Computation expressions

```
builder-expr { cexpr } =
  let b = builder-expr in b.Run (b.Delay(fun () -> {| cexpr |}))
```

Pokud `Run` nebo `Delay` neexistují, nezavolají se.

Přepisovací pravidla

```
{| let binds in cexpr |}                  = let binds in {| cexpr |}
{| let! pat = expr in cexpr |}            = b.Bind(expr, (fun pat -> {| cexpr |}))
{| do expr in cexpr |}                    = expr; {| cexpr |}
{| do! expr in cexpr |}                   = b.Bind(expr, (fun () -> {| cexpr |}))
{| yield expr |}                          = b.Yield(expr)
{| yield! expr |}                         = b.YieldFrom(expr)
{| return expr |}                         = b.Return(expr)
{| return! expr |}                        = b.ReturnFrom(expr)
{| use pat = expr in cexpr |}             = b.Using(expr, (fun pat -> {| cexpr |}))
{| use! v = expr in cexpr |}              = b.Bind(expr, (fun v ->
                                                 b.Using(v,(fun v -> {| cexpr |})))
{| if expr then cexpr0 |}                 = if expr then {|cexpr0|} else b.Zero()
{| if expr then cexpr0 else cexpr1 |}     = if expr then {|cexpr0|} else {|cexpr1|}
{| match expr with p_i -> cexpr_i |}      = match expr with p_i -> {| cexpr_i |}
{| for pat in enumeration do cexpr |}     = b.For(enumeration,fun pat -> {|cexpr|})
{| for idn=expr1 to expr2 do cexpr |}     = b.For(enumeration,fun idn -> {|cexpr|})
{| while expr do cexpr |}                 = b.While((fun () -> expr), {|cexpr|}Del)
{| try cexpr with p_i -> cexpr_i |}       = b.TryWith({| cexpr |}Del, (fun v ->
                                            match v with | (p_i:exn) -> {|cexpr_i|}
                                                         | _ -> raise exn)
{| try cexpr finally expr |}      = b.TryFinally( {| cexpr |}Del, (fun () -> expr))
{| cexpr0; cexpr1 |}              = b.Combine({| cexpr0 |}, {| cexpr1 |}Del)
{| other-expr0 ; cexpr1 |}        = other-expr; {| cexpr1 |}
{| other-expr |}                  = other-expr; b.Zero()
```

kde `{| cexpr |}Del` je `b.Delay(fun () -> {| cexpr |})`

Stm<'a>

Software transaction memory –– využití optimistického zamykacího protokolu na sdílenou paměť.

- ♣ `Stm<'a>` –– výpočet, který používá sdílenou paměť a vrací výsledek typu `'a`
- ♣ `TVar<'a>` –– sdílená proměnná (transaction variable), obsahuje hodnotu typu `'a`
  - `newTVar (value : 'a) : TVar<'a>` –– vytvoří novou inicializovanou sdílenou proměnnou
  - `readTVar (ref : TVar<'a>) : Stm<'a>` –– výpočet, který načte proměnnou
  - `writeTVar (ref : TVar<'a>) (value : 'a) : Stm<unit>` –– výpočet zapisující do proměnné
- ♣ `atomically (a : Stm<'a>) : 'a` –– provede výpočet, a to atomicky vzhledem k transakční paměti
- ♣ `retry () : Stm<'a>` –– "výpočet", který selže, říká "s touto hodnotou sdílené paměti nemohu běžet"
- ♣ `orElse (a : Stm<'a>) (b : Stm<'a>) : Stm<'a>` –– spustí první výpočet a pokud tento zavolá `retry`, spustí druhý výpočet. Pokud i tento zavolá `retry`, spustí se znovu

Můžeme například vytvořit jednoprvkovou frontu, vkládání a vybírání "blokuje" volající vlákno.

```
type Box<'a> = Box of TVar<'a option>

let emptyBox<'a> : Box<'a> = Box (newTVar None)
let readBox (Box box) = stm {
  let! content = readTVar box
  match content with
  | Some value -> return value
  | None -> return! retry()
}
let writeBox (Box box) value = stm {
  let! content = readTVar box
  match content with
  | None -> do! writeTVar box (Some value)
            return value
  | Some _ -> return! retry()
}
```

Nebo úplnou frontu, která blokuje jenom při vybírání prázdné fronty:

```
type Queue<'a> = Q of TVar<'a list*'a list>

let emptyQueue<'a> : Queue<'a> = Q (newTVar ([],[]))
```

```
let dequeue (Q queue) = stm {
  let! h, t = readTVar queue
  let h, t = if h.IsEmpty then List.rev t, [] else h, t
  match h, t with
  | x::xs,ys -> do! writeTVar queue (xs,ys)
                return x
  | _ -> return! retry()
}
let enqueue x (Q queue) = stm {
  let! h, t = readTVar queue
  return! writeTVar queue (h,x::t)
}

let q = emptyQueue<int>
atomically (enqueue 1 q)
let hd = atomically (dequeue q)
```

Čtení z několika front:

```
let dequeAny queues =
  queues |> Seq.map dequeue
         |> Seq.reduce orElse

let dequeAny queues =
  queues |> Seq.mapi (fun i q -> stm { let! x = dequeue q
                                       return x, i; })
         |> Seq.reduce orElse

let dequeAny queues =
  queues |> Seq.mapi (fun i q -> stm.Bind(dequeue q, fun x -> stm.Return(x, i)))
         |> Seq.reduce orElse

let dequeAny queues =
  let rec dequeAny’ num (q::qs) = stm {
    let! x = dequeue q
    return x, num
    if not qs.IsEmpty then return! dequeAny’ (num+1) qs
  }
  dequeAny’ 0 queues
```

MailboxProcessor

```
type Action = Store of string * string
            | Query of string * AsyncReplyChannel<string>

let storage_server =
  let storage = new Dictionary<string, string>()
  let rec listen (inbox:MailboxProcessor<_>) = async {
    let! action = inbox.Receive()
    match action with
      | Store (key, value) ->
          storage.[key] <- value
      | Query (key, channel) ->
          channel.Reply(if storage.ContainsKey(key)
                        then storage.[key] else null)
    return! listen inbox
  }
  MailboxProcessor.Start listen

storage_server.Post(Store ("klíč", "hodnota"))
storage_server.PostAndReply(fun chnl -> Query ("klíč", chnl)) |> printfn "%A"
```

♣ Zevnitř `MailboxProcessor`u lze používat:

`member Receive : ?int -> Async<’Msg>` čeká na zprávu, výjimka když timeout

`member TryReceive : ?int -> Async<’Msg option>` čeká na zprávu, `None` když timeout

`member Scan : (’Msg -> Async<’T> option) * ?int -> Async<’T>`

`member TryScan : (’Msg -> Async<’T> option) * ?int -> Async<’T option>`

Vrátí první zprávu z fronty, která vyhovuje danému filteru. Pokud je dán timeout, první varianta po něm vyhodí výjimku, druhá vrátí `None`

`member CurrentQueueLength : int` Aktuální délka fronty zpráv

♣ Zvenku `MailboxProcessor`u lze používat:

```
member Post : 'Msg -> unit                                    Pošle zprávu a okamžitě uspěje
member PostAndReply : (AsyncReplyChannel<'Reply> -> 'Msg) * int option -> 'Reply
  Pošle zprávu a synchronně čeká na odpověď, která přijde skrz daný kanál
member TryPostAndReply : (AsyncReplyChannel<'Reply> -> 'Msg) * ?int
  -> 'Reply option          Jako PostAndReply, ale když nastane timeout, vrátí None
member PostAndAsyncReply : (AsyncReplyChannel<'Reply> -> 'Msg) * ?int
  -> Async<'Reply>          Jako PostAndReply, jenom na odpověď čeká asynchronně
member PostAndTryAsyncReply : (AsyncReplyChannel<'Reply> -> 'Msg) * ?int
  -> Async<'Reply option>  Jako PostAndAsyncReply, ale když nastane timeout, vrátí None
member CurrentQueueLength :  int                    Aktuální délka fronty zpráv
member DefaultTimeout :  int with get, set   Výchozí timeout, −1 znamená žádný
member Error :  IEvent<Exception>                Event, když v processoru nastave výjimka
static member Start:(MailboxProcessor<'Msg> -> Async<unit>) * ?CancellationToken
  -> MailboxProcessor<'Msg>                        Spustí nový processor
```

Continuation passing style

```
let square x = x * x
let squareK x k = x * x |> k

type ContBuilder() =
  member this.Return(x) = fun k -> k x
  member this.ReturnFrom(k) = k
  member this.Bind(a, f) = fun k -> a (fun l -> (f l) k)
  member this.Zero() = fun k -> k ()
  member this.Delay a = a ()
  member this.Combine(a, b) = fun k -> a (fun () -> b k)
let cont = new ContBuilder()
let runC k = k id

let squareC x = cont { return x*x }
let sqrtC n = cont { if n >= 0 then return n|>float|>sqrt|>int else return -1 }
let compC n = cont { let! k = sqrtC n
                     return! squareC (k+3) }
runC <| squareC 10
```

CPS a callCC

```
let callCC f = fun k -> f (fun l -> (fun _ -> k l)) k

let squareC x = cont { return        x * x }
let squareC x = callCC <| fun k -> k (x * x)
let foo n = callCC <| fun k -> cont { let n' = n*n + 3
                                      if n' > 20 then return! k "over twenty\n"
                                      return string n' + "\n" }
```

Výjimky pomocí callCC

```
let sqrtExcept n handler =
  callCC <| fun ok ->
    cont { let! err = callCC <| fun notOk ->
                                      cont { if n < 0. then return! notOk "!!!"
                                             return! ok (sqrt n)
                                           }
           return! handler err
         }

let tryCont k handler =
  callCC <| fun ok ->
    cont { let! err = callCC <| fun notOk -> cont { let! x = k notOk
                                                    return! ok x }
           return! handler err }

type SqrtException = LessThanZero
let sqrtExc n throw = cont { if n < 0. then return! throw LessThanZero
                             return sqrt n }
runC <| tryCont (sqrtExc -3.) (fun n -> printfn "%A" n; exit 1)
```

Použití Continuation passing style k redukci zásobníku

```
let rec qsort = function   //val qsort: 'a list -> 'a list
  | [] -> []
  | x::xs' -> let (l, r) = List.partition ((>) x) xs'
              List.concat [(qsort l);[x];(qsort r)]

let qsortCPS xs =        //val qsortCPS: 'a list -> 'a list
    let rec loop xs cont = match xs with
      | [] -> cont []
      | x::xs' -> let (l, r) = List.partition ((>) x) xs'
                  loop l (fun lacc ->
                  loop r (fun racc -> cont (lacc @ x :: racc)))
    loop xs (fun x -> x)
```

Cizí kód

```
let morseTable = [         'A', ".-"  ; 'B', "-..."; 'C', "-.-."; 'D', "-.." ;
 'E', "."   ; 'F', "..-."; 'G', "--." ; 'H', "...."; 'I', ".."  ; 'J', ".---";
 'K', "-.-" ; 'L', ".-.."; 'M', "--"  ; 'N', "-."  ; 'O', "---" ; 'P', ".--.";
 'Q', "--.-"; 'R', ".-." ; 'S', "..." ; 'T', "-"   ; 'U', "..-" ; 'V', "...-";
 'W', ".--" ; 'X', "-..-"; 'Y', "-.--"; 'Z', "--.."; ]
let toMorse s =
    let d = dict morseTable
    System.String.Join("", [|for c in s do yield d.[c]|])
let rec possiblyNextLetters n (morse:string) =
    match n, morse with
    | 0,_ -> [[' ']]
    | _,"" -> [[' ']]
    | _ ->
      [for c,m in morseTable do
          if morse.StartsWith(m) then
              let r = possiblyNextLetters (n-1) (morse.Substring(m.Length))
              let r2 = [for x in r -> c::x]
              yield! r2]

open System
let mutable committed = ""
let toDecode = "......-...-..---.-----.-..-..-.."
while true do
    let committedMorse = toMorse committed
    let restMorse = toDecode.Substring(committedMorse.Length)
    assert(toDecode.StartsWith(committedMorse))
    Console.BackgroundColor <- ConsoleColor.Blue
    Console.Write(" {0}", committedMorse)
    Console.BackgroundColor <- ConsoleColor.Black
    Console.WriteLine(restMorse)
    let nexts = possiblyNextLetters 3 restMorse
                |> List.map (fun cs -> System.String(Seq.toArray cs))
    for n in nexts do
        Console.BackgroundColor <- ConsoleColor.Blue
        Console.Write(" {0}", committed)
        Console.BackgroundColor <- ConsoleColor.Black
        Console.WriteLine(n)
    let k = Console.ReadKey()
    Console.WriteLine()
    if k.Key = ConsoleKey.Backspace && committed.Length > 0 then
        committed <- committed.Substring(0, committed.Length - 1)
    else
        let k = k.KeyChar
        let k = System.Char.ToUpper(k)
        if k >= 'A' && k <= 'Z' then
            if nexts |> Seq.exists (fun s -> s.StartsWith(string k)) then
                committed <- committed + string k
            else
                Console.WriteLine(" Not a legal next char! ")
        else
            Console.WriteLine(" Press a letter to commit, <- to uncommit one ")
    Console.WriteLine()
```