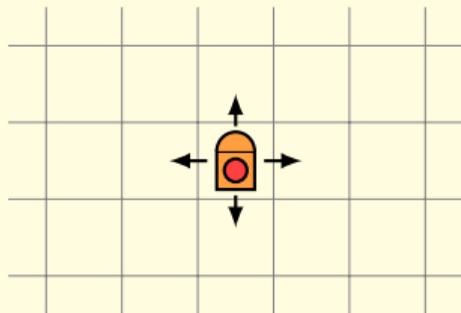


38. Knobelaufgabe #22

Ein kleiner Roboter wird in einem 2-dimensionalen Gitter ausgesetzt, das sich in alle vier Richtungen unendlich ausdehnt. Der Roboter wird in einem Schritt in eine der vier Himmelsrichtungen bewegt; er meldet jeweils zurück, ob er die Zielposition bereits vorher besucht hat.



Zum Beispiel: auf die Eingaben NEESWNNWSS antwortet der Roboter mit NNNNNYNNYY.

type *Direction* = | N | E | S | W

robot : *Direction* → *Bool*

Programmieren Sie den Roboter so, dass er zu jeder Eingabe die Ausgabe in *konstanter* Zeit ermittelt.

38. Dynamische Semantik

- ▶ Die Auswertung verändert sich mit dem Einzug von Effekten.
- ▶ Wie müssen wir die Auswertungsregeln modifizieren, um Ausnahmen modellieren zu können?
- ▶ Eine Auswertung hat jetzt zwei mögliche Ergebnisse:
 - ▶ einen „gewöhnlichen“ Wert wie 4711 oder
 - ▶ einen „außergewöhnlichen“ Wert wie *Div*.
- ▶ *Idee*: Wir modifizieren die Auswertungsrelation

$$\delta \vdash e \Downarrow r$$

wobei r ein *Resultat* ist, ein Wert oder eine Ausnahme.

38. Dynamische Semantik

Die geworfene Ausnahme nennt man auch *Paket*.

$r \in \text{Result}$

Resultate

$r ::= \nu$

Wert

| $\boxed{\nu}$

Paket

☞ Das Kästchen $\boxed{\nu}$ soll verdeutlichen, dass es sich um ein Paket handelt.

☞ Der Inhalt des Pakets, zum Beispiel *Div* oder *Insufficient* 4711, ist ein normaler Wert — statisch gesehen ein Element des Typs *Exception*.

38. Dynamische Semantik

Auswertungsregeln:

$$\frac{\delta \vdash e \Downarrow \nu}{\delta \vdash \mathbf{raise} \ e \Downarrow \boxed{\nu}}$$

$$\frac{\delta \vdash e \Downarrow \boxed{\nu}}{\delta \vdash \mathbf{raise} \ e \Downarrow \boxed{\nu}}$$

☞ Wenn bei der Auswertung von e bereits ein Paket geworfen wird (!), so wirft **raise** dieses Paket weiter.

☞ Werte werden niemals doppelt verpackt: ein Ergebnis der Form $\boxed{\nu}$ gibt es *nicht*.

38. Dynamische Semantik

Auswertungsregeln:

$$\frac{\delta \vdash e \Downarrow \nu}{\delta \vdash \mathbf{try} \ e \ \mathbf{with} \ m \ \Downarrow \ \nu}$$

$$\frac{\delta \vdash e \Downarrow \boxed{\nu} \quad \delta \vdash m(\nu) \Downarrow r}{\delta \vdash \mathbf{try} \ e \ \mathbf{with} \ m \ \Downarrow r}$$

$$\frac{\delta \vdash e \Downarrow \boxed{\nu} \quad \delta \vdash m(\nu) \Downarrow \downarrow}{\delta \vdash \mathbf{try} \ e \ \mathbf{with} \ m \ \Downarrow \boxed{\nu}}$$

☞ Zunächst wird e ausgewertet („die Auswertung wird versucht“).

☞ Resultiert die Auswertung in einem Wert, so ist dieser auch der Wert des gesamten Ausdrucks („der Versuch ist erfolgreich“).

☞ Wird das Paket $\boxed{\nu}$ geworfen („die Auswertung misslingt“), so fängt **with** das Paket auf und packt es aus ($\delta \vdash m(\nu) \Downarrow r$ beschreibt das „pattern matching“, siehe Skript).

38. Dynamische Semantik

☞ Insgesamt gibt es vier unterschiedliche Konstellationen:

try 4711 **with** | *Div* → 815 ↓ 4711

☞ Die „versuchte“ Auswertung ist erfolgreich.

try 4711 / 0 **with** | *Div* → 815 ↓ 815

☞ Die Auswertung misslingt; die Ausnahme wird gefangen und behandelt; die Ausnahmebehandlung resultiert in einem Wert.

try 4711 / 0 **with** | *Div* → **raise Match** ↓ Match

☞ Die Auswertung misslingt; die Ausnahme wird gefangen und behandelt; die Ausnahmebehandlung resultiert in einem Paket.

try 4711 / 0 **with** | *Match* → 815 ↓ Div

☞ Die Auswertung misslingt; die Ausnahme wird gefangen, aber nicht behandelt; die Ausnahme wird weitergeworfen.

38. Dynamische Semantik

☞ Mit Hilfe von Ausnahmen können wir die fehlenden Auswertungsregeln für die Division usw. nachtragen.

$$\frac{\delta \vdash e_2 \Downarrow 0}{\delta \vdash e_1 / e_2 \Downarrow \boxed{\text{Div}}} \qquad \frac{\delta \vdash e_2 \Downarrow 0}{\delta \vdash e_1 \% e_2 \Downarrow \boxed{\text{Mod}}}$$

$$\frac{\delta \vdash e \Downarrow \nu \qquad \delta \vdash m(\nu) \Downarrow \zeta}{\delta \vdash \mathbf{match\ } e \mathbf{ with\ } m \Downarrow \boxed{\text{Match}}}$$

$$\frac{\delta \vdash e_1 \Downarrow s \qquad \delta \vdash e_2 \Downarrow i}{\delta \vdash e_1.[e_2] \Downarrow \boxed{\text{Subscript}}} \quad i \notin \text{dom } s$$

38. Dynamische Semantik

☞ Wie immer müssen wir *alle* bisher aufgeführten Auswertungsregeln anpassen.

raise Div + *raise Match*

Die folgenden beiden Regeln kommen zu der ursprünglichen *hinzu*:

$$\frac{\delta \vdash e_1 \Downarrow \boxed{\nu}}{\delta \vdash e_1 + e_2 \Downarrow \boxed{\nu}} \qquad \frac{\delta \vdash e_1 \Downarrow \nu_1 \qquad \delta \vdash e_2 \Downarrow \boxed{\nu}}{\delta \vdash e_1 + e_2 \Downarrow \boxed{\nu}}$$

☞ Die Rechnung wird abgebrochen, wenn einer der beiden Summanden eine Ausnahme wirft.

38. Dynamische Semantik

Im Allgemeinen müssen wir für eine Regel mit n Voraussetzungen n zusätzliche Regeln angeben. Die Regel

$$\frac{\delta_1 \vdash e_1 \Downarrow \nu_1 \quad \delta_2 \vdash e_2 \Downarrow \nu_2 \quad \dots \quad \delta_n \vdash e_n \Downarrow \nu_n}{\delta \vdash e \Downarrow \nu}$$

wird um die folgenden n Regeln *ergänzt*:

$$\frac{\delta_1 \vdash e_1 \Downarrow \boxed{\nu}}{\delta \vdash e \Downarrow \boxed{\nu}}$$
$$\frac{\delta_1 \vdash e_1 \Downarrow \nu_1 \quad \delta_2 \vdash e_2 \Downarrow \boxed{\nu}}{\delta \vdash e \Downarrow \boxed{\nu}}$$
$$\vdots$$
$$\frac{\delta_1 \vdash e_1 \Downarrow \nu_1 \quad \delta_2 \vdash e_2 \Downarrow \nu_2 \quad \dots \quad \delta_n \vdash e_n \Downarrow \boxed{\nu}}{\delta \vdash e \Downarrow \boxed{\nu}}$$

38. Vertiefung: Panik

☞ Nicht immer lässt sich eine abgebrochene Rechnung sinnvoll weiterführen.

exception *Panic of String*

```
let panic (message : String) : 'a =  
    raise (Panic message)
```

☞ Die Funktion *panic* hat den Ergebnistyp *'a* und zeigt damit an, dass sie keinen Wert zurückgibt. (Warum?)

38. Vertiefung: *panic*

☞ Wenn man möchte, kann man *Panic* Ausnahmen abfangen — am besten mit einem *try*-Ausdruck um das komplette Programm.

```
try  
    ...  
with  
| Panic msg →  
  putline ("panic! (the 'impossible' happened)\n" ^ msg ^  
           "\nPlease report it as a bug to" ^  
           "support@harry-hacker.org.")
```

38. Vertiefung

☞ Der Begriff Ausnahme suggeriert, dass die Konstrukte **raise** und **try** tatsächlich nur in Ausnahmesituationen verwendet werden sollten.

☞ Das ist zwar prinzipiell nicht ganz falsch, schöpft aber das Potential der Konstrukte nicht aus.

Aufgabe: Produkt einer Liste von Zahlen berechnen. Mit dem Peano Entwurfsmuster erhalten wir die folgende Definition.

```
let product (list : List ⟨Nat⟩) : Nat =  
  match list with  
  | []      → 1  
  | n :: ns → n * product ns
```

☞ Enthält die Liste irgendwo eine Null, dann wird unnötige Arbeit verrichtet.

38. Vertiefung

exception Zero

let product (list : List ⟨Nat⟩) : Nat =

try

let rec worker = **function**

| [] → 1

| n :: ns → **if** n = 0 **then raise** Zero **else** n * worker ns

in worker list

with

| Zero → 0

 Mit **raise** Zero „springen wir aus der Rekursion heraus“.

38. Vertiefung: einfache Parser

☞ Mit Hilfe von Ausnahmen können wir *einfache* Parser programmieren, hier vorgeführt an der Funktion *read-nat*, die einen String in eine natürliche Zahl überführt.

exception *Read*

let *read-nat* (*s* : *String*) : *Nat* =

let rec *nat* = **function**

| [] → 0

| *c* :: *cs* → **if** *Char.isDigit c* **then**

Nat.ord c ÷ *Nat.ord '0'* + 10 * *nat cs*

else

raise *Read*

in

if *s* = "" **then raise** *Read* **else** *nat (List.rev (explode s))*

☞ Diese Version beklagt sich, wenn der String leer ist oder wenn andere Zeichen als Ziffern vorkommen.

38. Vertiefung: Validatoren

☞ Diese Version von *read-nat* können wir zum Beispiel verwenden, um die Funktion *is-nat* etwas kürzer zu definieren.

```
let is-nat (s : String) : Result <Nat> =  
  try  
    Okay (read-nat s)  
  with  
    | Read → Error "natural number expected"
```

38. Vertiefung: *Option* und *Result*

☞ Die Datentypen *Option* und *Result* dienen dem gleichen Zweck wie Ausnahmen.

<i>Option</i>	<i>Result</i>	Ausnahmen
<i>Some e</i>	<i>Okay e</i>	<i>e</i>
<i>None</i>	<i>Error e</i>	ν
<i>match ... with</i>	<i>match ... with</i>	<i>raise e</i> ν
		<i>try ... with</i>

☞ Mit Hilfe von *Option* und *Result* können/müssen wir das Werfen und Fangen von Ausnahmen selbst programmieren!

38. Vertiefung: Eingabe mit Validierung

☞ Reimplementieren wir *checked-query* mit Hilfe von Ausnahmen.

Ohne Ausnahmen: Ein Validator hat den Typ $t_1 \rightarrow \text{Result} \langle t_2 \rangle$.

Mit Ausnahmen: Ein Validator hat den Typ $t_1 \rightarrow t_2$. Eine fehlerhafte Eingabe wird jetzt durch das Werfen einer Ausnahme signalisiert.

```
let rec checked-query (prompt : String, check : String → 'a) : 'a =
  try
    check (query (prompt ^ ": "))
  with
    | Panic msg →
      putline ("*** " ^ msg); checked-query (prompt, check)
```

☞ Die Fallunterscheidung mit *match* ist einem *try*-Ausdruck gewichen.

38. Vertiefung: Eingabe mit Validierung

☞ Die Validatoren haben einen einfacheren Typ: sie geben entweder den semantischen Wert zurück oder werfen eine *Panic* Ausnahme.

```
let is-nat (s : String) : Nat =  
  try readnat s with  
    | Read → panic "natural number expected"  
  
let is-less (n : Nat) : Nat → Nat =  
  fun m → if m < n  
    then m  
    else panic ("number must be less than " ^ show n)
```

38. Vertiefung: Eingabe mit Validierung

☞ Die Funktion *both* ist jetzt schlicht und einfach die (Vorwärts-) Komposition von Funktionen.

```
let both (first : 'a → 'b, second : 'b → 'c) : 'a → 'c =  
  fun x → second (first x)
```

38. Vertiefung

☞ Für den Anwender von *checked-query* ändert sich nichts; die Aufrufe funktionieren unverändert:

```
Mini> checked-query ("age", both (is-nat, is-less 123))
```

```
age : Ralf
```

```
*** natural number expected
```

```
age : 4711
```

```
*** number must be less than 123
```

```
age : 41
```

```
41
```

38. Vertiefung: *Option* und *Result*

Vorteile von Option und Result:

- ▶ Der Typ von Ausdrücken und Funktionen macht explizit, wer „Ausnahmen wirft“ und wer nicht.
- ▶ Man kann nicht vergessen, Ausnahmen zu behandeln.
- ▶ Man kommt nicht in Versuchung, Ausnahmen zu fangen, die gar nicht geworfen werden.

Nachteil von Option und Result:

- ▶ Die Auswertungsregeln für Ausnahmen müssen im Prinzip nachprogrammiert werden.

38. Vertiefung: *Option und Result*

Mit *Ausnahmen*:

$e_1 + e_2$

Mit *Result*:

```
match e1 with  
  | Okay n1 → match e2 with  
    | Okay n2 → Okay (n1 + n2)  
    | Error msg → Error msg  
  | Error msg → Error msg  
end
```

 Jeder der drei Zweige korrespondiert zu einer Auswertungsregel.

38. Vertiefung: Taschenrechner

Aufgabe: Programmierung eines interaktiven Taschenrechners.

Variante 1: ein UPN-Rechner. *UPN* steht für Umgekehrte Polnische Notation, ein Synonym für Postfixnotation.

```
UPN> 4711
UPN> 815
UPN> 2765
UPN> *
UPN> +
UPN> .
2258186
```

 Der Postfix-Ausdruck wird Zeile für Zeile eingegeben: 4711 815 2765 * + entspricht dem Infix-Ausdruck $4711 + 815 * 2765$.

38. Vertiefung: UPN-Rechner

☞ Ausdrücke in Postfixnotation lassen sich besonders leicht ausrechnen: Wir merken uns die eingegebenen Zahlen; jede Operation ersetzt die letzten beiden Zahlen durch das Ergebnis.

4711	4711
815	4711 815
2765	4711 815 2765
*	4711 2253475
+	2258186

☞ Die Liste der Zahlen nennt man auch *Stack* oder *Stapel*: optisch muss man die Listen um 90° nach links drehen.

38. Vertiefung: UPN-Rechner

☞ Einen Stapel können wir durch eine Speicherzelle repräsentieren, die eine Liste von Zahlen enthält.

```
exception Pop
exception Top

module Stack =
  let private stack = ref []

  let push (x : Nat) =
    stack := x :: !stack

  let pop () : Nat =
    match !stack with
    | []      → raise Pop
    | x :: xs → stack := xs; x

  ...
```

38. Vertiefung: UPN-Rechner

...

```
let top () : Nat =  
  match !stack with  
  | []      → raise Top  
  | x :: xs → x
```

☞ Die Funktion *push* legt ein Element auf dem Stapel ab, *pop* entfernt ein Element und *top* inspiziert das oberste Element.

☞ Die Speicherzelle, auf der die Funktionen arbeiten, ist gekapselt, also nur lokal sichtbar.

38. Vertiefung: UPN-Rechner

☞ Der UPN-Rechner implementiert eine einfache Kommandozeile.

```
let rec upn-calculator () =  
  try  
    match query "UPN > " with  
      | "" → ()  
      | "+" → Stack.push (Stack.pop () + Stack.pop ())  
      | "*" → Stack.push (Stack.pop () * Stack.pop ())  
      | "." → print (Stack.top ())  
      | s → Stack.push (read-nat s)  
  with  
    | Pop | Top → putline "stack is empty"  
    | Read → putline "enter a number or an operator"  
upn-calculator ()
```

38. Vertiefung: Mini²-F# Rechner

Variante 2: ein Mini²-F# Rechner, der Infixnotation unterstützt.

```
Mini2> 4711 + 815 * 2765
```

```
2258186
```

```
Mini2> 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10
```

```
55
```

```
Mini2> 1 * 2 * 3 * 4 * 5 * 6 * 7 * 8 * 9 * 10
```

```
3628800
```

```
Mini2> Hello, world !
```

```
lexical error : illegal character
```

```
Mini2> 4711 815 2765 * +
```

```
syntax error
```

38. Vertiefung: Mini²-F# Rechner

```
let rec read-eval-print-loop () =  
  try  
    let s = query "Mini-BPL > "  
    if s = "" then  
      ()  
    elif contains s ["q"; "quit"; "halt"; "stop"] then  
      raise EOF  
    else  
      match abstract-syntax-tree s with  
        | None → putline "syntax error"  
        | Some e → print (evaluate e)  
  with  
    | Panic s → putline ("lexical error: " ^ s)  
  read-eval-print-loop ()
```

☞ Typisch für einen Kommandozeileninterpreter (REPL) ist der rekursive Aufruf am Ende.

38. Vertiefung: Mini²-F# Rechner

```
let mini2 () =  
  try  
    read-eval-print-loop ()  
  with  
    | EOF → putline "bye bye"
```

☞ Signalisiert die Benutzer*in das Ende der Eingabe, dann wirft *query* eine *EOF* Ausnahme (end of file).

38. Paradigmen: imperative Programmierung

imperare

1. befehlen, gebieten, anordnen;
2. auferlegen, zu liefern befehlen;
3. herrschen, den Oberbefehl haben.

C is quirky, flawed, and an enormous success.

— Dennis M. Ritchie

B can be thought of C without types; more accurately,
it is BCPL squeezed into 8K bytes of memory and
filtered through Thompson's brain.

— Dennis M. Ritchie

38. Lösung Knobelaufgabe #17

Welche Sprache bezeichnet der folgende reguläre Ausdruck?

$$riddle = (a a \mid b b)^* ((a b \mid b a) (a a \mid b b)^* (a b \mid b a) (a a \mid b b)^*)^*$$

Beobachtung: jedes abgeleitete Wort hat eine gerade Länge.

Beobachtung: der Ausdruck *riddle* hat die gleiche Struktur wie

$$0^* (1 0^* 1 0^*)^*$$

☞ Der Ausdruck bezeichnet die Sprache aller Wörter mit einer geraden Anzahl von Einsen und beliebig vielen Nullen.

38. Lösung Knobelaufgabe #17

$$riddle = (a a \mid b b)^* ((a b \mid b a) (a a \mid b b)^* (a b \mid b a) (a a \mid b b)^*)^*$$

bezeichnet die Sprache aller Wörter mit einer geraden Anzahl von Wörter der Form $a b \mid b a$ und beliebig vielen Wörtern der Form $a a \mid b b$.

☞ *Also:* eine gerade Anzahl von Wörtern, die eine ungerade Anzahl *as* *und* eine ungerade Anzahl *bs* enthalten, und eine beliebige Anzahl von Wörtern, die eine gerade Anzahl *as* *und* eine gerade Anzahl *bs* enthalten.

☞ *Mit anderen Worten:* die Wörter enthalten eine gerade Anzahl *as* *und* eine gerade Anzahl *bs*.

☞ *riddle / a* enthält entsprechend eine ungerade Anzahl *as* *und* eine gerade Anzahl *bs*;
riddle / b enthält eine gerade Anzahl *as* *und* eine ungerade Anzahl *bs*.