31. Knobelaufgabe #16

Einem Wort über dem Alphabet $\{\mathtt{a}, \mathtt{b}\}$ kann man ansehen, ob es eine gerade Anzahl von as enthält (bs beliebig).

Einer *endlichen* Sprache kann man ansehen, ob *alle* in der Sprache enthaltenen Wörter eine gerade Anzahl von as enthalten.

Kann man einem regulären Ausdruck ansehen, ob alle in der von dem regulären Ausdruck bezeichneten Sprache enthaltenen Wörter eine gerade Anzahl von as enthalten?

31. Akzeptoren

- Ein regulärer Ausdruck r beschreibt eine Sprache.
- Wie können wir feststellen, ob ein gegebenes Wort w in der Sprache enthalten ist: $w \in [\![r]\!]$?
- Können wir ein Mini–F# Programm schreiben, das uns diese Aufgabe abnimmt?
- Ja! Ein solches Programm nennt man *Akzeptor*.
- Im Folgenden führen wir die Konstruktion exemplarisch für den regulären Ausdruck

$$r_{00} = (\mathtt{a} \mid \mathtt{b})^* \; \mathtt{b} \; (\mathtt{a} \mid \mathtt{b})$$

durch. Der Ausdruck beschreibt die Sprache aller Wörter, die an der vorletzten Position ein $\mathtt{b}$ haben.

31. Akzeptoren — Schnittstelle

Wir gehen davon aus, dass das Alphabet durch eine Variantentypdefinition gegeben ist.

```
type Alphabet = | A | B

let ord-Alphabet (a : Alphabet) : Int =
  match a with A → 0 | B → 1
```

☞ Zu einem Alphabet gehört eine injektive Funktion, die sogenannte Codierungsfunktion, die jedem Zeichen eine ganze Zahl zuordnet, den sogenannten Zeichencode.

Ein Akzeptor ist dann eine Funktion des Typs

$$\textit{accept-}r_{00} : \textit{List}\ \langle \textit{Alphabet} \rangle \to \textit{Bool}$$

☞ *accept-*r_{00} testet, ob die Eingabe w ein Element der durch r_{00} bezeichneten Sprache ist: $w \in [\![r_{00}]\!]$.

31. Akzeptoren — Implementierung

Mit dem Struktur Entwurfsmuster für *List* erhalten wir:

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{accept\text{-}r_{00}}\ (\mathit{input} : \mathit{List}\ \langle \mathit{Alphabet} \rangle) : \mathit{Bool} = \\
\quad \textbf{\textit{match}}\ \mathit{input}\ \textbf{\textit{with}} \\
\quad\quad \mid [\,] \qquad\quad \to \ldots \\
\quad\quad \mid A :: \mathit{rest} \to \ldots\ \mathit{accept\text{-}r_{00}}\ \mathit{rest} \ldots \\
\quad\quad \mid B :: \mathit{rest} \to \ldots\ \mathit{accept\text{-}r_{00}}\ \mathit{rest} \ldots
\end{array}
$$

☞ Der rekursive Aufruf von $\mathit{accept\text{-}r_{00}}$ hilft uns an dieser Stelle *nicht* weiter: aus $w \in [\![r_{00}]\!]$ können wir im Allgemeinen keine Rückschlüsse auf $a\ w \in [\![r_{00}]\!]$ ziehen.

31. Implementierung — Rekursionsbasis

Im Fall $input = []$ müssen wir überlegen, ob das leere Wort in der Sprache enthalten ist: $\epsilon \in [\![r]\!]$. Können wir dem regulären Ausdruck das ansehen?

- $[\![a]\!]$ enthält ϵ nicht;
- $[\![\epsilon]\!]$ enthält ϵ;
- $[\![r_1\ r_2]\!]$ enthält ϵ, wenn $[\![r_1]\!]$ *und* $[\![r_2]\!]$ das leere Wort enthalten;
- $[\![\emptyset]\!]$ enthält ϵ nicht;
- $[\![r_1 \mid r_2]\!]$ enthält ϵ, wenn $[\![r_1]\!]$ *oder* $[\![r_2]\!]$ das leere Wort enthalten;
- $[\![r^*]\!]$ enthält ϵ.

31. Enthält eine Sprache das leere Wort?

Formalisierung:

$$
\begin{array}{ll}
\mathit{nullable}(a) & = \mathit{false} \\
\mathit{nullable}(\epsilon) & = \mathit{true} \\
\mathit{nullable}(r_1\ r_2) & = \mathit{nullable}(r_1) \wedge \mathit{nullable}(r_2) \\
\mathit{nullable}(\emptyset) & = \mathit{false} \\
\mathit{nullable}(r_1 \mid r_2) & = \mathit{nullable}(r_1) \vee \mathit{nullable}(r_2) \\
\mathit{nullable}(r^*) & = \mathit{true}
\end{array}
$$

☞ *nullable* ist eine mathematische Funktion, kein Mini–F# Programm.

☞ Wie auch $[\![-]\!]$ orientiert sich *nullable* an der Struktur regulärer Ausdrücke.

☞ Eine semantische Eigenschaft, $\epsilon \in [\![r]\!]$, wird anhand der Syntax geklärt, $\mathit{nullable}(r)$.

31. Enthält eine Sprache das leere Wort?

Für r_{00} erhalten wir:

$$\begin{aligned} & \mathit{nullable}((\mathtt{a} \mid \mathtt{b})^*\ \mathtt{b}\ (\mathtt{a} \mid \mathtt{b})) \\ = \quad & \mathit{nullable}((\mathtt{a} \mid \mathtt{b})^*) \wedge \mathit{nullable}(\mathtt{b}\ (\mathtt{a} \mid \mathtt{b})) \\ = \quad & \mathit{nullable}(\mathtt{b}\ (\mathtt{a} \mid \mathtt{b})) \\ = \quad & \mathit{nullable}(\mathtt{b}) \wedge \mathit{nullable}(\mathtt{a} \mid \mathtt{b}) \\ = \quad & \mathit{false} \end{aligned}$$

☞ ϵ ist *nicht* in $[\![r_{00}]\!]$ enthalten.

31. Implementierung — Rekursionsschritt

Somit können wir den ersten Zweig von $\mathit{accept\text{-}r}_{00}$ mit Leben füllen:

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{accept\text{-}r}_{00}\ (\mathit{input} : \mathit{List}\ \langle \mathit{Alphabet} \rangle) : \mathit{Bool} = \\
\quad \textbf{\textit{match}}\ \mathit{input}\ \textbf{\textit{with}} \\
\quad\ \mid [\,] \qquad\ \to \mathit{false} \\
\quad\ \mid A :: \mathit{rest} \to \ldots \mathit{accept\text{-}r}_{00}\ \mathit{rest} \ldots \\
\quad\ \mid B :: \mathit{rest} \to \ldots \mathit{accept\text{-}r}_{00}\ \mathit{rest} \ldots
\end{array}
$$

☞ Im Fall $\mathit{input} = A :: \mathit{rest}$ wäre es nützlich zu wissen, was von der Sprache „übrigbleibt“, jetzt da wir a bereits gesehen haben. Dann könnten wir die Arbeit an eine weitere Funktion delegieren, den Akzeptor für die „Restsprache“.

31. Rechtsfaktoren

Diese „Restsprache“ heißt im Fachjargon *Rechtsfaktor* und ist für ein Wort w wie folgt definiert:

$$L \,/\, w = \{x \mid w\, x \in L\}$$

☞ Der Rechtsfaktor $L \,/\, w$ (lies: L durch w) ist die Menge aller Restworte x, so dass $w\, x$ in L enthalten ist.

☞ Wenn L eine reguläre Sprache ist, ist dann $L \,/\, w$ ebenfalls regulär? Zunächst einmal gilt

$$L \,/\, \epsilon \quad = L$$
$$L \,/\, a\, w = (L \,/\, a) \,/\, w$$

☞ Somit können wir uns auf die Definition von $L \,/\, a$ konzentrieren.

31. Rechtsfaktoren

Ziel: Definition von '/' auf regulären Ausdrücken, so dass

$$[\![r \,/\, x]\!] = [\![r]\!] \,/\, x$$

gilt. Eine semantische Operation auf Sprachen, $L \,/\, x$, wird auf reguläre Ausdrücke (Syntax), $r \,/\, x$, übertragen.

Schwierigster Fall: Konkatenation $r_1\ r_2 \,/\, x$.

- Entweder wir entfernen x aus r_1
- oder aus r_2; das geht aber nur, wenn $\epsilon \in [\![r_1]\!]$. Wir definieren:

$$\Delta(r) \quad = \quad \begin{cases} \epsilon & \text{falls } \mathit{nullable}(r) \\ \emptyset & \text{sonst} \end{cases}$$

31. Rechtsfaktoren

Damit erhalten wir

$$
\begin{array}{lcl}
a \,/\, x & = & \begin{cases} \epsilon & \text{falls } a = x \\ \emptyset & \text{sonst} \end{cases} \\
\epsilon \,/\, x & = & \emptyset \\
r_1\ r_2 \,/\, x & = & (r_1 \,/\, x)\ r_2 \mid \Delta(r_1)\ (r_2 \,/\, x) \\
\emptyset \,/\, x & = & \emptyset \\
r_1 \mid r_2 \,/\, x & = & (r_1 \,/\, x) \mid (r_2 \,/\, x) \\
r^* \,/\, x & = & (r \,/\, x)\ r^*
\end{array}
$$

☞ Die Definition des Rechtsfaktors ist der *Ableitung von Funktionen* sehr ähnlich (lies ϵ als 1, $\emptyset$ als 0, die Alternative als Summe und die Konkatenation als Produkt).

☞ Wir sehen: es war eine gute Idee, $\emptyset$ mit zur Sprache hinzuzunehmen.

31. Rechtsfaktoren

Wie sehen die Rechtsfaktoren für unser Beispiel aus?

$$
\begin{aligned}
& r_{00} \,/\, \mathtt{a} \\
= \;& (\mathtt{a} \mid \mathtt{b})^* \; \mathtt{b} \; (\mathtt{a} \mid \mathtt{b}) \,/\, \mathtt{a} \\
= \;& ((\mathtt{a} \mid \mathtt{b})^* \,/\, \mathtt{a}) \; (\mathtt{b} \; (\mathtt{a} \mid \mathtt{b})) \mid (\Delta((\mathtt{a} \mid \mathtt{b})^*)) \; (\mathtt{b} \; (\mathtt{a} \mid \mathtt{b}) \,/\, \mathtt{a}) \\
= \;& ((\mathtt{a} \mid \mathtt{b})^* \,/\, \mathtt{a}) \; (\mathtt{b} \; (\mathtt{a} \mid \mathtt{b})) \mid (\mathtt{b} \; (\mathtt{a} \mid \mathtt{b}) \,/\, \mathtt{a}) \\
= \;& ((\mathtt{a} \mid \mathtt{b})^* \,/\, \mathtt{a}) \; (\mathtt{b} \; (\mathtt{a} \mid \mathtt{b})) \mid ((\mathtt{b} \,/\, \mathtt{a}) \; (\mathtt{a} \mid \mathtt{b})) \mid ((\Delta \mathtt{b}) \; ((\mathtt{a} \mid \mathtt{b}) \,/\, \mathtt{a})) \\
= \;& ((\mathtt{a} \mid \mathtt{b})^* \,/\, \mathtt{a}) \; (\mathtt{b} \; (\mathtt{a} \mid \mathtt{b})) \\
= \;& (\mathtt{a} \mid \mathtt{b} \,/\, \mathtt{a}) \; (\mathtt{a} \mid \mathtt{b})^* \; \mathtt{b} \; (\mathtt{a} \mid \mathtt{b}) \\
= \;& ((\mathtt{a} \,/\, \mathtt{a}) \mid (\mathtt{b} \,/\, \mathtt{a})) \; (\mathtt{a} \mid \mathtt{b})^* \; \mathtt{b} \; (\mathtt{a} \mid \mathtt{b}) \\
= \;& (\mathtt{a} \mid \mathtt{b})^* \; \mathtt{b} \; (\mathtt{a} \mid \mathtt{b}) \\
= \;& r_{00}
\end{aligned}
$$

☞ $r_{00} \,/\, \mathtt{a} = r_{00}$!

31. Rechtsfaktoren

Fortsetzung:

$$
\begin{aligned}
& r_{00} \,/\, \mathtt{b} \\
= \;& (\mathtt{a} \mid \mathtt{b})^* \; \mathtt{b} \; (\mathtt{a} \mid \mathtt{b}) \,/\, \mathtt{b} \\
= \;& ((\mathtt{a} \mid \mathtt{b})^* \,/\, \mathtt{b}) \; (\mathtt{b} \; (\mathtt{a} \mid \mathtt{b})) \mid (\Delta((\mathtt{a} \mid \mathtt{b})^*)) \; (\mathtt{b} \; (\mathtt{a} \mid \mathtt{b}) \,/\, \mathtt{b}) \\
= \;& ((\mathtt{a} \mid \mathtt{b})^* \,/\, \mathtt{b}) \; (\mathtt{b} \; (\mathtt{a} \mid \mathtt{b})) \mid (\mathtt{b} \; (\mathtt{a} \mid \mathtt{b}) \,/\, \mathtt{b}) \\
= \;& ((\mathtt{a} \mid \mathtt{b})^* \,/\, \mathtt{b}) \; (\mathtt{b} \; (\mathtt{a} \mid \mathtt{b})) \mid ((\mathtt{b} \,/\, \mathtt{b}) \; (\mathtt{a} \mid \mathtt{b})) \mid ((\Delta \mathtt{b}) \; ((\mathtt{a} \mid \mathtt{b}) \,/\, \mathtt{b})) \\
= \;& ((\mathtt{a} \mid \mathtt{b})^* \,/\, \mathtt{b}) \; (\mathtt{b} \; (\mathtt{a} \mid \mathtt{b})) \mid (\mathtt{a} \mid \mathtt{b}) \\
= \;& (\mathtt{a} \mid \mathtt{b} \,/\, \mathtt{b}) \; (\mathtt{a} \mid \mathtt{b})^* \; \mathtt{b} \; (\mathtt{a} \mid \mathtt{b}) \mid (\mathtt{a} \mid \mathtt{b}) \\
= \;& ((\mathtt{a} \,/\, \mathtt{b}) \mid (\mathtt{b} \,/\, \mathtt{b})) \; (\mathtt{a} \mid \mathtt{b})^* \; \mathtt{b} \; (\mathtt{a} \mid \mathtt{b}) \mid (\mathtt{a} \mid \mathtt{b}) \\
= \;& (\mathtt{a} \mid \mathtt{b})^* \; \mathtt{b} \; (\mathtt{a} \mid \mathtt{b}) \mid (\mathtt{a} \mid \mathtt{b}) \\
= \;& r_{00} \mid (\mathtt{a} \mid \mathtt{b})
\end{aligned}
$$

☞ Wir taufen den resultierenden Ausdruck r_{10}.

31. Implementierung — Rekursionsschritt

Damit können wir endlich die Definition von $\mathit{accept}\text{-}r_{00}$ vervollständigen.

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{accept}\text{-}r_{00}\ (\mathit{input} : \mathit{List}\ \langle \mathit{Alphabet} \rangle) : \mathit{Bool} = \\
\quad \textbf{\textit{match}}\ \mathit{input}\ \textbf{\textit{with}} \\
\quad\ \mid [\,] \qquad\ \to \mathit{false} \\
\quad\ \mid A :: \mathit{rest} \to \mathit{accept}\text{-}r_{00}\ \mathit{rest} \\
\quad\ \mid B :: \mathit{rest} \to \mathit{accept}\text{-}r_{10}\ \mathit{rest}
\end{array}
$$

☞ Ist das erste Zeichen ein A, dann erfolgt ein rekursiver Aufruf;
ist das Zeichen ein B, dann wird die weitere Arbeit an $\mathit{accept}\text{-}r_{10}$ delegiert.

31. Implementierung

Es verbleibt einen Akzeptor für r_{10} zu schreiben. Diese Aufgabe gehen wir nach dem gleichen Schema wie für r_{00} an.

$r_{10} = r_{00} \mid (\mathtt{a} \mid \mathtt{b})$
$\mathit{nullable}(r_{10}) = \mathit{false}$
$r_{10} \,/\, \mathtt{a} = r_{00} \mid \epsilon \qquad = r_{01}$
$r_{10} \,/\, \mathtt{b} = r_{00} \mid (\mathtt{a} \mid \mathtt{b}) \mid \epsilon = r_{11}$

Wir erhalten zwei neue reguläre Ausdrücke und rechnen weiter ...

31. Implementierung

$r_{01} = r_{00} \mid \epsilon$

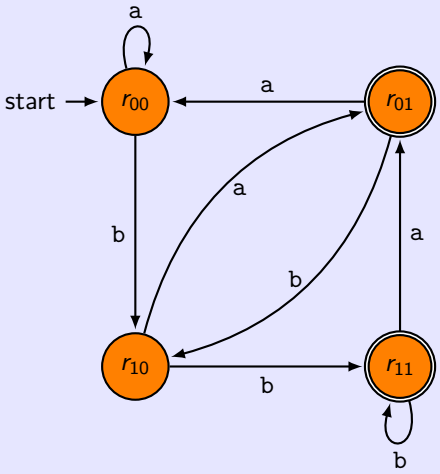
$nullable(r_{01}) = true$
$r_{01} \,/\, \mathtt{a} = r_{00}$
$r_{01} \,/\, \mathtt{b} = r_{10}$

$r_{11} = r_{00} \mid (\mathtt{a} \mid \mathtt{b}) \mid \epsilon$

$nullable(r_{11}) = true$
$r_{11} \,/\, \mathtt{a} = r_{01}$
$r_{11} \,/\, \mathtt{b} = r_{11}$

☞ Die Aufrufstruktur ist chaotisch: die Funktion *accept-*r_{00} ruft *accept-*r_{10} auf, diese ruft *accept-*r_{11} auf, diese ruft *accept-*r_{01} auf und diese wiederum *accept-*r_{00}.

31. Implementierung — Aufrufgraph



☞ Die vier Akzeptoren sind *verschränkt rekursiv*: die Definitionen müssen mit dem Schlüsselwort ***and*** verbunden werden.

31. Implementierung

```
let rec accept-r00 (input : List ⟨Alphabet⟩) : Bool =
  match input with | []        → false
                   | A :: rest → accept-r00 rest
                   | B :: rest → accept-r10 rest
and accept-r01 (input : List ⟨Alphabet⟩) : Bool =
  match input with | []        → true
                   | A :: rest → accept-r00 rest
                   | B :: rest → accept-r10 rest
and accept-r10 (input : List ⟨Alphabet⟩) : Bool =
  match input with | []        → false
                   | A :: rest → accept-r01 rest
                   | B :: rest → accept-r11 rest
and accept-r11 (input : List ⟨Alphabet⟩) : Bool =
  match input with | []        → true
                   | A :: rest → accept-r01 rest
                   | B :: rest → accept-r11 rest
```

r_{00}, r_{10}, r_{01}, r_{11}? Ich blick da nicht mehr durch. Es geht doch darum, zu erkennen, ob das vorletzte Zeichen ein b ist. Dann merke ich mir einfach immer die beiden letzten Zeichen und überprüfe am Ende, ob das vorletzte Zeichen gleich b ist.

Genau das machen r_{00}, r_{10}, r_{01}, r_{11}!

Kapier ich nicht.

Schau Dir die regulären Ausdrücke genau an:

$$
\begin{array}{lcl}
r_{00} & = & (\mathtt{a} \mid \mathtt{b})^* \, \mathtt{b} \, (\mathtt{a} \mid \mathtt{b}) \mid \emptyset \mid \emptyset \\
r_{10} & = & (\mathtt{a} \mid \mathtt{b})^* \, \mathtt{b} \, (\mathtt{a} \mid \mathtt{b}) \mid (\mathtt{a} \mid \mathtt{b}) \mid \emptyset \\
r_{01} & = & (\mathtt{a} \mid \mathtt{b})^* \, \mathtt{b} \, (\mathtt{a} \mid \mathtt{b}) \mid \emptyset \mid \epsilon \\
r_{11} & = & (\mathtt{a} \mid \mathtt{b})^* \, \mathtt{b} \, (\mathtt{a} \mid \mathtt{b}) \mid (\mathtt{a} \mid \mathtt{b}) \mid \epsilon
\end{array}
$$

(a ∣ b) besagt „das letzte Zeichen war ein b“ (danach muss *ein* beliebiges Zeichen kommen); ϵ besagt „das vorletzte Zeichen war ein b“ (danach muss das Eingabeende kommen).

Die Nummerierung r_{ij} spiegelt das übrigens wieder: $i = 1$ gdw. das letzte Zeichen ein b war; $j = 1$ gdw. das vorletzte Zeichen ein b war. *Zum Knobeln:* Wieviele Rechtsfaktoren hat $(\mathtt{a} \mid \mathtt{b})^* \; \mathtt{b} \; (\mathtt{a} \mid \mathtt{b})^n$?

So langsam dämmert's. Aber meine Lösung finde ich trotzdem einfacher.

Für dieses spezielle Beispiel magst Du Recht haben. Der Punkt ist aber, dass die Konstruktion der Akzeptoren *mechanisch* — ohne Nachdenken — funktioniert. Sie klappt für beliebige reguläre Ausdrücke, eben nicht nur für dieses Beispiel. Die kompliziertere Herangehensweise zahlt sich durch die größere Allgemeinheit aus!

Ok, ok, ich gebe mich geschlagen. Aber einfacher ist sie trotzdem ...

31. Rechtsfaktoren — Eigenschaften

- Der reguläre Ausdruck $(\mathtt{a} \mid \mathtt{b})^* \; \mathtt{b} \; (\mathtt{a} \mid \mathtt{b})$ hat vier verschiedene Rechtsfaktoren.
- Der Ausdruck $(\mathtt{a} \; \mathtt{b}^* \; \mathtt{a} \mid \mathtt{b})^*$ hat zwei (welche?).
- Allgemein: jede reguläre Sprache hat nur *endlich* viele Rechtsfaktoren.
- Diese Eigenschaft können wir ausnutzen, um zu zeigen, dass eine Sprache *nicht* durch einen regulären Ausdruck bezeichnet werden kann.
- Die Sprache $L = \{\mathtt{a}^n \; \mathtt{b}^n \mid n \in \mathbb{N}\}$ ist ein Beispiel für eine solche Sprache. Die Rechtsfaktoren für Wörter der Form $\mathtt{a}^k$ sind gegeben durch:

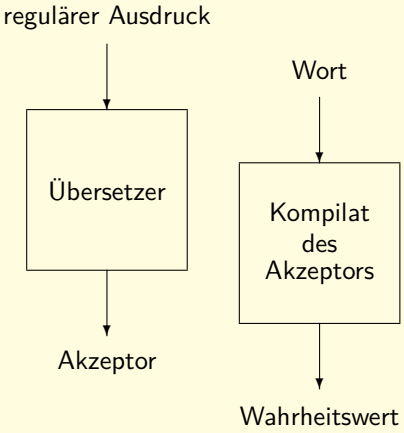
$$L \,/\, \mathtt{a}^k = \{\mathtt{a}^n \; \mathtt{b}^{n+k} \mid n \in \mathbb{N}\}$$

Alle diese Sprachen sind verschieden, also ist L keine reguläre Sprache.

31. Übersetzer und Interpreter

- Da die Anzahl aller Rechtsfaktoren endlich ist, kann man das obige Verfahren verwenden, um aus einem regulären Ausdruck einen Akzeptor zu generieren.
- *Mehr noch:* wir können das Verfahren auch automatisieren: wir können ein Mini–F# Programm schreiben, das uns diese Aufgabe abnimmt!
- Dieses Mini–F# Programm würde
  - als Eingabe einen regulären Ausdruck verarbeiten (in konkreter oder abstrakter Syntax) und
  - als Ausgabe ein Mini–F# Programm, einen Akzeptor für den regulären Ausdruck, erzeugen (in konkreter Syntax).
- Ein Programm, das ein anderes Programm erzeugt, nennt man *Übersetzer* (engl. compiler).
- In diesem Fall würde man auch von einem *Scanner-Generator* sprechen.
- (Ein bekannter Scanner-Generator ist `lex`).

31. Übersetzer und Interpreter



31. Übersetzer und Interpreter

- Der Übersetzer und der generierte Akzeptor sind zwei getrennte Programme.
- Der Übersetzer erzeugt ein Programm, das in einem zweiten Schritt ausgeführt wird.
- Die beiden Programme können alternativ auch enger miteinander verzahnt werden.
- Die Struktur der einzelnen Akzeptoren ist identisch — das ist nicht weiter verwunderlich, wir haben sie ja nach dem gleichen Schema konstruiert.
- Lassen sich die vier Funktionen zu einer zusammenfassen?
- *Idee:* Wir nummerieren die Funktionen durch und machen die Hausnummer zu einem zusätzlichen Parameter.

$$accept\ (k : Nat, input : List\ \langle Alphabet \rangle) : Bool$$

- Diese allgemeine Funktion müssen wir mit Informationen ausstatten, welcher Wahrheitswert im [] Zweig zurückgegeben wird, und welche Hausnummer als nächstes an der Reihe ist, wenn ein a bzw. ein b gesehen wurde.

31. Übersetzer und Interpreter

Aus einer Kontrollstruktur wird eine Datenstruktur!

$$\begin{aligned}&\textbf{\textit{type}}\ \mathit{Control} = \{\,\mathit{nullable} : \mathit{Array}\,\langle \mathit{Bool}\rangle;\\ &\qquad\qquad\quad \mathit{next}\quad\ : \mathit{Array}\,\langle \mathit{Array}\,\langle \mathit{Nat}\rangle\rangle\,\}\end{aligned}$$

Für unser laufendes Beispiel erhalten wir die folgenden Daten:

$$\begin{aligned}&\textbf{\textit{let}}\ \mathit{control}\text{-}r_{00} = \{\,\mathit{nullable} = [|\ \mathit{false}; \mathit{true}; \mathit{false}; \mathit{true}\ |];\\ &\qquad\qquad\quad \mathit{next}\quad\ = [|\ [|\ 0; 0; 1; 1\ |];\qquad (*\ A\ *)\\ &\qquad\qquad\qquad\qquad\ \ [|\ 2; 2; 3; 3\ |]\ |]\}\qquad (*\ B\ *)\end{aligned}$$

☞ Die Funktionen sind fortlaufend von oben nach unten beginnend mit 0 durchnummeriert.

31. Übersetzer und Interpreter

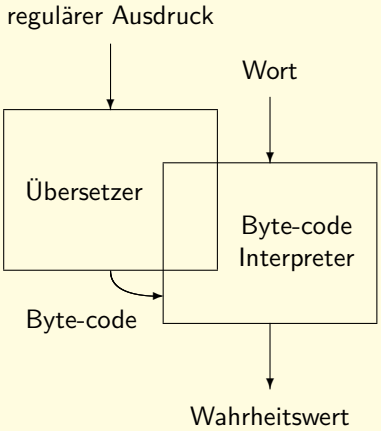
Ein generischer Akzeptor:

```
let generic-accept (control : Control) : List ⟨Alphabet⟩ → Bool =
  let rec accept (k : Nat, input : List ⟨Alphabet⟩) : Bool =
    match input with
    | []        → control.nullable.[k]
    | a :: rest → accept (control.next.[ord-Alphabet a].[k], rest)
  in
    fun input → accept (0, input)
let accept-r00 = generic-accept control-r00
```

31. Übersetzer und Interpreter

- Der Übersetzer für reguläre Ausdrücke — den wir nicht angegeben haben — würde in diesem Szenario kein Mini–F# Programm erzeugen, sondern ein Element des Typs *Control*.
- Der allgemeine Akzeptor *generic-accept interpretiert* diese Kontrollinformation, um zu einem gegebenen Wort zu entscheiden, ob es in der Sprache enthalten ist oder nicht.
- Bei dem allgemeinen Akzeptor handelt es sich somit um einen Interpreter, genauer um einen *Byte-code Interpreter*.
- Byte-code deswegen, weil die regulären Ausdrücke durch kleine Zahlen repräsentiert werden und die Funktionen *nullable* und $r \,/\, x$ mit Hilfe von *done* und *next* codiert werden.
- Da die beiden Programme, der Übersetzer und der Byte-code Interpreter, über eine Datenstruktur miteinander kommunizieren, können sie in einem Programm zusammengefasst werden.

31. Übersetzer und Interpreter



31. Übersetzer und Interpreter

- Der Übersetzer generiert Byte-code, den der Byte-code Interpreter abarbeitet.
- Der Übersetzer hat sozusagen die Aufgabe reguläre Ausdrücke *vorzuverdauen*.
- Alternativ können wir einen Akzeptor schreiben, der *direkt* auf den regulären Ausdrücken arbeitet, sozusagen auf den *unverdauten* Eingaben.
- Aus dem Byte-code Interpreter wird ein „echter“ Interpreter.
- Zu diesem Zweck müssen wir
  - die regulären Ausdrücke durch einen Datentyp modellieren und
  - die Funktionen *nullable* und $r \,/\, x$ in Mini–F# Programme überführen.

31. Interpreter

Wir transliterieren die abstrakte Syntax regulärer Ausdrücke in einen Variantentyp.

```
type Reg =
    | Eps                  // das leere Wort
    | Sym of Alphabet      // einzelnes Zeichen ≙ Terminalsymbol
    | Cat of Reg * Reg     // Konkatenation ≙ Sequenz
    | Empty                // die leere Sprache
    | Alt of Reg * Reg     // Alternative
    | Rep of Reg           // Wiederholung
```

Der reguläre Ausdruck r_{00} kann direkt in Mini–F# definiert werden.

```
let any = Alt (Sym A, Sym B)
let r00 = Cat (Rep any, Cat (Sym B, any))
```

31. Interpreter

Die Funktion *nullable* lässt sich direkt übertragen: die sechs Gleichungen werden zu den sechs Zweigen eines ***match***-Ausdrucks.

```
let rec nullable (reg : Reg) : Bool =
  match reg with
    | Eps          → true
    | Sym _        → false
    | Cat (r1, r2) → nullable r1 && nullable r2
    | Empty        → false
    | Alt (r1, r2) → nullable r1 || nullable r2
    | Rep r        → true
```

31. Interpreter

Ähnlich direkt ist die Umsetzung der mathematischen Funktion $r \,/\, x$.

```
let rec divide (reg : Reg, x : Alphabet) : Reg =
  match reg with
  | Eps          → Empty
  | Sym a        → if x = a then Eps else Empty
  | Cat (r1, r2) → if nullable r1
                   then alt (cat (divide (r1, x), r2), divide (r2, x))
                   else      cat (divide (r1, x), r2)
  | Empty        → Empty
  | Alt (r1, r2) → alt (divide (r1, x), divide (r2, x))
  | Rep r        → cat (divide (r, x), Rep r)
```

31. Interpreter — Demo


```
Mini⟩ nullable r00
false
Mini⟩ divide (r00, A)
Alt (Cat (Cat (Eps, Rep (Alt (Sym A, Sym B))),
          Cat (Sym B, Alt (Sym A, Sym B))),
     Cat (Empty, Alt (Sym A, Sym B)))
Mini⟩ divide (r00, B)
Alt (Cat (Cat (Eps, Rep (Alt (Sym A, Sym B))),
          Cat (Sym B, Alt (Sym A, Sym B))),
     Cat (Eps, Alt (Sym A, Sym B)))
```

☞ *divide* (r_{00}, A) ist semantisch äquivalent zu r_{00}, aber nicht syntaktisch gleich. Das liegt daran, dass bei der Konstruktion des Rechtsfaktors keine algebraischen Vereinfachungen wie etwa $r\ \epsilon = r$ vorgenommen werden.

31. Interpreter

Der generische Akzeptor:

```
let rec generic-accept (reg : Reg, input : List ⟨Alphabet⟩) : Bool =
  match input with
  | []        → nullable reg
  | a :: rest → generic-accept (divide (reg, a), rest)
```

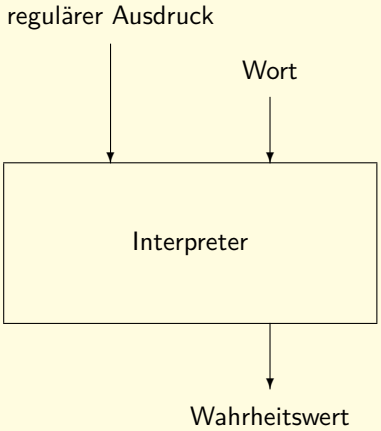
☞ An die Stelle der Hausnummern sind die regulären Ausdrücke selbst getreten.

```
let accept-r00 = fun input → generic-accept (r00, input)
```

31. Interpreter — Demo

```
Mini⟩ accept-r00 (A :: A :: A :: [])
false
Mini⟩ accept-r00 (A :: B :: A :: [])
true
Mini⟩ let even-no-of-as =
          Rep (Alt (Cat (Sym A, Cat (Rep (Sym B), Sym A)), Sym B))
val even-no-of-as : Reg
Mini⟩ generic-accept (even-no-of-as, A :: B :: A :: [])
true
Mini⟩ generic-accept (even-no-of-as, A :: B :: B :: [])
false
```

31. Übersetzer und Interpreter



31. Übersetzer und Interpreter

Aus einem regulären Ausdruck lässt sich automatisch ein passender Akzeptor konstruieren. Je nach Verzahnungsgrad kann man mindestens drei Ansätze unterscheiden:

- reiner Übersetzer,
- Mischform aus Übersetzer und Interpreter,
- reiner Interpreter.

☞ Übersetzer und Interpreter sind keineswegs spezifisch für reguläre Ausdrücke, sondern stellen allgemeine Konzepte dar.

Im allgemeinen übersetzt ein Übersetzer ein Wort einer Sprache, der Quellsprache, in ein Wort einer anderen Sprache, der Zielsprache. Ein Interpreter interpretiert ein Wort direkt.

31. Lösung Knobelaufgabe #15

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

Ja! Wir zeigen, wie man die Fakultät ohne Rekursion programmieren kann. *Idee:* wir machen den rekursiven Aufruf zum Parameter und rufen die resultierende Funktion mit sich selbst auf (Selbstapplikation).

$$
\begin{array}{l}
\textbf{\textit{let}}\ \mathit{fac}\ (\mathit{self}, n : \mathit{Nat}) : \mathit{Nat} = \\
\quad \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ 1 \\
\qquad\qquad \textbf{\textit{else}}\ \ \mathit{self}\ (\mathit{self}, n - 1) * n \\
\textbf{\textit{let}}\ \mathit{factorial}\ (n : \mathit{Nat}) : \mathit{Nat} = \\
\quad \mathit{fac}\ (\mathit{fac}, n)
\end{array}
$$

31. Lösung Knobelaufgabe #15

Was ist der Typ des formalen Parameters *self*?

```
let fac (self, n : Nat) : Nat =
  if n = 0 then 1
           else self (self, n − 1) ∗ n
```

☞ Sei $self : t$. Da *self* auf sich selbst angewendet wird, muss gelten: $t = t * Nat \to Nat$.

31. Lösung Knobelaufgabe #15

☞ Wir können keinen Typ t definieren, der *gleich* dem Typ $t * Nat \to Nat$ ist.

Aber wir können einen Typ definieren, der *isomorph* zu dem Typ $t * Nat \to Nat$ ist.

$$
\begin{array}{l}
\textbf{\textit{type}}\ Factorial = \{\, apply : Factorial * Nat \to Nat \,\} \\
\textbf{\textit{let}}\ fac\ (self : Factorial, n : Nat) : Nat = \\
\quad \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}}\ 1 \\
\qquad\qquad \textbf{\textit{else}}\ \ self.apply\ (self, n - 1) * n \\
\textbf{\textit{let}}\ factorial\ (n : Nat) : Nat = \\
\quad fac\ (\{\, apply = fac \,\}, n)
\end{array}
$$

☞ $\textbf{\textit{fun}}\ f \to f.apply$ und $\textbf{\textit{fun}}\ g \to \{\, apply = g \,\}$ sind die Isomorphismen zwischen $Factorial$ und $Factorial * Nat \to Nat$.