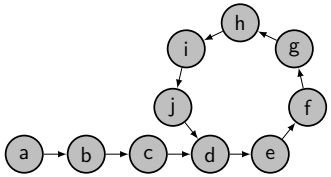
36. Knobelaufgabe #21

Nicht alle Elemente von *Necklace* $\langle 'a \rangle$ sind tatsächlich Perlenketten. Im Allgemeinen haben die Elemente die Form eines Loopings:

type *Necklace* $\langle 'a \rangle =$
$\{$ ***mutable*** *bead* $: 'a$
mutable *next* $:$ *Necklace* $\langle 'a \rangle \}$



Wie lässt sich der Anfang der „Loop“ bestimmen? (Im obigen Beispiel die Perle *d*, ausgehend von *a*.)

Benötigen Sie zusätzliche Sprachfeatures? Kommt Ihr Programm mit konstantem Speicherplatz aus?

36. Motivation

Zur Erinnerung: Arrays können auf zwei Weisen konstruiert werden:

- durch Aufzählung der Elemente: $[|2;3;5;7;11|]$;
- durch Angabe einer Bildungsvorschrift: $[|$***for*** i ***in*** $0\,..\,99 \to i * i|]$.

☞ Gilt in gleicher Weise für Listen.

☞ $[$***for*** i ***in*** $e_1 \to e_2]$ ist tatsächlich eine Abkürzung für $[$***for*** i ***in*** e_1 ***do yield*** $e_2]$.

☞ *Array-* bzw. *Listenbeschreibungen* sind sehr viel allgemeiner:

Mini⟩ $[$***yield*** $4711;$ ***for*** i ***in*** $0\,..\,5$ ***do yield*** $i * i]$
$[4711; 0; 1; 4; 9; 16; 25]$
Mini⟩ $[$***for*** i ***in*** $0\,..\,9$ ***do if*** $i \,\%\, 2 = 1$ ***then yield*** $i * i]$
$[1; 9; 25; 49; 81]$

☞ Mit einer einarmigen Alternative können Elemente gefiltert werden.

36. Motivation

☞ Generatoren dürfen geschachtelt werden.

Mini⟩ $[$***for*** i ***in*** $1\,..\,8$ ***do***
for j ***in*** $1\,..\,8$ ***do***
if $(i+j) \,\%\, 2 = 0$ ***then***
yield $(i,j)]$
$[(1,1);(1,3);(1,5);(1,7);(2,2);(2,4);(2,6);(2,8);(3,1);(3,3);(3,5);$
$(3,7);(4,2);(4,4);(4,6);(4,8);(5,1);(5,3);(5,5);(5,7);(6,2);(6,4);$
$(6,6);(6,8);(7,1);(7,3);(7,5);(7,7);(8,2);(8,4);(8,6);(8,8)]$

☞ Positionen aller schwarzen Felder auf einem 8×8-Schachbrett.

36. Abstrakte Syntax

Wir erweitern Ausdrücke um *Listenbeschreibungen* (analog für Arrays).

$e \in$ Expr ::=	*Ausdrücke:*
$\mid [\, se \,]$	Listenbeschreibung

Innerhalb der Klammern steht ein sogenannter *Sequenzausdruck*, eine neue syntaktische Kategorie.

$se \in$ SExpr ::=	*Sequenzausdrücke:*
$\mid$ ***yield*** e	Element
$\mid$ $se_1; se_2$	Konkatenation
$\mid$ ***if*** e ***then*** se	Filter
$\mid$ ***for*** x ***in*** e ***do*** se	Generator

☞ ***yield*** e ist *kein* Ausdruck.

☞ Statische Semantik, siehe Skript.

36. Dynamische Semantik

☞ Listenbeschreibungen sind „syntaktischer Zucker“: Sie versüßen das Leben beim Programmieren, sind aber nicht lebensnotwendig.

☞ Die Bedeutung der Konstrukte können wir erklären, indem wir sie entzuckern, in uns bekannte Ausdrücke *übersetzen*. Zum Beispiel:

$$[\textbf{for}\ i\ \textbf{in}\ 0\,..\,9\ \textbf{do yield}\ i*i] = \mathit{map}\ (\textbf{fun}\ i \to i*i)\ [0\,..\,9]$$
$$[\textbf{if}\ i\,\%\,2 = 1\ \textbf{then yield}\ i*i] = \textbf{if}\ i\,\%\,2 = 1\ \textbf{then}\ [i*i]\ \textbf{else}\ []$$

36. Dynamische Semantik

☞ Die Listenbeschreibung $[se]$ wird in den Ausdruck $[\![se]\!]$ übersetzt, wobei $[\![se]\!]$ wie folgt definiert ist.

$$\begin{aligned}
[\![\textbf{yield}\ e]\!] &= [e] \\
[\![se_1; se_2]\!] &= [\![se_1]\!] \mathbin{@} [\![se_2]\!] \\
[\![\ \textbf{if}\ e\ \textbf{then}\ se]\!] &= \textbf{if}\ e\ \textbf{then}\ [\![se]\!]\ \textbf{else}\ [] \\
[\![\textbf{for}\ x\ \textbf{in}\ e\ \textbf{do}\ se]\!] &= \mathit{collect}\ (\textbf{fun}\ x \to [\![se]\!])\ e
\end{aligned}$$

Die Funktion $\mathit{collect} : ('a \to 'b\ \mathit{list}) \to ('a\ \mathit{list} \to 'b\ \mathit{list})$ ist vordefiniert:

```
let rec collect f = function
  | []      -> []
  | x :: xs -> f x @ collect f xs
```

☞ *collect f list* wendet *f* auf jedes Element von *list* an und konkateniert die resultierenden Listen.

36. Dynamische Semantik: Beispiel

$$\begin{aligned}
& [\textbf{for}\ i\ \textbf{in}\ [0\,..\,9]\ \textbf{do if}\ i\,\%\,2 = 1\ \textbf{then yield}\ i*i] \\
= \quad & \{\ \text{Übersetzung Listenbeschreibung}\ \} \\
& [\![\textbf{for}\ i\ \textbf{in}\ [0\,..\,9]\ \textbf{do if}\ i\,\%\,2 = 1\ \textbf{then yield}\ i*i]\!] \\
= \quad & \{\ \text{Übersetzung Generator}\ \} \\
& \mathit{collect}\ (\textbf{fun}\ i \to [\![\ \textbf{if}\ i\,\%\,2 = 1\ \textbf{then yield}\ i*i]\!])\ [0\,..\,9] \\
= \quad & \{\ \text{Übersetzung Filter}\ \} \\
& \mathit{collect}\ (\textbf{fun}\ i \to \textbf{if}\ i\,\%\,2 = 1\ \textbf{then}\ [\![\textbf{yield}\ i*i]\!]\ \textbf{else}\ [])\ [0\,..\,9] \\
= \quad & \{\ \text{Übersetzung}\ \textbf{yield}\ \} \\
& \mathit{collect}\ (\textbf{fun}\ i \to \textbf{if}\ i\,\%\,2 = 1\ \textbf{then}\ [i*i]\ \textbf{else}\ [])\ [0\,..\,9] \\
= \quad & \{\ \text{Definition von}\ [l\,..\,u]\ \} \\
& \mathit{collect}\ (\textbf{fun}\ i \to \textbf{if}\ i\,\%\,2 = 1\ \textbf{then}\ [i*i]\ \textbf{else}\ [])\ [0;1;2;3;4;5;6;7;8;9] \\
= \quad & \{\ \text{Definition von}\ \mathit{collect}\ \} \\
& [] \mathbin{@} [1] \mathbin{@} [] \mathbin{@} [9] \mathbin{@} [] \mathbin{@} [25] \mathbin{@} [] \mathbin{@} [49] \mathbin{@} [] \mathbin{@} [81] \\
= \quad & \{\ \text{Definition von ‘@’}\ \} \\
& [1;9;25;49;81]
\end{aligned}$$

36. Vertiefung

☞ Mit Listenbeschreibungen lassen sich bequem „Datenbankanfragen“ formulieren. *Beispiel:* eine Prüfungsdatenbank.

```
type Name   = String // student name
type GUID   = Int    // student id (Globally Unique Identifier)

type Course = | ALG | FPR | IPR | OOP
type Mark   = | Insufficient | Sufficient
              | Good | VeryGood | Excellent

type Student = {name : Name; guid : GUID}
type Result  = {course : Course; guid : GUID; mark : Mark}
```

```
let students : Student list =
  [{name = "Ralf"; guid = 4711};
   {name = "Lisa"; guid = 815 };...]
let results : Result list =
  [{course = ALG; guid = 4711; mark = Good      };
   {course = FPR; guid = 815;  mark = Excellent };
   {course = FPR; guid = 4711; mark = Sufficient};...]
```

36. Vertiefung: Datenbanken

Wie heißt der Student mit der Matrikelnummer 4711?

```
let query1 : Name list =
  [for student in students do
     if student.guid = 4711 then
       yield student.name]
```

Liste die Ergebnisse der Vorlesung *ALG*.

```
let query2 : Mark list =
  [for result in results do
     if result.course = ALG then
       yield result.mark]
```


36. Vertiefung: Datenbanken

Welche Ergebnisse hat der Student namens `"Ralf"` erzielt?

```
let query4 : Mark list =
  [for student in students do
     if student.name = "Ralf" then
       for result in results do
         if student.guid = result.guid then
           yield result.mark]
```

Welche Student*in hat mindestens ein exzellentes Ergebnis in einem ihrer Kurse?

```
let query5 : Name list =
  [for result in results do
     if result.mark = Excellent then
       for student in students do
         if result.guid = student.guid then
           yield student.name]
```


37. Motivation

☞ Mit der Einführung von Operationen zur Ein- und Ausgabe und zur Speichermanipulation wird es notwendig, Effekte zu kontrollieren:

- Welche Effekte treten auf,
- in welcher Reihenfolge treten sie auf und
- wie oft werden sie gegebenenfalls wiederholt?

Mit dem Einzug von Effekten verändert sich nicht nur die Natur des Rechnens, auch der Programmierstil wird potentiell ein anderer.

- Der *problemnahe*, *deskriptive* Charakter („Was?“) weicht
- einem *maschinennahen*, *präskriptiven* Stil („Wie?“).

Das „Wie“, der Kontrollfluss, kann mit speziellen Sprachmitteln, den Kontrollstrukturen, präzisiert werden.

37. Motivation

☞ Eine ***for***-Schleife kann verwendet werden, um effektvolle Ausdrücke aneinanderzureihen:

```
Mini⟩ print 4711; for i in 0 .. 5 do print (i * i)
4711
0
1
4
9
16
25
Mini⟩ for i in 0 .. 9 do if i % 2 = 1 then print (i * i)
1
9
25
49
81
```

☞ Kommt uns die Syntax bekannt vor?

37. Motivation

Die Syntax erinnert stark an Sequenzausdrücke; lediglich ***yield*** ist durch *print* ersetzt worden.

```
Mini⟩  for i in 1 .. 8 do
         for j in 1 .. 8 do
           if (i + j) % 2 = 0 then
             print (i, j)
(1, 1)
(1, 3)
...
(8, 6)
(8, 8)
```


37. Motivation

Es ist nicht zwingend, die Elemente auszugeben. Das folgende Beispiel illustriert einen anderen Effekt.

```
let sum (xs : List ⟨Int⟩) : Int =
  let mutable acc = 0
  for x in xs do
    acc ← acc + x
  acc
```

☞ Die Funktion *sum* summiert die Elemente der angegebenen Liste.

☞ Eine ***for***-Schleife wiederholt einen Effekt für *alle* Elemente einer Liste. Das ist nicht für alle Aufgabenstellungen adäquat ...

37. Motivation

Mit Hilfe einer ***while***-Schleife kann man die lineare Suche *iterativ* implementieren.

```
let linear-search (key : 'elem) (a : Array ⟨'elem⟩) : Option ⟨Int⟩ =
  let mutable i = 0
  while i < a.Length && a.[i] < key do
    i ← i + 1
  if i < a.Length && a.[i] = key then Some i
                                 else None
```

☞ Der Rumpf der ***while***-Schleife wird wiederholt, solange die Schleifenbedingung wahr ist.

☞ ***while***-Schleifen sind *nur* im Zusammenspiel mit Effekten sinnvoll. Die Schleifenbedingung wird wiederholt ausgerechnet; damit sich der Wahrheitswert ändert, *muss* der Ausdruck von Benutzereingaben oder vom Speicher abhängen.

37. Motivation

Auch die binäre Suche lässt sich iterativ formulieren:

```
let ternary-search (key : 'elem) (a : Array ⟨'elem⟩) : Option ⟨Int⟩ =
  let mutable l = 0
  let mutable u = a.Length − 1
  let mutable found = None
  while l ⩽ u && found = None do
    let m = (l + u) / 2
    if    key < a.[m]   then u ← m − 1
    elif  key = a.[m]   then found ← Some m
    (* key > a.[m] *) else  l ← m + 1
  found
```

☞ Wir definieren drei Veränderliche: die Intervallgrenzen *l* und *u* und den Suchstatus *found*. Die Schleifenbedingung *l* ⩽ *u* && *found* = *None* involviert alle drei Veränderliche.

37. Abstrakte Syntax

Ausdrücke vom Typ *Unit*, die *nur* ihres Effektes wegen ausgerechnet werden, heißen auch *Anweisungen*.

$e ::= \cdots$	*Kontrollstrukturen:*
$\mid e_1 ; e_2$	Sequenz
$\mid$ ***if*** e_1 ***then*** e_2	einarmige Alternative
$\mid$ ***if*** e_1 ***then*** e_2 ***else*** e_3	zweiarmige Alternative
$\mid$ ***for*** x ***in*** e_1 ***do*** e_2	beschränkte Wiederholung
$\mid$ ***while*** e_1 ***do*** e_2	unbeschränkte Wiederholung

37. Statische Semantik

Die ***for***-Schleife ist ein „Binder“: Die Schleifenvariable ist im Rumpf der Schleife sichtbar.

$$\frac{\Sigma \vdash e_1 : \mathit{List}\,\langle t_1 \rangle \qquad \Sigma, \{x_1 \mapsto t_1\} \vdash e : \mathit{Unit}}{\Sigma \vdash \textbf{\textit{for}}\ x_1\ \textbf{\textit{in}}\ e_1\ \textbf{\textit{do}}\ e : \mathit{Unit}}$$

$$\frac{\Sigma \vdash e_1 : \mathit{Bool} \qquad \Sigma \vdash e_2 : \mathit{Unit}}{\Sigma \vdash \textbf{\textit{while}}\ e_1\ \textbf{\textit{do}}\ e_2 : \mathit{Unit}}$$

☞ Der Schleifenrumpf ist jeweils eine Anweisung.

37. Dynamische Semantik

Schleifen sind wie Listenbeschreibungen syntaktischer Zucker.

☞ Die Schleife ***for*** x ***in*** e_1 ***do*** e_2 entspricht dem Aufruf *foreach* e_1 (***fun*** $x \to e_2$), wobei *foreach* wie folgt definiert ist.

```
let rec foreach (list : List ⟨'a⟩) (body : 'a → Unit) : Unit =
  match list with
    | []      → ()
    | x :: xs → body x; foreach xs body
```

☞ *foreach* ist *endrekursiv*: der rekursive Aufruf ist die „letzte Aktion“ im Funktionsrumpf.

37. Dynamische Semantik

☞ Die Schleife ***while*** e_1 ***do*** e_2 wird in den Ausdruck *whiledo* (***fun*** () $\to e_1$) (***fun*** () $\to e_2$) übersetzt.

```
let rec whiledo (test : Unit → Bool) (body : Unit → Unit) : Unit =
  if test () then
    body ()
    whiledo test body
```

☞ Warum hat *test* den Typ *Unit* → *Bool*?

☞ Auch *whiledo* ist *endrekursiv*.

Verhältnis Rekursion und Iteration:

- eine Schleife kann in eine endrekursive Funktion übersetzt werden; und umgekehrt
- kann eine endrekursive Funktion in eine Schleife übersetzt werden.

☞ Mehr dazu im Skript.

37. Vertiefung

Aufgabe: Sortieren eines Arrays an „Ort und Stelle“ (engl. in place, lat. in situ), das heißt, *ohne* zusätzlichen Speicher zu allokieren.

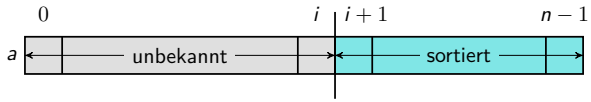
```
Mini⟩ let revenues = [| 815; 4; 7; 1; 1 |]
Mini⟩ selection-sort revenues
()
Mini⟩ revenues
[| 1; 1; 4; 7; 815 |]
```

Die Sortierfunktion hat kein interessantes Ergebnis, aber einen bemerkenswerten Effekt: Nach dem Aufruf ist das Eingabearray sortiert.

☞ Was wir bisher verschwiegen haben: Auch Arrays sind veränderlich; die Elemente eines Arrays können mittels einer Zuweisung der Form $a.[e_1] \leftarrow e_2$ verändert werden.

37. Vertiefung: Sortieren durch Einfügen

Idee: Wir teilen das zu sortierende Array gedanklich in zwei Zonen auf.



Über die linke, graue Zone wissen wir nichts; die rechte, grüne Zone ist sortiert. Wir lassen sie schrittweise wachsen, indem wir jeweils ein graues Element in die grüne Zone einfügen.

37. Vertiefung: Sortieren durch Einfügen

```
let insertion-sort (a : Array ⟨'elem⟩) =
  let n = a.Length
  for i in n − 2 .. −1 .. 0 do
    let key = a.[i]
    // insert key into the sorted sub-array a.[i + 1], . . . , a.[n − 1]
    let mutable j = i + 1
    while j < n && key > a.[j] do
      a.[j − 1] ← a.[j]
      j ← j + 1
    a.[j − 1] ← key
```

☞ $l \,..\, d \,..\, r$ generiert die Elemente l, $l + d$, $l + 2 \cdot d$, …, r.

37. Vertiefung: Sortieren durch Einfügen

Schleifeninvariante: $a.[i + 1] \leqslant a.[i + 2] \leqslant \cdots \leqslant a.[n - 1]$

- die Initialisierung der Laufvariable etabliert die Invariante:

  $i = n - 2$

- der Schleifenrumpf erhält die Invariante:

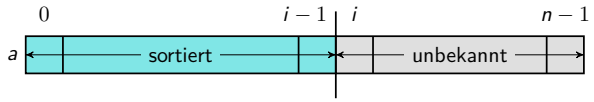
```
let key = a.[i]
let mutable j = i + 1
while j < n && key > a.[j] do
  a.[j − 1] ← a.[j]
  j ← j + 1
a.[j − 1] ← key
```

- nach dem letzten Schleifendurchlauf impliziert die Invariante die Nachbedingung:

  $i = -1$

37. Vertiefung: Sortieren durch Auswählen

Idee: Wir teilen das zu sortierende Array gedanklich in zwei Zonen auf.



Die linke, grüne Zone ist sortiert; über die rechte, graue Zone wissen wir nichts. Die grüne Zone wird schrittweise nach rechts ausgedehnt, indem jeweils das Minimum der grauen Zone mit dem ersten Element der grauen Zone ausgetauscht wird.

Schleifeninvariante: Die grüne Zone ist sortiert *und* die Elemente der grünen Zone sind höchstens so groß wie die Elemente der grauen Zone.

37. Vertiefung: Sortieren durch Auswählen

```
let selection-sort (a : Array ⟨elem⟩) =
  let n = a.Length
  for i in 0 .. n − 2 do
    // determine minimum of a.[i], . . . , a.[n − 1]
    let mutable m = i
    for j in i + 1 .. n − 1 do
      if a.[j] < a.[m] then
        m ← j;
    // swap a.[i] and a.[m]
    let tmp = a.[i]
    a.[i]   ← a.[m]
    a.[m] ← tmp
```


37. Vertiefung: Sortieren durch Zählen

Zur Erinnerung: Jedes Sortierverfahren, das auf dem Vergleichen von Elementen basiert, benötigt im schlechtesten Fall mindestens $n \lg n$ Vergleiche.

☞ Es nicht immer zwingend zu vergleichen; um ein Array zu sortieren, das nur Zahlen aus einem gegebenen Intervall, $0 \,..\, m$, enthält, bietet es sich an, zu zählen, wie häufig jedes Element aus dem Intervall vorkommt. ☞ Auch Arrays sind ***mutable***:

```
let counting-sort (m : Int, array : Array ⟨Int⟩) : Array ⟨Int⟩ =
  let counts = [| for i in 0 .. m → 0 |]
  for j in array do
    counts.[j] ← counts.[j] + 1
  [| for i in 0 .. m do
       for c in 1 .. counts.[i] do
         yield i |]
```

☞ Die Laufzeit ist proportional zu $\max\{m, n\}$, wobei n die Größe des Eingabearrays ist.

38. Motivation

Nicht jede Rechnung lässt sich einem erfolgreichen Ende zuführen:

$$815 \,/\, (4711 - 7 * 673) + 1$$

☞ Die Auswertungsregel für '/' ist nicht anwendbar.

- Die Teilrechnung $815 \,/\, (4711 - 7 * 673)$ kann nicht weitergeführt werden.
- Daraus folgt aber nicht, dass wir auch die Gesamtrechnung abbrechen sollten.
- *Im Gegenteil:* resultiert der Divisor zum Beispiel aus einer interaktiven Eingabe, dann sollte das Programm die Benutzer*in auf den Fehler hinweisen.
- *Wunsch:* ein allgemeiner Mechanismus, um Ausnahmesituationen zu signalisieren, Rechnungen abzubrechen und an anderer Stelle wieder aufzunehmen.

38. Motivation: Werfen einer Ausnahme

Idee: Eine Rechnung wird abgebrochen, indem eine sogenannte *Ausnahme* (engl. exception) ausgelöst wird. Die ausgelöste Ausnahme kann an einer anderen Stelle behandelt werden; an dieser anderen Stelle wird die Rechnung fortgesetzt.

Bildlicher: eine Ausnahme wird *geworfen* und *gefangen*.

Die Division wirft die Ausnahme *Div*, wenn der Divisor Null ist.

Mini⟩ $815 \;/\; (4711 - 7 * 673) + 1$
uncaught exception : *Div*

☞ Da die Ausnahme nicht gefangen wird, trifft der Wurf auf die Benutzungsoberfläche und führt zu einer Fehlermeldung.

38. Motivation: Werfen einer Ausnahme

☞ Eine Ausnahme kann auch explizit mit ***raise*** geworfen werden.

Mini⟩ ***raise*** *Div* $+ 1$
uncaught exception : *Div*

☞ Dieser Ausdruck hat den gleichen Effekt wie $815 \;/\; (4711 - 7 * 673) + 1$.

☞ Da die Rechnung abgebrochen wird, besitzt der Ausdruck ***raise*** *Div* $+ 1$ *keinen* Wert, sondern hat nur einen Effekt.

38. Motivation: Fangen einer Ausnahme

Eine geworfene Ausnahme kann mit dem ***try***-Konstrukt gefangen werden.

Mini⟩ ***try*** *show* $(815 \;/\; (4711 - 6 * 773) + 1)$ ***with*** | *Div* $\to$ "?"
"12"
Mini⟩ ***try*** *show* $(815 \;/\; (4711 - 7 * 673) + 1)$ ***with*** | *Div* $\to$ "?"
"?"

☞ Zwischen den Schlüsselwörtern ***try*** und ***with*** steht der auszurechnende Ausdruck.

☞ Gelingt dessen Auswertung, so ist der berechnete Wert auch der Wert des gesamten Ausdrucks.

☞ Wird hingegen während der Auswertung eine Ausnahme geworfen, dann kommt der Teil nach dem Schlüsselwörtern ***with*** zum Einsatz.

38. Motivation: Fangen einer Ausnahme

☞ Passt die Ausnahme auf die linke Seite einer Regel, so wird mit der Auswertung der rechten Seite fortgefahren.

Mini⟩ ***try*** *show* $(815 \;/\; (4711 - 7 * 673) + 1)$ ***with*** | *Div* $\to$ "?"
"?"

☞ Man sagt auch, die Ausnahme wird behandelt.

☞ Passt kein Muster, so wird die Ausnahme weitergeworfen.

Mini⟩ ***try*** *show* $(815 \;/\; (4711 - 7 * 673) + 1)$ ***with*** | *Match* $\to$ "?"
uncaught exception : *Div*

38. Motivation

☞ Ausnahmen sind normale Werte, Elemente des Typs *Exception*, und können wie Elemente eines Variantentyps verwendet werden.

```
Mini⟩ let exns = [|Div; Match|]
val exns : Array ⟨Exception⟩
Mini⟩ match exns.[1] with | Div → "div" | Match → "match"
"div"
```

Zunächst wird ein Array von Ausnahmen konstruiert; anschließend wird das zweite Arrayelement mit einer Fallunterscheidung analysiert.

38. Motivation

Die Ausnahme *Match* signalisiert, dass eine Fallunterscheidung fehlgeschlagen ist, dass keines der angegebenen Muster auf den analysierten Wert gepasst hat.

```
let head (list : List ⟨'a⟩) : 'a =
  match list with
    | x :: _ → x
```

Die Funktion *head* bestimmt das erste Element einer Liste; der Aufruf *head e* wirft die Ausnahme *Match*, wenn *e* zur leeren Liste auswertet. ☞ Mathematisch gesehen ist *head* eine *partielle Funktion*.

```
Mini⟩ head (head ([] :: [4711] :: []))
uncaught exception : Match
Mini⟩ try head (head ([] :: [4711] :: [])) with | Match → 0
0
```

(☞ Dichtung und Wahrheit: In F# wird tatsächlich eine *Microsoft.FSharp.Core.MatchFailureException* geworfen, siehe Skript.)

38. Motivation: Definition von Ausnahmen

☞ Man kann auch eigene Ausnahmen definieren, um Fehlerfälle genauer beschreiben zu können.

```
exception Head

let head (list : List ⟨'a⟩) : 'a =
  match list with
    | []     → raise Head
    | x :: _ → x
```

☞ **exception** führt eine neue Ausnahme ein; der Datentyp *Exception* wird so um ein Element erweitert.

☞ Die neue Ausnahme wird geworfen, wenn *head* mit der leeren Liste aufgerufen wird; der Programmtext dokumentiert auf diese Weise, dass *head* diesen Fall nicht behandeln kann (oder will).

38. Motivation

☞ Ausnahmen können auch ein Argument haben, um Informationen vom Ort des Abbruchs zum Ort der Wiederaufnahme zu schicken.

```
exception Insufficient of Nat

module Account =
  let private funds = ref 0

  let deposit (amount : Nat) =
        funds := !funds + amount

  let withdraw (amount : Nat) =
    if !funds ⩾ amount then
        funds := !funds ∸ amount
    else
        raise (Insufficient (!funds))

  let balance () = !funds
```

☞ Die Funktion *withdraw* bucht den gewünschten Betrag nur ab, wenn das Konto gedeckt ist. Die Ausnahme gibt den maximal abhebbaren Betrag an.

38. Motivation

Das ursprüngliche Verhalten von *withdraw* — soviel abheben wie möglich — lässt sich nachprogrammieren, indem man die geworfene Ausnahme fängt und dann das Konto leerräumt.

```
let maximal-withdraw (amount : Nat) : Nat =
  try
    Account.withdraw amount; amount
  with
  | Insufficient rest → Account.withdraw rest; rest
```


38. Abstrakte Syntax

Wir erweitern Definitionen um Ausnahmedefinitionen.

$d ::= \cdots$	*Deklarationen:*
$\mid$ ***exception*** C ***of*** t	Definition einer Ausnahme

☞ Wird bei einer ***exception*** Definition kein Typargument angegeben, dann fassen wir das, wie bei „normalen“ Konstruktoren, als Abkürzung für C ***of*** *Unit* auf.

Wir erweitern Ausdrücke um Konstrukte zum Werfen und Fangen von Ausnahmen.

$e ::= \cdots$	*Ausdrücke:*
$\mid$ ***raise*** e	Werfen einer Ausnahme
$\mid$ ***try*** e ***with*** m	Fangen einer Ausnahme

☞ Syntaktisch entspricht ***try*** e ***with*** m einer erweiterten Fallunterscheidung: e ist ein Ausdruck und m eine Folge von Regeln der Form $p \to e$.

38. Statische Semantik

Ausnahmen haben den Typ *Exception*.

$t ::= \cdots$	*Typen:*
$\mid$ *Exception*	Typ der Ausnahmen

☞ Der Typ entspricht im wesentlichen einem Variantentyp mit dem Unterschied, dass die Konstruktoren nicht sofort, sondern peu à peu mit Hilfe von ***exception*** Definitionen eingeführt werden.

Typregeln:

exception C ***of*** t

☞ Eine Ausnahmedefinition wird ähnlich wie eine Variantentypdefinition gehandhabt: der Ausnahmekonstruktor korrespondiert zu einer Funktion vom Typ $t \to$ *Exception*. Wir erlauben es *nicht*, Ausnahmekonstruktoren zu redefinieren oder lokal zu definieren.

38. Statische Semantik

Typregeln:

$$\frac{\Sigma \vdash e : \mathit{Exception}}{\Sigma \vdash \textbf{\textit{raise}}\ e : t}$$

$$\frac{\Sigma \vdash e : t \qquad \Sigma \vdash m\ (\mathit{Exception}) : t}{\Sigma \vdash \textbf{\textit{try}}\ e\ \textbf{\textit{with}}\ m : t}$$

☞ Die Regel m muss, angewendet auf eine Ausnahme, zu einem Element des Typs t auswerten, siehe Skript.

38. Statische Semantik

☞ ***raise*** *e* hat einen beliebigen Typ!

Somit kann ***raise*** *e* überall verwendet werden:

- als Boolescher Wert: ***raise*** *Div* && *e*,
- als natürliche Zahl: ***raise*** *Div* $+ 4711$,
- als Funktion: (***raise*** *Div*) 4711,
- als Paar: *snd* (***raise*** *Div*),
- usw.

☞ ***raise*** *e* bricht die aktuelle Rechnung ab, deswegen kann der Ausdruck in jedem beliebigen Kontext stehen; der Kontext bekommt den Wert von ***raise*** *e* ja niemals zu Gesicht.