17. Knobelaufgabe #7

Harry Hacker hat die Fakultätsfunktion auf die ganzen Zahlen portiert.

```
let rec factorial (n : int) : int =    // natural numbers are
  if n = 0 then 1                      // for quiche eaters
           else factorial (n − 1) ∗ n
```

Die ersten Tests sind vielversprechend ...

```
Mini⟩ factorial 5
120
Mini⟩ factorial 10
3628800
```

...aber irgendwo muss sich ein Fehler eingeschlichen haben:

```
Mini⟩ factorial 17
− 288522240
Mini⟩ factorial 34
0
```


17. Leibniz Entwurfsmuster

- Beim Peano Entwurfsmuster wird das Problem für n auf das Problem für $n \dot{-} 1$ zurückgeführt.
- Wir können alternativ versuchen, das Problem für n auf das Problem für $n \div 2$ zurückzuführen.
- Zur Erinnerung:

$$a = (a \div 2) * 2 + (a \,\%\, 2)$$

☞ Jede Zahl a lässt sich eindeutig in einen Quotienten und in einen Rest zerlegen.

17. Potenzfunktion — da capo

Programmieren wir die Potenzfunktion neu.

```
let rec power (x : Nat, n : Nat) : Nat =
  if n = 0 then ...
           else ... power (x, n ÷ 2) ...
```


17. Potenzfunktion — da capo

- *Rekursionsbasis*: $x^0 = 1$.

```
let rec power (x : Nat, n : Nat) : Nat =
  if n = 0 then 1
           else ... power (x, n ÷ 2) ...
```

- *Rekursionsschritt*: $x^n = x^{(n \div 2) * 2 + (n \% 2)} = (x^{n \div 2})^2 * x^{n \% 2}$.

```
let rec power (x : Nat, n : Nat) : Nat =
  if n = 0 then 1
           else if n % 2 = 0 then square (power (x, n ÷ 2))
                             else square (power (x, n ÷ 2)) ∗ x
```

17. Potenzfunktion — Laufzeit

- Wieviele rekursive Aufrufe benötigt *power* (x, n) jetzt? (Vorher waren es n.)
- In jedem Schritt wird n halbiert; das können wir insgesamt $\lg n = \log_2 n$ mal machen.
- *power* hat eine *logarithmische* Laufzeit.
- Die erste Version hat eine *lineare* Laufzeit.
- Programme mit logarithmischer Laufzeit haben einen erheblichen Geschwindigkeitsvorteil gegenüber solchen mit linearer Laufzeit.

n		$\lg n$
1.000	$\approx$	10
1.000.000	$\approx$	20
1.000.000.000	$\approx$	30

17. Multiplikation — da capo

Mit dem gleichen Entwurfsmuster lässt sich auch die Multiplikation verbessern.

let rec *mul* $(m : Nat, n : Nat) : Nat =$
  if $m = 0$ ***then*** $\ldots$
    else $\ldots$ *mul* $(m \div 2, n) \ldots$

17. Multiplikation — da capo

- *Rekursionsbasis*: $0 * n = 0$.

let rec *mul* $(m : Nat, n : Nat) : Nat =$
  if $m = 0$ ***then*** 0
    else $\ldots$ *mul* $(m \div 2, n) \ldots$

- *Rekursionsschritt*: $m * n = ((m \div 2) * 2 + (m \,\%\, 2)) * n = 2\,((m \div 2) * n) + (m \,\%\, 2) * n$.

let rec *mul* $(m : Nat, n : Nat) : Nat =$
  if $m = 0$ ***then*** 0
    else if $m \,\%\, 2 = 0$ ***then*** $2 * mul\ (m \div 2, n)$
      else $2 * mul\ (m \div 2, n) + n$

Also, etwas merkwürdig ist das schon.

$$2 * mul\ (m \div 2, n)$$

Die Multiplikation soll doch implementiert werden, dafür kann ich doch die Multiplikation '$*$' selbst nicht verwenden.

Verdopplung ist ja ein ziemlich einfaches Produkt: *statt* $2 * x$ kannst Du ja $x + x$ rechnen.

Ok, dann schreibe ich das mal um ...

$$mul\ (m \div 2, n) + mul\ (m \div 2, n)$$

Ich glaube, das ist keine gute Idee. Du willst doch das Ergebnis verdoppeln, nicht die Rechnung.

$$\textbf{\textit{let}}\ p = mul\ (m \div 2, n)\ \textbf{\textit{in}}\ p + p$$

Um noch einmal auf Harrys Ausgangspunkt zurückzukommen. Der Punkt ist, dass die Operationen $e * 2$, $e \div 2$, $e \,\%\, 2$ sehr leicht in Hardware zu implementieren sind.

$2 * x$ ist einfacher als $x + x$?

Ja, so wie in unserem Dezimalsystem $10 * x$ sehr viel einfacher ist als $x + x + x + x + x + x + x + x + x + x$.

Ja, klar! Rechner verwenden das Dualsystem: $2 * x$ hängt hinten eine 0 dran, $x \div 2$ streicht das letzte Bit und $x \,\%\, 2$ ist das letzte Bit.

17. Lösung Knobelaufgabe #7

Der Effekt lässt sich einfacher vorführen, wenn man an Stelle des Typs *int* (vorzeichenbehaftete 32-Bit Zahl) den Typ *sbyte* (vorzeichenbehaftete 8-Bit Zahl) betrachtet.

```
Mini⟩ 1y * 2y * 3y * 4y * 5y
120y
Mini⟩ 1y * 2y * 3y * 4y * 5y * 6y
-48y
Mini⟩ 1y * 2y * 3y * 4y * 5y * 6y * 7y * 8y * 9y * 10y
0y
```

Warum $0y$? Die Multiplikation $2y * x$ „hängt hinten eine 0 dran“. Wenn man sich die Primfaktorenzerlegung des Produkts anschaut,

```
Mini⟩ 1y * 2y * 3y * (2y * 2y) * 5y * (2y * 3y) * 7y * (2y * 2y * 2y) *
      (3y * 3y) * (2y * 5y)
```

sieht man, dass insgesamt 8-mal mit $2y$ multipliziert wird: alle 8 Positionen einer 8-Bit Zahl sind somit 0.

17. Leibniz Entwurfsmuster

Haben wir die Aufgabe eine Funktion $f : \mathit{Nat} \to t$ zu erstellen, dann sieht ein zweiter Entwurf folgendermaßen aus.

```
let rec f (n : Nat) : t =                 Leibniz Entwurfsmuster:
  if n = 0 then ...                       Rekursionsbasis
       else ... f (n ÷ 2) ...             Rekursionsschritt
```

Die Ellipsen müssen mit Leben gefüllt werden:

- *Rekursionsbasis*: ein Ausdruck des Typs t.
- *Rekursionsschritt*: ein Ausdruck, der die Teillösung $f\ (n \div 2)$ vom Typ t zu einer Gesamtlösung vom Typ t erweitert.

17. Gottfried Wilhelm Leibniz (1646 – 1716)

Gottfried Wilhelm Leibniz war ein deutscher Philosoph und Wissenschaftler, Mathematiker, Diplomat, Physiker, Historiker, Bibliothekar und Doktor des weltlichen und des Kirchenrechts. Er gilt als der universale Geist seiner Zeit und war einer der bedeutendsten Philosophen des ausgehenden 17. und beginnenden 18. Jahrhunderts.

Auszug aus „Explication de l'Arithmétique Binaire“:

Cette expreſſion des Nombres étant établie, ſert à faire tres-facilement toutes ſortes d'operations.

Pour *l'Addition* par exemple.

110	6	101	5	1110	14
111	7	1011	11	10001	17
1101	13	10000	16	11111	31

17. Quadratwurzel — da capo

Versuchen wir das Leibniz Entwurfsmuster auf die Quadratwurzel anzuwenden.

```
let rec square-root (n : Nat) : Nat =
  if n = 0 then ...
           else ... square-root (n ÷ 2) ...
```


17. Quadratwurzel — da capo

- *Rekursionsbasis*: $\lfloor\sqrt{0}\rfloor = 0$.

```
let rec square-root (n : Nat) : Nat =
  if n = 0 then 0
           else ... square-root (n ÷ 2) ...
```

- *Rekursionsschritt*: Wie lässt sich aus $\lfloor\sqrt{n \div 2}\rfloor$ eine Lösung für $\lfloor\sqrt{n}\rfloor$ herleiten?

17. Quadratwurzel — da capo

☞ Es ist nicht zwingend durch zwei zu dividieren. Für unser Problem ist eine Quadratzahl, zum Beispiel vier, eine geschicktere Wahl.

```
let rec square-root (n : Nat) : Nat =
  if n = 0 then ...
           else ... square-root (n ÷ 4) ...
```


17. Quadratwurzel — da capo

- *Rekursionsbasis*: $\lfloor\sqrt{0}\rfloor = 0$.

```
let rec square-root (n : Nat) : Nat =
  if n = 0 then 0
           else ... square-root (n ÷ 4) ...
```

- *Rekursionsschritt*: Da $2\lfloor\sqrt{n/4}\rfloor$ und $\lfloor\sqrt{n}\rfloor$ höchstens um eins differieren, können wir die erste Version adaptieren.

```
let rec square-root (n : Nat) : Nat =
  if n = 0 then 0
           else let r = 2 * square-root (n ÷ 4)
                in if n < square (r + 1) then r else r + 1
```

17. Demo

```
Mini⟩ power (2, 10)
1024
Mini⟩ power (2, 16)
65536
Mini⟩ square-root it
256
Mini⟩ power (2, 8)
256
Mini⟩ square-root (factorial 10)
1904
Mini⟩ square it ⩽ factorial 10 && factorial 10 < square (it + 1)
true
```


17. Vertiefung: Ratespiel

Programmieren wir ein kleines Spiel:

Spieler A denkt sich eine Zahl aus, höchstens sechsstellig, die Spielerin B erraten muss. Spielerin B darf dazu Fragen der Form „Ist die gesuchte Zahl gleich oder kleiner als 815?“ stellen, die Spieler A wahrheitsgemäß beantworten muss.

Unsere Aufgabe ist es, die Logik von Spielerin B zu entwerfen und zu implementieren.

17. Schnittstelle

- Spieler A: muss zu einem gegebenen n Auskunft geben, ob die gesuchte Zahl gleich oder kleiner als n ist.

  $\mathit{player\text{-}A} : \mathit{Nat} \to \mathit{Bool}$

  ☞ Eine solche Funktion heißt auch *Orakel*.
- Spielerin B: ihr machen wir Spieler A bekannt.

  $\mathit{player\text{-}B} : (\mathit{Nat} \to \mathit{Bool}) \to \mathit{Nat}$

17. Spieler A

Beispielimplementierung von Spieler A.

let $\mathit{player\text{-}A}\ (\mathit{guess} : \mathit{Nat}) : \mathit{Bool} =$
$4711 \leqslant \mathit{guess}$

Die Repräsentation des Orakels stellt sicher, dass keine der beteiligten Parteien mogelt:

- die gesuchte Zahl ist fest verdrahtet — Spieler A kann sie nicht nachträglich ändern,
- die gesuchte Zahl kann aber auch nicht eingesehen werden — Spielerin B kann das Orakel nur auf eine Zahl anwenden und aus dem Ergebnis ihre Schlüsse ziehen.

17. Spielerin B

Logik von Spielerin B:

```
let player-B (oracle : Nat → Bool) : Nat =
  let rec search (n : Nat) : Nat =
    if oracle n then n
                else search (n + 1)
  in search 0
```


Das Programm ist aber nicht nach einem Entwurfsmuster gestrickt: das Problem für n wird auf das Problem für $n+1$ zurückgeführt.

Wieso, Hauptsache es klappt:

Mini⟩ *player-B player-A*
4711

Und was ist, wenn Spieler A sich gar keine Zahl ausgedacht hat?

Mini⟩ *player-B* (***fun*** $k \to$ *false*)

[wartet] Komisch, tut sich nichts. Wohl mal wieder ein schnellerer Prozzi fällig.

17. Terminierung

- Der Aufruf *player-B* (***fun*** $k \to$ *false*) *terminiert nicht*.
  - *search* 0 ruft *search* 1 auf.
  - *search* 1 ruft *search* 2 auf.
  - ...
- Die Semantik ordnet dem Aufruf keinen Wert zu.
  - Um einen Beweisbaum für *search* $0 \Downarrow \nu_0$ zu konstruieren, wird ein Beweisbaum für *search* $1 \Downarrow \nu_1$ benötigt.
  - Um einen Beweisbaum für *search* $1 \Downarrow \nu_1$ zu konstruieren, wird ein Beweisbaum für *search* $2 \Downarrow \nu_2$ benötigt.
  - ...

17. Terminierung

Das Peano Entwurfsmuster bändigt die Rekursion und stellt insbesondere die Terminierung sicher:

- das Problem für n wird auf das Problem für $n \dot{-} 1$ zurückgeführt;
- irgendwann wird auf diese Weise der Basisfall $n = 0$ erreicht.

17. Spielerin B

Laut Aufgabenstellung ist die Zahl höchstens sechsstellig.

Wir parametrisieren *player-B* mit der oberen, sowie der unteren Grenze des Suchintervalls.

```
let player-B (oracle : Nat → Bool,
              lower : Nat, upper : Nat) : Nat =
  let rec search (n : Nat) : Nat =
    if n = upper then upper
                 else if oracle n then n
                                  else search (n + 1)
  in search lower
```

☞ Ist die Terminierung sichergestellt?

17. Spielerin B

Beherzigen wir das Peano Entwurfsmuster und rekurrieren nicht über den Ratekandidaten selbst, sondern über den *Abstand* des Kandidaten zur oberen Schranke.

```
let player-B (oracle : Nat → Bool,
              lower : Nat, upper : Nat) : Nat =
  let rec search (d : Nat) : Nat =
    if d = 0 then upper
             else if oracle (upper ∸ d) then upper ∸ d
                                        else search (d ∸ 1)
  in search (upper ∸ lower)
```

☞ Voilà. Die Terminierung ist sichergestellt.

Die Terminierung der ersten Version steht aber noch aus.

Ich hab das mal nachgerechnet. Beide Versionen sind gleich! Pass auf: die Parameter der zwei Versionen von *search* stehen in der folgenden Beziehung:

$$n + d = upper$$

Beim ersten Aufruf von *search* wird die Beziehung hergestellt:

$$lower + (upper \dot{-} lower) = upper$$

Der rekursive Aufruf erhält die Beziehung:

$$(n + 1) + (d \dot{-} 1) = upper$$

Alle anderen korrespondierenden Ausdrücke ergeben genau die gleichen Werte. Zum Beispiel

$$n = upper \quad \text{und} \quad d = 0.$$

Ich bin beeindruckt! Aber irgendwo muss sich doch ein Fehler eingeschlichen haben.

Mini⟩ *player-B* (***fun*** $k \to$ *false*, $9, 0$)

[wartet] Terminiert nicht, das gibt's doch nicht!

Der Ansatz war schon goldrichtig! Auch ein gescheiterter Beweis kann wertvoll sein. Wo läuft es schief: Der Zusammenhang $a + (b \dot{-} a) = b$ gilt nur, wenn $b \geqslant a$.

Argh, '$\dot{-}$' ist ja die Subtraktion auf den natürlichen Zahlen!

Fazit: nie ohne guten Grund von den Entwurfsmustern abweichen! Programmierfehler sind oft sehr subtil und aus einem Programmierfehler wird heutzutage schnell ein „Sicherheitsloch".

17. Spielerin B — korrigierte Fassung

Korrigierte Fassung der ersten Version: '$\geqslant$' statt '$=$'.

```
let player-B (oracle : Nat → Bool,
              lower : Nat, upper : Nat) : Nat =
  let rec search (n : Nat) : Nat =
    if n ⩾ upper then upper
                 else if oracle n then n
                                  else search (n + 1)
  in search lower
```


17. Demo

```
Mini⟩ player-B (player-A, 0, 999999)
4711
Mini⟩ player-B (fun k → 815 ⩽ k, 0, 999)
815
Mini⟩ player-B (fun k → 815 < square (k + 1), 0, 999)
28
Mini⟩ square it ⩽ 815 && 815 < square (it + 1)
true
```

☞ Auf diese Weise lässt sich auch die Quadratwurzel einer Zahl bestimmen: die gedachte Zahl ist durch eine Formel gegeben.

17. Quadratwurzel — da capo

Eine neue Version von *square-root*:

```
let square-root (n : Nat) : Nat =
  linear-search (fun k → n < square (k + 1), 0, n)
```

☞ Da aus der Implementierung eines Spiels ein Programmstück von allgemeinem Nutzen geworden ist, haben wir *player-B* in *linear-search* umbenannt.

17. Spielerin B — da capo

Können wir die Laufzeit mit Hilfe des Leibniz Entwurfsmusters verbessern? Ja! *Idee:* das Suchintervall in jedem Schritt halbieren.

```
let binary-search (oracle : Nat → Bool,
                   lower : Nat, upper : Nat) : Nat =
  let rec search (l : Nat, u : Nat) : Nat =
    if l ⩾ u then u
    else let m = (l + u) ÷ 2
         in if oracle m then search (l,     m)
                        else search (m + 1, u)
  in search (lower, upper)
```

17. Demo

Es ist instruktiv, sich die Abfolge der Rateversuche anzuschauen. Zu diesem Zweck lassen wir Spieler A die geratenen Zahlen am Bildschirm ausgeben.

```
let player-A (guess : Nat) : bool =
  putline ("Geratene Zahl: " ^ show guess);
  4711 ⩽ guess
```


17. Demo

```
Mini⟩ binary-search (player-A, 0, 999999)
Geratene Zahl : 499999
Geratene Zahl : 249999
Geratene Zahl : 124999
Geratene Zahl : 62499
Geratene Zahl : 31249
Geratene Zahl : 15624
Geratene Zahl : 7812
Geratene Zahl : 3906
Geratene Zahl : 5859
Geratene Zahl : 4883
Geratene Zahl : 4395
Geratene Zahl : 4639
Geratene Zahl : 4761
Geratene Zahl : 4700
Geratene Zahl : 4731
Geratene Zahl : 4716
Geratene Zahl : 4708
Geratene Zahl : 4712
Geratene Zahl : 4710
Geratene Zahl : 4711
4711
```

☞ Die 4711 wird in 20 Runden systematisch eingekreist.

17. Quadratwurzel — da capo

Mit der verbesserten Suchstrategie lässt sich auch *square-root* auf eine logarithmische Laufzeit beschleunigen.

```
let square-root (n : Nat) : Nat =
  binary-search (fun k → n < square (k + 1), 0, n)
```


17. Zusammenfassung

Wir haben

- die grundlegenden Bestandteile von Mini–F# kennengelernt:
  - Boolesche Werte,
  - natürliche Zahlen,
  - Wertedefinitionen,
  - nicht-rekursive und rekursive Funktionen.
- alle Konzepte rigoros definiert:
  - abstrakte Syntax: Baumsprachen,
  - statische Semantik: Typregeln,
  - dynamische Semantik: Auswertungsregeln.
- Entwurfsmuster kennengelernt, die bei der systematischen Programmentwicklung hilfreich sind, und die die Terminierung sicherstellen,
- gesehen, dass das gleiche Problem sich oft auf verschiedene Art und Weisen und unterschiedlich effizient lösen lässt.