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 |})`

Parallel for

```
type PForBuilder() =
  member this.Zero() = ()
  member this.For(s, action) =
    System.Threading.Tasks.Parallel.ForEach(s, action)

let pfor = PForBuilder()

let test = pfor {
  for i = 1 to 10000 do printfn "%A" i
}
```

computation expressions -- Maybe

```
type MaybeBuilder() =
  member x.Return a = Some a
  member x.Bind(a, f) = match a with | None -> None
                                     | Some a -> f a
  member x.Zero() = None

let maybe = new MaybeBuilder()
let inc a = maybe { let! v = a
                    return v + 1 }
```

Computation expressions -- Parser

```
type 'a Parser = char list -> seq<'a * char list>

type ParserBuilder() =
  member x.Return a : 'a Parser = fun s -> Seq.singleton (a, s)
  member x.Bind(a, f) : 'a Parser =
    fun s -> a s |> Seq.collect (fun (b, s') -> s' |> f b)
let parser = new ParserBuilder()
```

```
let char : char Parser = function
    | [] -> Seq.empty
    | s::ss -> Seq.singleton (s, ss)

type ParserBuilder with
  member x.Zero() : 'a Parser = fun s -> Seq.empty
  member x.Delay a = fun s -> Seq.delay (fun () -> a () s)
  member x.Combine(a, b) : 'a Parser = fun s -> Seq.append (a s) (b s)
  member x.ReturnFrom(p) = p

let sat pred = parser { let! c = char
                        if pred c then return c }
let space = sat System.Char.IsWhiteSpace
let digit = sat System.Char.IsDigit

let rec many1 p = parser { let! r = p
                           let! rs = many p
                           return r::rs }
    and many p = parser { return! many1 p
                          return [] }
let spaces = many space
let digits = many digit
let number = parser { let! ds = digits
                      return List.fold (fun n d -> n * 10 + int d - int '0') 0 d
s }

let addop = parser { let! op = char
                     if op = '+' then return (+)
                     if op = '-' then return (-) }

let aplusb = parser { let! a = number
                      let! _ = spaces
                      let! op = addop
                      let! _ = spaces
                      let! b = number
                      return op a b }

let parse parser str =
  for res in parser (List.ofSeq str) do
    printf "%A %A\n" (fst res) (snd res)

parse aplusb "123 + 223"
```

Async<'T>

```
type AsyncBuilder() =
  ...
let async = new AsyncBuilder()
```

Nejdůležitější je operace `Bind`. Při jejím volání dochází k asynchronnímu čekání na výsledek, pokud není k dispozici. Poté, co je výsledek vypočten, dochází v nějakém vláknu threadpoolu k pokračování.