```
                                    Vlákna v .NET
                                    -------------

open System.Threading

let mutable n = 0
let inc () = n <- n + 1

let many_inc num =
  n <- 0
  let threads = [ for i in [1..num] -> new Thread(inc) ]
  threads |> List.iter (fun t -> t.Start())
  threads |> List.iter (fun t -> t.Join())
  n


                                 Interlocked operace
                                 -------------------
let nr = ref 0
let incr() = Interlocked.Increment nr

Interlocked.{Increment,Decrement,Exchange}
Interlocked.Read (* cte 64-bitovou hodnotu atomicky *)
Interlocked.Add  (* k prvnimu intu pricte druhy *)
Interlocked.CompareExchange(ref x, val, comp) (* x=val if x==comp;return ori x*)


                                      Zamykání
                                      --------
let incl() = lock nr (fun () -> nr := !nr + 1)

let inline lock (lockobj : 'a when 'a : not struct) f  =
    System.Threading.Monitor.Enter(lockobj);
    try f()
    finally System.Threading.Monitor.Exit(lockobj)

System.Threading.Mutex.{WaitOne(),ReleaseMutex()}
system.Threading.Semapthore.{WaitOne(),Release()}
let inline rlock (rwlock : ReadWriterLock) f =
    rwlock.AcquireReaderLock(Timeout.Infinite)
    try f()
    finally rwlock.ReleaseReaderLock()
let inline wlock (rwlock : ReadWriterLock) f =
    rwlock.AcquireWriterLock(Timeout.Infinite)
    try f()
    finally rwlock.ReleaseWriterLock()

                              Software Transaction Monad
                              --------------------------
#r "STM.Core.dll"
#r "STM.dll"

open Stm
let nt = newTVar 0
let inc () = atomically <| stm { let! n = readTVar nt
                                 do! writeTVar nt (n+1) }

let many_inc num =
  atomically (writeTVar nt 0)
  let threads = [ for i in [1..num] -> new Thread(inc) ]
  threads |> List.iter (fun t -> t.Start())
  threads |> List.iter (fun t -> t.Join())
  atomically (readTVar nt)
```

Stm.fs

```
type Stm<'a> = (Stm.Core.TLog -> 'a)
type TVar<'a> = Stm.Core.TVar<'a>
let newTVar (value : 'a) : TVar<'a> =
    Stm.Core.TLog.NewTVar(value)
let readTVar (ref : TVar<'a>) : Stm<'a> =
    fun trans -> trans.ReadTVar(ref)
let writeTVar (ref : TVar<'a>) (value : 'a) : Stm<unit> =
    fun trans -> trans.WriteTVar(ref, value)
let retry () : Stm<'a> =
    fun trans -> trans.Retry<_>()
let orElse (a : Stm<'a>) (b : Stm<'a>) : Stm<'a> =
    fun trans -> trans.OrElse<_>((fun x -> a x), (fun x -> b x))
let atomically (a : Stm<'a>) : 'a =
    Stm.Core.TLog.Atomic<_>(fun x -> a x)

type StmBuilder () =
    member b.Return(x) : Stm<_> = fun _ -> x
    member b.Bind(p : Stm<_>, rest : _ -> Stm<_>) : Stm<_> =
        fun trans -> rest (p trans) trans
    member b.Let(p, rest) : Stm<_> = rest p
    member b.Delay(f : unit -> Stm<_>) : Stm<_> = fun trans -> f () trans
    member b.Combine(p, q) : Stm<_> = orElse p q
    member b.Zero() = retry ()
let stm = new StmBuilder ()

let ifM p x = if p then x else stm.Return(())
let liftM f x = stm { let! x' = x in return f x' }
let sequence (ms : seq<Stm<_> >) : Stm<seq<_> > =
    fun trans -> ms |> Seq.map (fun x -> x trans) |> Seq.cache |>
                                                          (fun t -> t :> seq<_>)
let mapM f ms = ms |> Seq.map f |> sequence
let sequence_ (ms : seq<Stm<_> >) : Stm<_> =
    fun trans -> ms |> Seq.iter (fun x -> x trans)
let mapM_ f ms = ms |> Seq.map f |> sequence_
```

STM Retry

```
let produce qt x () =
  atomically <| stm { let! q = readTVar qt
                      do! writeTVar qt (seq { yield x; yield! q }) }

let consume qt lt () =
  atomically <| stm { let! q = readTVar qt
                      return! if Seq.length q < 10
                              then retry()
                              else stm { do! writeTVar qt (Seq.skip 10 q)
                                         let! l = readTVar lt
                                         do! writeTVar lt (l-1)
                                         return () } }

let work() =
  let qt = newTVar (Seq.empty)
  let lt = newTVar 10
  [ for _ in [1..10] -> (new Thread(consume qt lt)).Start() ] |> ignore
  [ for i in [1..100] -> (new Thread(produce qt i)).Start() ] |> ignore
  atomically <| stm { let! l = readTVar lt
                      printf "Have %d, waiting\n" l
                      if l > 0 then return! retry() else return () }
```

Parsování

```
(* convertHtml.fsl *)
rule convertHtml chan = parse
  | '<' { fprintf chan "&lt;";
          convertHtml chan lexbuf }
  | '>' { fprintf chan "&gt;";
          convertHtml chan lexbuf }
  | eof { () }
  | _   { fprintf chan "%s" (Lexing.lexeme lexbuf);
          convertHtml chan lexbuf }

open ConvertHtml
open System.IO
open System.Text
let convert (fin : string) (fout : string) =
  use rin = new StreamReader(fin)
  let lexBuf = Lexing.from_text_reader Encoding.ASCII rin

  use rout = new StreamWriter(fout)
  convertHtml rout lexBuf

LexBuffer.{StartPos,EndPos}
Lexing.{from_string, from_text_reader, from_binary_reader, lexeme}

(* convertHtml.fsl *)
rule convertHtml chan = parse
  ...
  | '\n' | '\r' '\n' { lexbuf.EndPos <- lexbuf.EndPos.NextLine;
                       convertHtml chan lexbuf }
  ...

(* calc.fsl *)
{
type token =
  | INT of int
  | FLOAT of float
  | ID of string
  | STRING of string
  | PLUS | MINUS | TIMES | HAT
  | EOF
}

let num = ['0'-'9']+
let intNum = '-'? num
let floatNum = '-'? num ('.' num)? (['e' 'E'] num)?
let ident = ['a'-'z']+
let whitespace = ' ' | '\t'
let newline = '\n' | '\r' '\n'

rule token = parse
  | intNum { INT (Int32.of_string (Lexing.lexeme lexbuf)) }
  | floatNum { FLOAT (Float.of_string (Lexing.lexeme lexbuf)) }
  | ident { ID (Lexing.lexeme lexbuf) }
  | '+' { PLUS }
  | '-' { MINUS }
  | '*' { TIMES }
  | '^' { HAT }
  | whitespace { token lexbuf }
  | newline { lexbuf.EndPos <- lexbuf.EndPos.NextLine; token lexbuf }
  | eof { EOF }
  | _ { failwithf "unrecognized input: '%s'" (Lexing.lexeme lexbuf) }
```

```
(* comments.fsl *)
rule token = parse
  ...
  | "(*"  { comment lexbuf; token lexbuf }

and comment = parse
  | "(*"  { comment lexbuf; comment lexbuf }
  | "*)"  { () }
  | eof   { failwith "Unterminated comment!" }
  | _     { comment lexbuf }
```

Gramatický parser

```
(* calc.fsy *)
%{
open ...
%}

%start start

%token <int> INT
%token PLUS MINUS TIMES DIV LPAREN RPAREN

%type <int> start

%%

start: Expr                { $1 }

Expr: Term PLUS Expr       { $1 + $3 }
    | Term MINUS Expr      { $1 + $3 }
    | Term                 { $1 }

Term: Factor TIMES Term    { $1 * $3 }
    | Factor DIV Term      { $1 / $3 }
    | Factor               { $1 }

Factor: INT                { $1 }
      | LPAREN Expr RPAREN { $2}

val start : (LexBuffer<_> -> token) -> LexBuffer<_> -> int
```