ausdruck ist *mehrdeutig*; ein Wort kann auf verschiedene Weisen abgeleitet werden.

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.

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?)

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.

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.

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.

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:

  $a \mathbin{**} b \mathbin{**} c$ steht für $\mathbin{**} a \mathbin{**} b\ c$

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.

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 $a \oplus (b \oplus c)$

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: $xs_1 \mathbin{@} (xs_2 \mathbin{@} xs_3)$ und $(xs_1 \mathbin{@} xs_2) \mathbin{@} xs_3$ sind zwar gleichwertig, aber der erste Ausdruck ist schneller ausgerechnet. (Warum?)

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.

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.

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

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

32. Operatoren — eindeutige Grammatiken

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

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 rechtsassoziierend fest. (Warum?)

32. Operatoren — eindeutige Grammatiken

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

Zum Beispiel:

$$
\begin{aligned}
E_0 &\longrightarrow E_1 + E_0 \\
&\longrightarrow E_2 + E_0 \\
&\longrightarrow num + E_0 \\
&\longrightarrow 4711 + E_0 \\
&\longrightarrow 4711 + E_1 \\
&\longrightarrow 4711 + E_2 * E_1 \\
&\longrightarrow 4711 + num * E_1 \\
&\longrightarrow 4711 + 815 * E_1 \\
&\longrightarrow 4711 + 815 * E_2 \\
&\longrightarrow 4711 + 815 * num \\
&\longrightarrow 4711 + 815 * 2765
\end{aligned}
$$

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 rechtsassoziierend) und die Bindungsstärke $(0 \ldots)$ 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.

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* | *num* | *expr* + *expr* | *expr* * *expr* | *decl* `in` *expr*
and *decl* $\rightarrow$ `let` *id* = *expr*

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

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!

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***.

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*** $x \to e$.

☞ Der Ausdruck ***fun*** $x \to x + x$ ist mehrdeutig: $(\textbf{\textit{fun}}\ x \to x) + x$ und $\textbf{\textit{fun}}\ x \to (x + x)$ 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.

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

32. Syntax von Mini–F# — Grammatik

// 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.

32. Syntax von Mini–F# — Grammatik

// Deklarationen

and *decl*	$\rightarrow$	*pat* = *expr*	Wertedefinition
	|	*funcdecl*	Funktionsdefinition
	|	`rec` $\mathit{fundecl}^{+}_{\texttt{and}}$	rekursive Funktionsdefinitionen
and *fundecl*	$\rightarrow$	*id* apat^{+} : *type* = *expr*	Funktionsdefinition

// Muster

and *pat*	$\rightarrow$	*apat*	atomares Muster
	|	*pat* : *type*	Typangabe
and *apat*	$\rightarrow$	*id*	Bezeichner
	|	($\mathit{pat}^{*}_{,}$)	Tupelmuster oder Gruppierung

// Typausdrücke

and *type*	$\rightarrow$	*atype*	atomarer Typ
	|	*type* * *type*	Tupeltyp
	|	*type* -> *type*	Funktionstyp
and *atype*	$\rightarrow$	*ld*	Typbezeichner
	|	(*type*)	Gruppierung

32. Syntax von Mini–F# — Operatoren

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

33. Akzeptoren

- *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“ $E = \textbf{\textit{rec}}\ x \to \epsilon \mid \mathtt{a}\ x\ \mathtt{b}$ hat die Rechtsfaktoren

$$E \,/\, \mathtt{a} = (\epsilon \mid \mathtt{a}\ E\ \mathtt{b}) \,/\, \mathtt{a} = E\ \mathtt{b}$$
$$E \,/\, \mathtt{b} = (\epsilon \mid \mathtt{a}\ E\ \mathtt{b}) \,/\, \mathtt{b} = \emptyset$$

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

$$E\ \mathtt{b} \,/\, \mathtt{a} = (\epsilon \mid \mathtt{a}\ E\ \mathtt{b})\ \mathtt{b} \,/\, \mathtt{a} = E\ \mathtt{b}^2$$
$$E\ \mathtt{b} \,/\, \mathtt{b} = (\epsilon \mid \mathtt{a}\ E\ \mathtt{b})\ \mathtt{b} \,/\, \mathtt{b} = \epsilon$$

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

33. Akzeptoren — Idee

- *Beobachtung:* E tritt im Rechtsfaktor wieder auf, allerdings gefolgt von unterschiedlichen Ausdrücken ($\mathtt{b}$, $\mathtt{b}^2$, ...).

$$E \,/\, \mathtt{a} \quad = (\epsilon \mid \mathtt{a}\; E\; \mathtt{b}) \,/\, \mathtt{a} \quad = E\; \mathtt{b}$$
$$E\; \mathtt{b} \,/\, \mathtt{a} = (\epsilon \mid \mathtt{a}\; E\; \mathtt{b})\; \mathtt{b} \,/\, \mathtt{a} = E\; \mathtt{b}^2$$

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

type $\mathit{Follow} = \mathit{List}\,\langle \mathit{Alphabet} \rangle \to \mathit{Bool}$

let $\mathit{accept\text{-}E}\,(\mathit{follow} : \mathit{Follow}) : \mathit{Follow}$

☞ 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.

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 →
  match input with
  | B :: rest → follow rest
  | _         → 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.

33. Akzeptoren — Klammersprache

Der Akzeptor für die Klammersprache:

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

☞ Für jedes gelesene `a` wird der Folgeakzeptor um *accept-b* erweitert; nach dem ersten `b` wird der akkumulierte Zopf von *accept-b*s abgearbeitet.

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

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 →
  match input with
  | []     → true
  | _ :: _ → false
```

33. Akzeptoren

Zum Vergleich:

- Akzeptor für einen regulären Ausdruck:

  $$\mathit{accept\text{-}r}_{00} : \mathit{List}\ \langle \mathit{Alphabet} \rangle \to \mathit{Bool}$$

  Ein Akzeptor für einen „isolierten“ regulären Ausdruck.

- „Akzeptor“ für einen kontextfreien Ausdruck:

  $$\textbf{\textit{type}}\ \mathit{Follow} = \mathit{List}\ \langle \mathit{Alphabet} \rangle \to \mathit{Bool}$$
  $$\mathit{accept\text{-}E} : \mathit{Follow} \to \mathit{Follow}$$

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

☞ Ein weiteres Beispiel für die Programmiertechnik 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* $\rightarrow$
  fun *input* $\rightarrow$
    follow input

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

33. Akzeptoren mit System

Die *Sequenz* $c_1\ c_2$:
ist $accept_i$ die Implementierung von c_i, 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* $\rightarrow$ $accept_1\ (accept_2\ follow)$. 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.

33. Akzeptoren mit System

Der *rekursive* Ausdruck ***rec*** $x \to c$:
ist *accept* die Implementierung von c, dann wird der rekursive Ausdruck ***rec*** $x \to c$ 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*.

33. Akzeptoren mit System

Der *verschränkt rekursive* Ausdruck ***rec*** $x_1 \to c_1$ ***and*** $\cdots$ ***and*** $x_n \to c_n$ wird entsprechend auf einen verschränkt rekursiven Mini–F# Ausdruck abgebildet.

let rec *accept-*x_1 $(follow) = accept_1\ follow$
    and $\ldots$
    and *accept-*x_n $(follow) = accept_n\ follow$
in *accept-*x_1

33. Akzeptoren mit System — Beispiel

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

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

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.

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
```

33. Akzeptoren mit noch mehr System

$$\textbf{let}\ eps : Acceptor =$$
$$\quad \textbf{fun}\ (follow : Follow) \rightarrow follow$$

$$\textbf{let}\ seq\ (accept_1 : Acceptor, accept_2 : Acceptor) : Acceptor =$$
$$\quad \textbf{fun}\ (follow : Follow) \rightarrow accept_1\ (accept_2\ follow)$$

$$\textbf{let}\ empty : Acceptor =$$
$$\quad \textbf{fun}\ (follow : Follow) \rightarrow$$
$$\quad\quad \textbf{fun}\ input \rightarrow false$$

$$\textbf{let}\ alt\ (accept_1 : Acceptor, accept_2 : Acceptor) : Acceptor =$$
$$\quad \textbf{fun}\ (follow : Follow) \rightarrow$$
$$\quad\quad \textbf{fun}\ input \rightarrow accept_1\ follow\ input\ ||\ accept_2\ follow\ input$$

33. Akzeptoren mit noch mehr System — Beispiel

Mit Hilfe dieser Bibliothek kann der Akzeptor für die Klammersprache

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

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.

33. Beispiel — einfache arithmetische Ausdrücke

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

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

Das diesem Ausdruck zugrundeliegende Alphabet ist *Token*:

type $Token = \mid Num$ ***of*** $Nat \mid LParen \mid RParen \mid Asterisk \mid Plus$

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

33. Beispiel — einfache arithmetische Ausdrücke

Die Umsetzung von

$$\begin{array}{ll} \textbf{rec} & expr_0 \to expr_1 \mid expr_1 \texttt{ + } expr_0 \\ \textbf{and} & expr_1 \to expr_2 \mid expr_2 \texttt{ * } expr_1 \\ \textbf{and} & expr_2 \to num \mid \texttt{(} expr_0 \texttt{)} \end{array}$$

geht mechanisch vonstatten:

$$\begin{array}{l} \textbf{let rec } accept\text{-}expr_0\ (follow : Follow) : Follow = \\ \quad alt\ (accept\text{-}expr_1, \\ \qquad seq\ (seq\ (accept\text{-}expr_1, symbol\ Plus), accept\text{-}expr_0)) \qquad follow \\ \textbf{and } accept\text{-}expr_1\ (follow : Follow) : Follow = \\ \quad alt\ (accept\text{-}expr_2, \\ \qquad seq\ (seq\ (accept\text{-}expr_2, symbol\ Asterisk), accept\text{-}expr_1)) \quad follow \\ \textbf{and } accept\text{-}expr_2\ (follow : Follow) : Follow = \\ \quad alt\ (accept\_num, \\ \qquad seq\ (seq\ (symbol\ LParen, accept\text{-}expr_0), symbol\ RParen))\ follow \end{array}$$

☞ Die Struktur des Mini–F# Ausdrucks (***let*** $\ldots$ ***in*** $accept\text{-}expr_0$) spiegelt exakt die Struktur des kontextfreien Ausdrucks wider.

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
```

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*** $x \to x$ 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*** $x \to x$ ist aber die leere Sprache. Deren korrekte Implementierung lautet

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

eine stets *terminierende* Funktion.

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 $x \rightarrow \epsilon \mid x\ \mathtt{a}\ x\ \mathtt{b}$

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

Die äquivalente, rechtsrekursive Formulierung bereitet keine Probleme.

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

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

33. Linksrekursion

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.

33. 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.

33. Abstrakte Syntax arithmetischer Ausdrücke

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.

33. Ein Auswerter für arithmetische Ausdrücke

Aufgabe: evaluate soll einen arithmetischen Ausdruck auswerten.

Mit dem Struktur Entwurfsmuster für *Expr* erhalten wir:

let rec *evaluate* $(expr : Expr) : Nat =$
  match *expr* ***with***
  $\mid$ *Const nat* $\rightarrow \ldots$
  $\mid$ *Add* $(expr_1, expr_2) \rightarrow \ldots$ *evaluate* $expr_1 \ldots$ *evaluate* $expr_2 \ldots$
  $\mid$ *Mul* $(expr_1, expr_2) \rightarrow \ldots$ *evaluate* $expr_1 \ldots$ *evaluate* $expr_2 \ldots$

33. Ein Auswerter für arithmetische Ausdrücke

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

let rec *evaluate* $(expr : Expr) : Nat =$
  match *expr* ***with***
  $\mid Const\ nat \rightarrow nat$
  $\mid Add\ (expr_1, expr_2) \rightarrow \ldots evaluate\ expr_1 \ldots evaluate\ expr_2 \ldots$
  $\mid Mul\ (expr_1, expr_2) \rightarrow \ldots evaluate\ expr_1 \ldots evaluate\ expr_2 \ldots$

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

let rec *evaluate* $(expr : Expr) : Nat =$
  match *expr* ***with***
  $\mid Const\ nat \rightarrow nat$
  $\mid Add\ (expr_1, expr_2) \rightarrow evaluate\ expr_1 + evaluate\ expr_2$
  $\mid Mul\ (expr_1, expr_2) \rightarrow \ldots evaluate\ expr_1 \ldots evaluate\ expr_2 \ldots$

33. Ein Auswerter für arithmetische Ausdrücke

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

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ evaluate\ (expr : Expr) : Nat = \\
\quad \textbf{\textit{match}}\ expr\ \textbf{\textit{with}} \\
\quad\quad \mid Const\ nat \qquad\qquad \to nat \\
\quad\quad \mid Add\ (expr_1, expr_2) \to evaluate\ expr_1 + evaluate\ expr_2 \\
\quad\quad \mid Mul\ (expr_1, expr_2) \to evaluate\ expr_1 * evaluate\ expr_2
\end{array}
$$

☞ Der Auswerter ist ein waschechter Interpreter.

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

33. Parser — semantische Werte

Ein *Folgeakzeptor* liefert einen Booleschen Wert als Ergebnis:

$$\textbf{type}\ \mathit{Follow} = \mathit{List}\,\langle \mathit{Token}\rangle \to \mathit{Bool}$$

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

$$\textbf{type}\ \mathit{Follow}\,\langle' v\rangle = \mathit{List}\,\langle \mathit{Token}\rangle \to \mathit{Option}\,\langle' v\rangle$$

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

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* $\langle t \rangle$ wird, dann ändert sich

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

match e_1 ***with***
| *Some a* $\rightarrow$ *Some a*
| *None* $\rightarrow e_2$

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 $Expr \to Follow\ \langle Expr \rangle$ geworden.

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

$$\textbf{type}\ \mathit{Acceptor} = \mathit{Follow} \to \mathit{Follow}$$

Ein *Parser* hat den Typ

$$\textbf{type}\ \mathit{Parser}\ \langle 'a \rangle = ('a \to \mathit{Follow}\ \langle \mathit{Expr} \rangle) \to \mathit{Follow}\ \langle \mathit{Expr} \rangle$$

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

33. Parser — semantische Werte

Welche Änderungen ergeben sich für die Programme?

- Aus

  $accept\text{-}expr_0$

  wird zum Beispiel

  $parse\text{-}expr_0\ (\textbf{\textit{fun}}\ e \to \ldots)$

  ☞ $parse\text{-}expr_0$ selbst übergibt der bereitgestellten Funktion den semantischen Wert.
- Aus

  $follow$

  wird zum Beispiel

  $follow\ (Add\ (e_1, e_2))$

  ☞ Dem Folgeparser wird der semantische Wert übergeben.

33. Parser — einfache arithmetische Ausdrücke

Änderungen an dem Parser für einfache arithmetische Ausdrücke:

```
let rec parse-expr_0 (follow : Expr → Follow ⟨Expr⟩) : Follow ⟨Expr⟩ =
  fun input →
    match parse-expr_1 (fun e → follow e) input with
    | None →
      parse-expr_1       (fun e_1 →
        symbol Plus      (
          parse-expr_0 (fun e_2 →
            follow (Add (e_1, e_2))))) input
    | Some e → Some e
```

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

33. Parser — einfache arithmetische Ausdrücke

Fortsetzung:

```
and parse-expr_1 (follow : Expr → Follow ⟨Expr⟩) : Follow ⟨Expr⟩ =
  fun input →
    match parse-expr_2 (fun e → follow e) input with
    | None →
      parse-expr_2          (fun e_1 →
        symbol Asterisk (
          parse-expr_1      (fun e_2 →
            follow (Mul (e_1, e_2))))) input
    | Some e → Some e
```

33. Parser — einfache arithmetische Ausdrücke

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 Kollekte: 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*).

33. Parser — einfache arithmetische Ausdrücke

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

Zu diesem Zweck kombinieren wir den Scanner *lex* mit dem Parser $\mathit{parse\text{-}expr}_0$.

let *abstract-syntax-tree* (*input* : *String*) : *Option* ⟨*Expr*⟩ =
  $\mathit{parse\text{-}expr}_0$ *end-of-input* (*lex* (*explode input*))

function *abstract-syntax-tree* (*input* : *String*) : *Option* ⟨*Expr*⟩ =
  $\mathit{parse\text{-}expr}_0$ *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; $\mathit{parse\text{-}expr}_0$ *end-of-input* überführt die Liste von Tokens in einen abstrakten Syntaxbaum.

33. Parser — Demo

```
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
```

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.