Script generated by TTT

Title: Seidl: Functional Programming and Verification (14.12.2018)

Date: Fri Dec 14 08:33:46 CET 2018

Duration: 85:56 min

Pages: 17

Pre-defined Constructors for Exceptions

`Division_by_zero`	division by 0
`Invalid_argument of string`	wrong usage
`Failure of string`	general error
`Match_failure of string * int * int`	incomplete match
`Not_found`	not found
`Out_of_memory`	memory exhausted
`End_of_file`	end of file
`Exit`	for the user ...

An exception is a first class citizen, i.e., a value from a datatype `exn` ...

Ausnahmebehandlung

As in Java, exceptions can be raised and handled:

```
# let teile (n,m) = try Some (n / m)
      with Division_by_zero -> None;;

# teile (10,3);;
- : int option = Some 3
# teile (10,0);;
- : int option = None
```

In this way, the `member` function can, e.g., be re-defined as


```
let rec member x l = try if x = List.hd l then true
                         else member x (List.tl l)
    with Failure _ -> false

# member 2 [1;2;3];;
- : bool = true
# member 4 [1;2;3];;
- : bool = false
```

Following the keyword `with`, the exception value can be inspected by means of pattern matching for the exception datatype `exn` :

```
try <exp>
with <pat1> -> <exp1> | ... | <patN> -> <expN>
```

$\Longrightarrow$ several exceptions can be caught (and thus handled) at the same time.

The programmer may trigger exceptions on his/her own by means of the keyword `raise` ...

```
# 1 + (2/0);;
Exception: Division_by_zero.
# 1 + raise Division_by_zero;;
Exception: Division_by_zero.
```

An exception is an error value which can replace any expression.

Handling of an exception, results in the evaluation of another expression (of the correct type) — or raises another exception.

Exception handling may occur at any sub-expression, arbitrarily nested:

```
# let f (x,y) = x / (y-1);;
# let g (x,y) = try let n = try f (x,y)
                            with Division_by_zero ->
                                 raise (Failure "Division by zero")
                  in string_of_int (n*n)
                with Failure str -> "Error: "^str;;

# g (6,1);;
- : string = "Error: Division by zero"
# g (6,3);;
- : string = "9"
```


5.2 Textual Input and Output

- Reading from the input and writing to the output violates the paradigm of purely functional programming !
- These operations are therefore realized by means of side-effects, i.e., by means of functions whose return value is irrelevant (e.g., `unit`).
- During execution, though, the required operation is executed

  $\Longrightarrow$ now, the ordering of the evaluation matters !!!

- Naturally, Ocaml allows to write to standard output:

```
# print_string "Hello World!\n";;
Hello World!
- : unit = ()
```

- Analogously, there is a function: `read_line : unit -> string` ...

```
# read_line ();;
Hello World!
- : string = "Hello World!"
```

In order to read from file, the file must be opened for reading ...

```
# let infile = open_in "test";;
val infile : in_channel = <abstr>
# input_line infile;;
- : string = "Die einzige Zeile der Datei ...";;
# input_line infile;;
Exception: End_of_file
```

If there is no further line, the exception `End_of_file` is raised.

If a channel is no longer required, it should be explicitly closed ...

```
# close_in infile;;
- : unit = ()
```


Further Useful Values

```
stdin             : in_channel
input_char        : in_channel -> char
in_channel_length : in_channel -> int
```

- `stdin` is the standard input as channel.
- `input_char` returns the next character of the channel.
- `in_channel_length` returns the total length of the channel.

Further Useful Values

```
stdin             : in_channel
input_char        : in_channel -> char
in_channel_length : in_channel -> int
```

- `stdin` is the standard input as channel.
- `input_char` returns the next character of the channel.
- `in_channel_length` returns the total length of the channel.

Output to files is analogous ...

```
# let outfile = open_out "test";;
val outfile : out_channel = <abstr>
# output_string outfile "Hello ";;
- : unit = ()
# output_string outfile "World!\n";;
- : unit = ()
...
```

Die einzeln geschriebenen Wörter sind mit Sicherheit in der Datei erst zu finden, wenn der Kanal geregelta The words written seperately, may only occur inside the file, once the file has been closed ...

```
# close_out outfile;;
- : unit = ()
```

Output to files is analogous ...

```
# let outfile = open_out "test";;
val outfile : out_channel = <abstr>
# output_string outfile "Hello ";;
- : unit = ()
# output_string outfile "World!\n";;
- : unit = ()
...
```

Die einzeln geschriebenen Wörter sind mit Sicherheit in der Datei erst zu finden, wenn der Kanal geregelta The words written seperately, may only occur inside the file, once the file has been closed ...

```
# close_out outfile;;
- : unit = ()
```


On this input, the compiler answers with the type of the module, its signature:

```
module Pairs :
  sig
    type 'a pair = 'a * 'a
    val pair : 'a * 'b -> 'a * 'b
    val first : 'a * 'b -> 'a
    val second : 'a * 'b -> 'b
  end
```

The definitions inside the module are not visible outside:

```
  # first;;
  Unbound value first
```


Access onto Components of a Module

Components of a module can be accessed via qualification:

```
# Pairs.first;;
- : 'a * 'b -> 'a = <fun>
```

Thus, several functions can be defined all with the same name:

```
# module Triples = struct
    type 'a triple = Triple of 'a * 'a * 'a
    let first (Triple (a,_,_)) = a
    let second (Triple (_,b,_)) = b
    let third (Triple (_,_,c)) = c
  end;;
...
```



```
# module A = struct let x = 1 end;;
module A : sig val x : int end
# module B = struct
    open A
    let y = 2
  end;;
module B : sig val y : int end
# module C = struct
    include A
    include B
  end;;
module C : sig val x : int val y : int end
```

Nested Modules

Modules may again contain modules:

```
module Quads  = struct
    module Pairs = struct
        type 'a pair = 'a * 'a
        let pair (a,b) = (a,b)
        let first (a,_) = a
        let second (_,b) = b
      end
    type 'a quad = 'a Pairs.pair Pairs.pair
    let quad (a,b,c,d) =
        Pairs.pair (Pairs.pair (a,b), Pairs.pair (c,d))
    ...
```


Nested Modules

Modules may again contain modules:

```
module Quads  = struct
    module Pairs = struct
        type 'a pair = 'a * 'a
        let pair (a,b) = (a,b)
        let first (a,_) = a
        let second (_,b) = b
      end
    type 'a quad = 'a Pairs.pair Pairs.pair
    let quad (a,b,c,d) =
        Pairs.pair (Pairs.pair (a,b), Pairs.pair (c,d))
    ...
```

291