35. Knobelaufgabe #20

Knut Don ist auf die Idee gekommen, natürliche Zahlen durch Bäume zu repräsentieren.

```
type Tree = | Tip | Fork of Tree * Tree
```

Der leere Baum stellt die Zahl 0 dar, der Baum *Fork* t_1 t_2 repräsentiert $2^{n_1} + n_2$, wobei n_i die Bedeutung von t_i ist. Als Mini–F# Programm:

```
let rec nat (t : Tree) : Nat =
  match t with
    | Tip          → 0
    | Fork (l, r) → power (2, nat l) + nat r
```

Eine natürliche Zahl hat viele Darstellungen: *Fork* l_1 (*Fork* l_2 r) und *Fork* l_2 (*Fork* l_1 r) bezeichnen z. B. stets die gleiche Zahl. Für das Rechnen ist es vorteilhaft, eine *eindeutige* Repräsentation zu haben. Deswegen fordern wir, dass stets $2^{n_1} > n_2$ gilt.

Implementieren Sie die arithmetischen und die Vergleichsoperationen für diese merkwürdige Zahlendarstellung.

35. Pascal und C

☞ In den Programmiersprachen Pascal (1970 geboren) und C (1972 geboren) spielen Speicherzellen eine weitaus größere Rolle als in Mini–F#.

Es wird gerechnet, indem Inhalte von Speicherzellen schrittweise verändert werden.

Schauen wir uns die Unterschiede etwas genauer an ...

35. Pascal und C: Variablen

Mini–F#:

```
let i = ref 0
```

Pascal:

```
var i:integer;
```

☞ Initialisierung nicht möglich!

C:

```
int i = 0;
```

☞ Initialisierung nicht nötig!

35. Pascal und C: Konstanten

Mini–F#:

```
let n = 4711
```

Pascal:

```
const n = 4711;
```

C:

```
const int n = 0;
```

35. Pascal und C: Funktionen

Mini–F#:

let *succ* $(n : Nat) = n + 1$

Pascal:

```
function succ (n : integer):integer;
begin
  succ := n + 1
end;
```

C:

```
int succ (int n) {
  return (n + 1);
}
```

☞ Funktionen können nur deklariert werden. In C dürfen Funktionen nicht lokal deklariert werden; in Pascal dürfen Funktionen nicht als Ergebnis zurückgegeben werden.

35. Pascal und C: Zuweisung

Mini–F#:

$$i := !i + 4711$$

Pascal:

```
i := i + 4711;
```

C:

```
i = i + 4711
```

☞ In Pascal und C werden auf der rechten Seite Speicherzellen automatisch dereferenziert: man spricht auch von L- und R-Werten einer Variablen.

Mini–F# hat übrigens eine alternative Notation für Speicherzellen, die Harry vielleicht entgegenkommt.

let mutable $funds = 0$

Mit ***mutable*** wird ein Bezeichner als veränderlich gekennzeichnet: aus einem Namen für einen Wert wird eine Speicherzelle. Zuweisungen an ***mutable***s werden wie folgt notiert.

$funds \leftarrow funds + amount$

Ein veränderlicher Bezeichner wird automatisch dereferenziert. *Lies:* $funds$ wird zu $funds + amount$ (engl. $funds$ becomes $funds + amount$). In Kürze mehr dazu.

35. Pascal und C

☞ In Pascal und C dürfen Funktionen keine aggregrierten Daten wie Paare oder Records zurückgeben.

Ausweg: Das Funktionsergebnis wird nicht über den Rückgabewert, sondern über das Argument kommuniziert!

Mini–F#:

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{factorial}\ (n : \mathit{Nat}, \mathit{result} : \mathit{Ref}\langle \mathit{Nat}\rangle) = \\
\quad \textbf{\textit{if}}\ n = 0\ \textbf{\textit{then}} \\
\qquad \mathit{result} := 1 \\
\quad \textbf{\textit{else}} \\
\qquad \mathit{factorial}\ (n \dot{-} 1, \mathit{result}); \\
\qquad \mathit{result} :=\ !\mathit{result} * n
\end{array}
$$

☞ Der Funktion wird die Adresse übergeben, unter der sie das Ergebnis ablegen soll — ähnlich einem Postfach.

35. Demo

```
Mini⟩ let post-office-box = ref 10
val post-office-box : Ref ⟨Nat⟩
Mini⟩ factorial (!post-office-box, post-office-box)
()
Mini⟩ !post-office-box
3628800
```

☞ Der zweite Ausdruck verwendet sowohl den L-Wert (*post-office-box*) als auch den R-Wert (!*post-office-box*) einer Variablen.

35. Pascal und C: call by reference

Pascal:

```
procedure factorial (n : integer, var result : integer);
begin
  if n=0 then
    result := 1
  else
    begin
      factorial (n - 1, result);
      result := result * n
    end
end;
```

☞ Der zweite Parameter ist ein *Variablen-* oder *Referenzparameter*.

```
factorial (pob, pob);
```

☞ Der erste Parameter wird *call by value* (R-Wert) übergeben, der zweite *call by reference* (L-Wert).

35. Pascal und C

C:

```
void factorial (int n, int* result) {
  if (n == 0)
    *result = 1;
  else {
    factorial (n - 1, result);
    *result *= n;
  }
}
```

☞ Der zweite Parameter ist ein Zeiger auf eine Zahl — Cs Terminologie für Adresse; * dereferenziert eine Adresse.

```
factorial (pob, &pob);
```

☞ & ist der Adressoperator; dieser macht die automatische Dereferenzierung rückgängig.

35. Lebensdauer

- Speicherzellen leben in Mini–F# prinzipiell ewig (der Speicher σ wird stets größer, nie kleiner).
- In Pascal und C endet die Lebensdauer in der Regel mit dem Ende der Sichtbarkeit: ist eine Variable nicht mehr sichtbar, wird sie deallokiert.
- *Hoffnung:* eine Speicherzelle wird nicht mehr benötigt, wenn ihre Adresse nicht mehr sichtbar ist.
- In Mini–F# gilt dies *nicht*: *funds* ist nach Abarbeitung des lokalen Moduls nicht mehr sichtbar, wird aber benötigt.
- In Pascal ist die Hoffnung berechtigt. *Aber:* Verlust an Ausdruckskraft.
- In C gilt dies *nicht* wegen des Adressoperators. *Trotzdem wird deallokiert.* ☞ dangling pointer.

35. Garbage collection

- Wann und wie werden in Mini–F# Speicherzellen deallokiert?
- Speicherzellen werden deallokiert, wenn sie tatsächlich nicht mehr benötigt werden und wenn der Speicherplatz knapp wird.
- Diese Aufgabe übernimmt eine sogenannter *Garbage collector* (engl. Müllmann), ein wichtiger Bestandteil des Mini–F# Interpreters bzw. des Laufzeitsystems.
- Pascal und C verfügen über keinen Garbage collector. Sie verwenden daher eine einfachere, aber durchaus bewährte Form der Speicherorganisation: Variablen am Anfang der Sichtbarkeit allokieren, am Ende deallokieren.

35. Pascal und C: Listen

☞ Da in Pascal und C das Konzept des Speichers dominierend ist, gehen diese Sprache auch die Implementierung von Datenstrukturen wie Listen oder Bäumen anders an. Wir programmieren eine typische Listenimplementierung in Mini–F# nach.

$$\begin{array}{lll} \textbf{type}\ \mathit{List}\,\langle'a\rangle & = \mathit{Ref}\,\langle\mathit{Item}\,\langle'a\rangle\rangle \\ \textbf{and}\ \ \mathit{Item}\,\langle'a\rangle & = \mid \mathit{Nil} \\ & \quad \mid \mathit{Cons}\ \textbf{of}\ {'a} * \mathit{List}\,\langle'a\rangle \end{array}$$

☞ Listen sowie sämtliche Restlisten sind Speicherzellen.

Zum Vergleich: die Listenimplementierung aus Kapitel 4.

$$\textbf{type}\ \mathit{List}\,\langle'a\rangle = \mathit{Nil} \mid \mathit{Cons}\ \textbf{of}\ {'a} * \mathit{List}\,\langle'a\rangle$$

35. Pascal und C: Listen

Listenoperationen arbeiten typischerweise „in situ“ (lat. am Platz) durch Modifikation der bestehenden Strukturen.

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{append}\ (\mathit{list1} : \mathit{List}\ \langle'a\rangle, \mathit{list2} : \mathit{List}\ \langle'a\rangle) = \\
\quad \textbf{\textit{match}}\ !\mathit{list1}\ \textbf{\textit{with}} \\
\qquad \mid \mathit{Nil} \qquad\quad \to \mathit{list1} := !\mathit{list2} \\
\qquad \mid \mathit{Cons}\ (x, xs) \to \mathit{append}\ (xs, \mathit{list2})
\end{array}
$$

☞ Es wird keine neue Liste konstruiert.

35. Pascal und C: Listen

Zum Vergleich: die Listenimplementierung aus Kapitel 4.

$$
\begin{array}{l}
\textbf{\textit{let rec}}\ \mathit{append}\ (\mathit{list1} : \mathit{List}\ \langle'a\rangle, \mathit{list2} : \mathit{List}\ \langle'a\rangle) : \mathit{List}\ \langle'a\rangle = \\
\quad \textbf{\textit{match}}\ \mathit{list1}\ \textbf{\textit{with}} \\
\quad\quad \mid \mathit{Nil} \qquad\quad \rightarrow \mathit{list2} \\
\quad\quad \mid \mathit{Cons}\ (x, xs) \rightarrow \mathit{Cons}\ (x, \mathit{append}\ (xs, \mathit{list2}))
\end{array}
$$

☞ *append* konstruiert eine Ergebnisliste.

35. Demo

let *primes* $=$ ***ref*** $(\mathit{Cons}\ (2, \textbf{\textit{ref}}\ (\mathit{Cons}\ (3, \textbf{\textit{ref}}\ \mathit{Nil}))))$
val *primes* : *List* $\langle \mathit{Nat} \rangle$
Mini$\rangle$ *primes*
$\{\mathit{contents} = \mathit{Cons}\ (2, \{\mathit{contents} = \mathit{Cons}\ (3, \{\mathit{contents} = \mathit{Nil}\})\})\}$
Mini$\rangle$ *append* $(\mathit{primes}, \textbf{\textit{ref}}\ (\mathit{Cons}\ (5, \textbf{\textit{ref}}\ (\mathit{Cons}\ (7, \textbf{\textit{ref}}\ \mathit{Nil})))))$
()
Mini$\rangle$ *primes*
$\{\mathit{contents} = \mathit{Cons}\ (2, \{\mathit{contents} = \mathit{Cons}\ (3, \{\mathit{contents} =$
$\quad \mathit{Cons}\ (5, \{\mathit{contents} = \mathit{Cons}\ (7, \{\mathit{contents} = \mathit{Nil}\})\})\})\})\})\}$

☞ *append* hängt das zweite Argument an das erste an und modifiziert dieses dabei. Die ursprüngliche Liste hat sich nach diesem Aufruf verflüchtigt.

☞ Aus diesem Grund spricht man auch von einer *ephemeren Datenstruktur* (griech. nur einen Tag dauernd, vorübergehend).

35. Pascal und C: Listen

Zum Vergleich: die Listenimplementierung aus Kapitel 4:

```
Mini⟩ let primes = Cons (2, Cons (3, Nil))
val primes : List ⟨Nat⟩
Mini⟩ primes
Cons (2, Cons (3, Nil))
Mini⟩ append (primes, Cons (5, Cons (7, Nil)))
Cons (2, Cons (3, Cons (5, Cons (7, Nil))))
Mini⟩ primes
Cons (2, Cons (3, Nil))
```

☞ *append* ist nicht destruktiv: *append* (*list1*, *list2*) konkateniert die Listen *list1* und *list2*; die Argumentlisten sind nach dem Aufruf unverändert, sie überdauern den Aufruf.

☞ Aus diesem Grund spricht man auch von einer *persistenten Datenstruktur* (lat. anhaltend, beharrlich).
☞ Persistente Datenstrukturen sind in der Regel ephemeren Datenstrukturen vorzuziehen: da sie effektfrei sind, sind sie leichter zu verstehen, zu verwenden und darüber hinaus auch flexibler.

35. Pascal und C: Listen

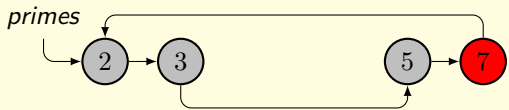
☞ Die Gefahr von Effekten illustriert die folgende Sitzung.

```
Mini⟩ primes
{contents = Cons (2, {contents = Cons (3, {contents =
    Cons (5, {contents = Cons (7, {contents = Nil})})})})}
Mini⟩ append (primes, primes)
()
Mini⟩ primes
...
```

☞ Wir hängen die Liste der ersten fünf Primzahlen an die Liste der ersten fünf Primzahlen und erhalten — ja, was eigentlich?

35. Pascal und C: Listen

Die Auswertung terminiert nicht! Der Aufruf *append* (*primes*, *primes*) hat eine *zyklische Liste* konstruiert.



☞ Das Struktur Entwurfsmuster garantiert nicht länger, dass die nach dem Muster gestrickten Funktionen auch terminieren!

☞ Aber es kommt noch schlimmer ...

35. Lösung Knobelaufgabe #18

☞ Selbst die Terminierung von nicht-rekursiven Funktionen ist nicht mehr gewährleistet!

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

☞ Die erste Wertebindung allokiert eine Speicherzelle des Typs $\mathit{Ref}\langle \mathit{Nat} \to \mathit{Nat} \rangle$. Die zweite Wertebindung weist dieser Speicherzelle eine Funktion zu, die die in dieser Speicherzelle enthaltene Funktion aufruft, also dank der Zuweisung sich selbst: ein zyklisches Speichergeflecht, das Rekursion simuliert.

☞ Ersetzen wir die Alternative durch $(\ !\mathit{fac})\ n$, dann terminiert *factorial* nicht.

35. Veränderliche

Neben Speicherzellen bietet F# noch ein weiteres Konstrukt an, um Programme mit einem Gedächtnis anzustatten: *Veränderliche* bzw. *Variablen*.

let mutable $funds = 0$

☞ Mit ***mutable*** wird ein Bezeichner als veränderlich gekennzeichnet: Aus einem Namen für einen Wert wird eine Speicherzelle.

Zuweisungen an ***mutable***s werden wie folgt notiert.

$funds \leftarrow funds + amount$

Lies: funds wird zu $funds + amount$. Wie in C oder Pascal wird ein veränderlicher Bezeichner automatisch dereferenziert. Der Bezeichner *funds* hat den Typ *Nat*.

35. Speicherzellen versus Veränderliche

	Allokation	Dereferenzierung	Zuweisung
Speicherzelle	***ref*** e	$!e$	$e_1 := e_2$
Veränderliche	***let mutable*** $x = e$	x	$x \leftarrow e$

☞ Die Vor- und Nachteile der automatischen Dereferenzierung haben wir bereits diskutiert; sie gelten in gleicher Weise für ***mutable***s.

35. Speicherzellen, da capo

Nicht nur Bezeichner können als veränderlich gekennzeichnet werden, auch Komponenten von Records.

Implementierung von Speicherzellen:

```
type Ref⟨'value⟩ =
  {mutable contents : 'value}
let ref value =
  {contents = value}
let (!) cell =
  cell.contents
let (:=) cell value =
  cell.contents ← value
```

☞ Der Unterschied zwischen einer Speicherzelle, einer Adresse, und ihrem Inhalt wird deutlich: *cell* ist die Speicherzelle und *cell.contents* ihr Inhalt.

The following anecdote is told of William James. [...] After a lecture on cosmology and the structure of the solar system, James was accosted by a little old lady.

"Your theory that the sun is the centre of the solar system, and the earth is a ball which rotates around it has a very convincing ring to it, Mr. James, but it's wrong. I've got a better theory," said the little old lady.

„And what is that, madam?" inquired James politely.

"That we live on a crust of earth which is on the back of a giant turtle."

Not wishing to demolish this absurd little theory by bringing to bear the masses of scientific evidence he had at his command, James decided to gently dissuade his opponent by making her see some of the inadequacies of her position.

„If your theory is correct, madam,"he asked, "what does this turtle stand on?"

"You're a very clever man, Mr. James, and that's a very good question," replied the little old lady, „but I have an answer to it. And it's this: The first turtle stands on the back of a second, far larger, turtle, who stands directly under him."

„But what does this second turtle stand on?" persisted James patiently.

To this, the little old lady crowed triumphantly, „It's no use, Mr. James — it's turtles all the way down."

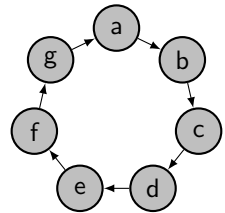
— J. R. Ross, Constraints on Variables in Syntax

35. Perlenketten

Besteht ein Record aus mehreren Komponenten, kann man selektiv für jede Komponente entscheiden, ob sie veränderlich sein soll.

type *Necklace* ⟨*'elem*⟩ =
  { *bead* : *'elem*
    mutable *next* : *Necklace* ⟨*'elem*⟩ }

type *Necklace* ⟨*'elem*⟩ =
  { ***mutable*** *bead* : *'elem*
    mutable *next* : *Necklace* ⟨*'elem*⟩ }



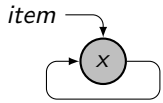
☞ Der Typ ist rekursiv definiert: Eine Kette besteht aus einer Perle und einer Kette, die aus einer Perle besteht und einer Kette, die ...

☞ Wo ist der Basisfall?

35. Konstruktion einer Perlenkette

Wie konstruieren wir ein zyklisches Geflecht? Durch eine rekursive *Werte*definition!

```
let single x =
  let rec item =
    { bead = x
      next = item }
  in item
```



☞ Rekursive Wertedefinitionen ergeben im Allgemeinen keinen Sinn: ***let rec*** $n = n + 1$.

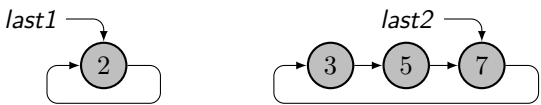
☞ Die Auswertung von *item* ist trickreich:

- Zunächst wird ***let*** $item = \{ bead = x; next = \bot \}$ angelegt, wobei $\bot$ ein beliebiger Dummywert ist;
- dann wird mittels $item.next \leftarrow item$ der Dummywert durch den zyklischen Verweis ersetzt.

Aus einer rekursiven Definition wird so ein zyklisches Geflecht.

35. Nicht-leere Sequenzen

Wir können Perlenketten verwenden, um nicht-leere, *endliche* Sequenzen von Elementen darzustellen.



☞ Wir verweisen jeweils auf das *letzte* Element der zu repräsentierenden Folge. (Warum?)

35. Nicht-leere Sequenzen: *head* und *tail*

Kopfelement und Restliste:

let *head* $(last : Necklace\,\langle 'elem\rangle) : 'elem =$
  $last.next.bead$

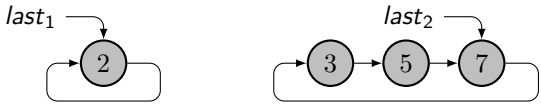
let *tail* $(last : Necklace\,\langle 'elem\rangle) : Necklace\,\langle 'elem\rangle =$
  $last.next \leftarrow last.next.next$
  $last$

☞ Perlenketten sind ephemeral: Die Funktion *tail* verändert ihr Argument.

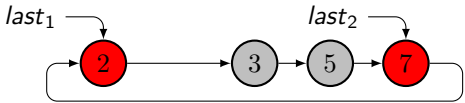
35. Nicht-leere Sequenzen: Konkatenation

Wir haben noch nicht verraten, warum wir ausgerechnet auf das letzte und nicht etwa auf das erste Element verweisen.

Diese Darstellung erlaubt es, zwei Folgen in *konstanter* Zeit zu konkatenieren.



Wir müssen lediglich $last_1.next$ und $last_2.next$ vertauschen.



let *swap* $(p : Necklace\ \langle'elem\rangle, q : Necklace\ \langle'elem\rangle) =$
  let $tmp = p.next$
  $p.next \leftarrow q.next$
  $q.next \leftarrow tmp$

35. Nicht-leere Sequenzen: Konkatenation

Das Ergebnis der Konkatenation ist die zweite Folge.

let *append* $(last_1 : Necklace\ \langle 'elem \rangle, last_2 : Necklace\ \langle 'elem \rangle) : Necklace\ \langle 'elem \rangle =$
  swap $(last_1, last_2)$
  $last_2$

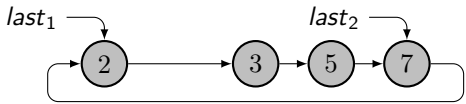
☞ Auch *append* verändert ihre Argumente: repräsentiert $last_i$ die Folge xs_i, dann

- repräsentiert $last_2$ *nach* dem Aufruf von *append* die Folge $xs_1 @ xs_2$,
- während $last_1$ die Folge $xs_2 @ xs_1$ darstellt.

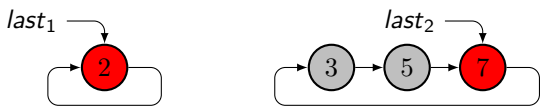
☞ Eine perfekte Symmetrie: *swap* implementiert so etwas wie die „symmetrische Konkatenation“ zweier Folgen.

35. Nicht-leere Sequenzen: Aufspaltung

Mit Hilfe von *swap* konkatenieren wir zwei Listen; mit Hilfe von *swap* können wir aber auch eine Liste aufspalten!



Wir müssen lediglich $last_1.next$ und $last_2.next$ vertauschen.



☞ Mit dem zweiten *swap* kehren wir zum alten Zustand zurück!

Verbesserte Definition von *tail*:

let $tail\ last = swap\ (last, last.next); last$

35. Nicht-leere Sequenzen: Demo

Mini⟩ ***let*** *xs* = *append* (*single* 1, *single* 2)
Mini⟩ ***let*** *ys* = *append* (*single* 815, *single* 4711)
Mini⟩ *swap* (*xs*, *ys*)
()
Mini⟩ *xs*
{ *bead* = 2; *next* = { *bead* = 815; *next* = { *bead* = 4711;
next = { *bead* = 1; *next* = . . . } } } }
Mini⟩ *ys*
{ *bead* = 4711; *next* = { *bead* = 1; *next* = { *bead* = 2;
next = { *bead* = 815; *next* = . . . } } } }
Mini⟩ *swap* (*xs*, *ys*)
()
Mini⟩ *xs*
{ *bead* = 2; *next* = { *bead* = 1; *next* = . . . } }
Mini⟩ *ys*
{ *bead* = 4711; *next* = { *bead* = 815; *next* = . . . } }

35. Anwendung: Ringpuffer

Aufgabe: schreibe einen Akzeptor für die Sprache

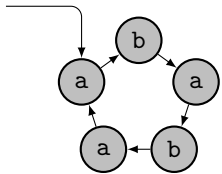
$$(\mathtt{a} \mid \mathtt{b})^* \; \mathtt{b} \; (\mathtt{a} \mid \mathtt{b})^n$$

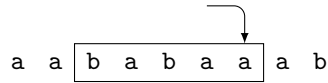
Wieviele Rechtsfaktoren hat die Sprache?

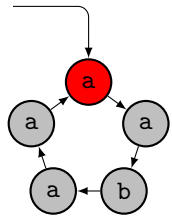
Idee: wir merken uns die letzten $n+1$ Zeichen in einem *Ringpuffer*. (Eigentlich müssten wir uns nur merken, *ob* das $(i+1)$-te Zeichen von hinten ein b oder ein anderes Zeichen ist.)

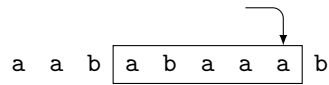
35. Anwendung: Ringpuffer — Beispiel

Mit Hilfe einer Perlenkette bewegen wir ein Sichtfenster über die Eingabefolge.









35. Anwendung: Ringpuffer — Programm

Allokation eines Ringpuffers der Größe $n + 1$:

```
let rec replicate (n : Nat) (a : 'a) : Necklace ⟨'a⟩ =
  if n = 0 then single a
           else append (replicate (n − 1) a, single a)
```

Der eigentliche Akzeptor:

```
let accept (n : Nat) : String → Bool =
  let mutable buffer = replicate n 'a'
  let rec loop = function
    | []      → buffer.next.bead = 'b'
    | x :: xs → buffer ← buffer.next
                buffer.bead ← x
                loop xs
  explode ≫ loop
```

35. Anwendung: Ringpuffer — Demo

```
Mini⟩ accept 3 "bb"
false
Mini⟩ accept 3 "bb"
false
Mini⟩ accept 3 "aa"
false
Mini⟩ accept 3 "aa"
false
```

```
Mini⟩ let accept3 = accept 3
Mini⟩ accept3 "bb"
false
Mini⟩ accept3 "bb"
true
Mini⟩ accept3 "aa"
true
Mini⟩ accept3 "aa"
false
```

Merkwürdig?

Der Aufruf *accept* 3 erzeugt jeweils einen *neuen* Ringpuffer.

In *accept3* ist *ein* Ringpuffer fest verdrahtet, der sich die vorherigen Eingaben merkt! So wird die Eingabe sozusagen kontinuierlich fortgeschrieben.

35. Persistente versus ephemerale Listen: Laufzeit

	persistent	ephemeral
head	konstant	konstant
tail	konstant	konstant
cons	konstant	konstant
append	linear zur Länge der ersten Liste	konstant

☞ Mit fortgeschrittenen Datenstrukturen lassen sich auch persistente Sequenzen implementieren, die die Konkatenation in konstanter Zeit unterstützen.