LLite : language friendly literate programming

Luca Bolognese

22/11/2012

Contents

1 Main ideas

My interest in literate programming comes from some realizations on my part:

- When I go back to code that I have written some time ago, I don't remember my reasoning
- When I write a blog post, my code seems to be better. Perhaps explaining things to people encourages me to be more precise
- I like to think top down, but the compiler forces me to write code bottom up, starting from details and going to higher level concepts

1.1 Unhappiness with existing tools

Many of the existing literate programming tools work similarly to the original CWeb.

- They have a tangle program that goes over your file and extract something that the compiler can understand
- They have a weave program that extracts from your file something that the document generator can understand

This scheme has the unfortunate limitation of breaking your code editor. Given that your file is not a valid code file anymore, the editor starts misbehaving (i.e. intellisense breaks). The debugger starts to get confused (albeit people tried to remediate that with cleaver use of `#line`. If your language has an interactive console, that would not work either.

1.2 A different interpretation

The main idea of this program is to add your narrative to the comment part of a code file by extending the comment tag (i.e. in C you could use /**). This keeps editor, debugger and interactive console working.

The weave phase as been retained and what you are reading is the program that goes over your code file and extracts a nicely formatted (for this program in 'markdown' format) file that can then be translated to HTML, PDF, latex, etc...

You got that? *The document you are reading now **is** the program.*

1.3 Multi-language, multi-document format

LLite works for any programming language, assuming it has open and close *comment* character sequences, and any documentation format, assuming it has open and close *code* character sequences (aka allows you to delimitate your code somehow), or it needs the code to be indented. This document uses markdown (with Pandoc extensions to generate table of contents and titles).

1.4 Usage

You invoke the program as documented below. The first set of parameters lets you choose the symbols that delimitate your language comments (or the default symbols below). The second set of parameters lets you choose how your target documentation language treats code. Either it delimits it with some symbols or it indents it.

```
module LLite

let langParamsTable     = [ "fsharp", ("(*" + "*", "*" + "*)") // The + is not to confuse th
                            "c", ("/**", "**/")
                            "csharp", ("/**", "**/")
                            "java", ("/**", "**/")] |> Map.ofList

let languages = langParamsTable |> Map.fold (fun state lang _ -> state + lang + " ") ""

let usage   = sprintf @"
Usage: llite inputFile parameters
where:
One of the following two sets of parameters is mandatory
    -no string : string opening a narrative comment
    -nc string : string closing a narrative comment
or
    -l language: where language is one of (%s)

One of the following two sets of parameters is mandatory
    -co string : string opening a code block
    -cc string : string closing a code block
```

```
or
    -indent N  : indent the code by N whitespaces

The following parameters are optional:
    -o outFile : defaults to the input file name with mkd extension" languages


let getLangNoNC lang    =
    match Map.tryFind lang langParamsTable with
    | Some(no, nc) -> no, nc
    | None -> failwith (lang + " is not a valid programming language")
```

1.5 Programming Languages limitations

One of the main tenets of literate programming is that the code should be written in the order that facilitates exposition to a human reader, not in the order that makes the compiler happy. This is very important.

If you have written a blog post or tried to explain a codebase to a new joiner, you must have noticed that you don't start from the top of the file and go down, but jump here and there trying to better explain the main concepts. Literate programming says that you should write your code the same way. But in our version of it, the compiler needs to be kept happy because the literate file *is* the code file.

Some ingenuity is required to achieve such goal:

- In C and C++ you can forward declare functions and classes, also class members can be in any order
- In C#, Java, VB.NET, F# (the object oriented part) you can write class members in any order
- In the functional part of F# you do have a problem (see later in this doc)

The F# trick below is used in the rest of the program. You'll understand its usage naturally by just reading the code

```
let declare<'a>  = ref Unchecked.defaultof<'a>
```

2 Implementation

At the core, this program is a simple translator that takes some code text and return a valid markdown/whatever text. We need to know:

- The strings that start and end a narrative comment (input symbols)
- How to translate a code block into a document. We support these variations:
  - Indented: indent them by N spaces
  - Surrounded by startCode/endCode strings

```
type CodeSymbols =
    | Indent of int                 // indentation level in whitespaces
    | Surrounded of string * string // start code * end code

type Options = {
    startNarrative  : string
    endNarrative    : string
    codeSymbols     : CodeSymbols
}

let translate   = declare<Options -> string -> string>
```

2.1 Going over the parse tree

We need a function that takes a string and returns a list with the various blocks. We can then go over each block, perform some operations and, in the end, transform it back to text

```
type Block =
| Code      of string
| Narrative of string

let blockize = declare<Options -> string -> Block list>
```

I could have used regular expressions to parse the program, but it seemed ugly. I could also have used FsParsec, but that brings with it an additional dll. So I decided to roll my own parser. This has several problems:

- It is probably very slow
- It doesn't allow narrative comments inside comments, in particular it doesn't allow the opening comment
- It doesn't allow opening comments in the program code (not even inside a string)

The latter in particular is troublesome. You'll need to use a trick in the code (i.e. concatenating strings) to foul this program in not seeing an opening comment, but it is inconvenient.

With all of that, it works.

TODO: consider switching to FsParsec

2.1.1 Lexer

The lexer is going to process list of characters. We need functions to check if a list of characters starts with certain chars and to return the remaining list after having removed such chars.

BTW: these functions are polymorphic and tail recursive

```
let rec startWith startItems listToCheck =
    match startItems, listToCheck with
    | [], _             -> true
    | _ , []            -> false
    | h1::t1, h2::t2  when h1 = h2  -> startWith t1 t2
    | _, _              -> false

let rec remove itemsToRemove listToModify =
    match itemsToRemove, listToModify with
    | [], l             -> l
    | _ , []            -> failwith "Remove not defined on an empty list"
    | h1::t1, h2::t2  when h1 = h2  -> remove t1 t2
    | _, _              -> failwith "itemsToRemove are not in the list"

let isOpening options       = startWith (List.ofSeq options.startNarrative)
let isClosing options       = startWith (List.ofSeq options.endNarrative)
let remainingOpen options   = remove (List.ofSeq options.startNarrative)
let remainingClose options  = remove (List.ofSeq options.endNarrative)
```

This is a pretty basic tokenizer. It just analyzes the start of the text and returns what it finds. It also keeps track of the line number for the sake of reporting it in the error message.

```
let NL = System.Environment.NewLine

type Token =
| OpenComment   of int
| CloseComment  of int
| Text          of string
```

```
let tokenize options source =

    let startWithNL = startWith (Seq.toList NL)

    let rec text line acc = function
        | t when isOpening options t    -> line, acc, t
        | t when isClosing options t    -> line, acc, t
        | c :: t as full                ->
            let line' = if startWithNL full then line + 1 else line
            text line' (acc + c.ToString()) t
        | []                            -> line, acc, []
    let rec tokenize' line acc = function
        | []                            -> List.rev acc
        | t when isOpening options t    -> tokenize' line
                                               (OpenComment(line)::acc)  (remainingOpen options
        | t when isClosing options t    -> tokenize' line
                                               (CloseComment(line)::acc) (remainingClose optio
        | t                             ->
            let line, s, t'= text line "" t
            tokenize' line (Text(s) :: acc) t'

    tokenize' 1 [] (List.ofSeq source)
```

2.1.2 Parser

The parse tree is just a list of Chunks, where a chunk can be a piece of narrative or a piece of code.

```
type Chunk =
| NarrativeChunk    of Token list
| CodeChunk         of Token list

let parse options source =

    let rec parseNarrative acc = function
        | OpenComment(l)::t         ->
            failwith ("Don't open narrative comments inside narrative comments at line "
                                                                                    + l.ToSt
        | CloseComment(_)::t        -> acc, t
        | Text(s)::t                -> parseNarrative (Text(s)::acc) t
        | []                        -> failwith "You haven't closed your last narrative comm

    let rec parseCode acc = function
        | OpenComment(_)::t as t'   -> acc, t'
        | CloseComment(l)::t        -> parseCode (CloseComment(l)::acc) t
```

```
        | Text(s)::t                -> parseCode (Text(s)::acc) t
        | []                        -> acc, []
    let rec parse' acc = function
        | OpenComment(_)::t         ->
            let narrative, t' = parseNarrative [] t
            parse' (NarrativeChunk(narrative)::acc) t'
        | Text(s)::t                ->
            let code, t' = parseCode [Text(s)] t
            parse' (CodeChunk(code)::acc) t'
        | CloseComment(l)::t           ->
            failwith ("Don't insert a close narrative comment at the start of your program a
                                                                                    + l.ToSt
        | []                -> List.rev acc

    parse' [] (List.ofSeq source)
```

2.1.3 Flattener

The flattening part of the algorithm is a bit unusual. At this point we have a parse tree that contains tokens, but we want to reduce it to two simple node types containing all the text in string form.

TODO: consider managing nested comments and comments in strings (the latter has to happen in earlier phases)

```
let flatten options chunks =
    let tokenToStringNarrative = function
    | OpenComment(l) | CloseComment(l)  -> failwith ("Narrative comments cannot be nested at
                                                                                    + l.ToSt
    | Text(s)                           -> s

    let tokenToStringCode = function
    | OpenComment(l)                -> failwith ("Open narrative comment cannot be in code a
                                                                    + l.ToString()) +
                                                ". Perhaps you have an open comment in" +
                                                " a code string before this comment tag?"
    | CloseComment(_)               -> string(options.endNarrative |> Seq.toArray)
    | Text(s)                       -> s

    let flattenChunk = function
    | NarrativeChunk(tokens)             ->
        Narrative(tokens |> List.fold (fun state token -> state + tokenToStringNarrative tok
    | CodeChunk(tokens)                  ->
        Code(tokens |> List.fold (fun state token -> state + tokenToStringCode token) "")

    chunks |> List.fold (fun state chunk -> flattenChunk chunk :: state) [] |> List.rev
```

We are getting there, now we have a list of blocks we can operate upon

```
blockize := fun options source -> source |> tokenize options |> parse options |> flatten opt
```

2.2 Narrative comments phases

Each phase is a function that takes the options and a block list and returns a block list that has been 'processed' in some way.

```
type Phase = Options -> Block List -> Block List

let removeEmptyBlocks   = declare<Phase>
let mergeBlocks         = declare<Phase>
let addCodeTags         = declare<Phase>

let processPhases options blockList =
    blockList
    |> !removeEmptyBlocks   options
    |> !mergeBlocks         options
    |> !addCodeTags         options
```

We want to manage how many newlines there are between different blocks, because we don't trust the programmer to have a good view of how many newline to keep from comment blocks and code blocks. We'll trim all newlines from the start and end of a block, and then add our own.

```
let newLines = [|'\n';'\r'|]

type System.String with
    member s.TrimNl () = s.Trim(newLines)
```

2.2.1 Remove empty blocks

There might be empty blocks (i.e. between two consecutive comment blocks) in the file. For the sake of formatting the file beautifully, we want to remove them.

```
let extract = function
    | Code(text)        -> text
    | Narrative(text)   -> text

removeEmptyBlocks := fun options blocks ->
                        blocks |> List.filter (fun b -> (extract b).TrimNl().Trim() <> "")
```

2.2.2 Merge blocks

Consecutive blocks of the same kind need to be merged, for the sake of formatting the overall text correctly.

TODO: make tail recursive

```
let rec mergeBlockList = function
    | []        -> []
    | [a]       -> [a]
    | h1::h2::t -> match h1, h2 with
                   | Code(t1), Code(t2)             -> mergeBlockList (Code(t1 + NL + t2)::t
                   | Narrative(n1), Narrative(n2)   -> mergeBlockList(Narrative(n1 + NL + n2
                   | _, _                           -> h1::mergeBlockList(h2::t)

mergeBlocks := fun options blocks -> mergeBlockList blocks
```

2.2.3 Adding code tags

Each code block needs a tag at the start and one at the end or it needs to be indented by N chars.

```
let indent n (s:string) =
    let pad = String.replicate n " "
    pad + s.Replace(NL, NL + pad)

addCodeTags := fun options blocks ->
    match options.codeSymbols with
    | Indent(n)         ->
        blocks |> List.map (function Narrative(s) as nar -> nar | Code(s) -> Code(indent n s
    | Surrounded(s, e)  ->
        blocks |> List.map (function
                            | Narrative(text)   -> Narrative(NL + text.TrimNl() + NL)
                            | Code(text)        -> Code(NL + s + NL + text.TrimNl() + NL + e
```

2.2.4 Flatten again

Once we have the array of blocks, we need to flatten them (transform them in a single string), which is trivial, and then finally implement our original translate function.

```
let sumBlock s b2 = s + extract b2

let flattenB blocks = (blocks |> List.fold sumBlock "").TrimStart(newLines)

translate := fun options text -> text |> !blockize options |> processPhases options |> flatt
```

2.3 Parsing command line arguments

Parsing command lines involves writing a function that goes from a sequence of strings to an input file name, output file name and Options record

```
let parseCommandLine = declare<string array -> string * string * Options>
```

To implement it, we are going to use a command line parser taken from here. The parseArgs function takes a sequence of argument values and map them into a (name,value) tuple. It scans the tuple sequence and put command name into all subsequent tuples without name and discard the initial (“”,“”) tuple. It then groups tuples by name and converts the tuple sequence into a map of (name,value seq)

For now, I don’t need the ‘value seq’ part as all my parameters take a single argument, but I left it in there in case I will need to pass multiple args later on.

```
open  System.Text.RegularExpressions

let (|Command|_|) (s:string) =
  let r = new Regex(@"^(?:-{1,2}|\/)(?<command>\w+)[=:]*(?<value>.*)$",RegexOptions.IgnoreCa
  let m = r.Match(s)
  if m.Success
  then
    Some(m.Groups.["command"].Value.ToLower(), m.Groups.["value"].Value)
  else
    None


let parseArgs (args:string seq) =
  args
  |> Seq.map (fun i ->
                    match i with
                    | Command (n,v) -> (n,v) // command
                    | _ -> ("",i)            // data
                  )
  |> Seq.scan (fun (sn,_) (n,v) -> if n.Length>0 then (n,v) else (sn,v)) ("","")
  |> Seq.skip 1
  |> Seq.groupBy (fun (n,_) -> n)
  |> Seq.map (fun (n,s) -> (n, s |> Seq.map (fun (_,v) -> v) |> Seq.filter (fun i -> i.Lengt
  |> Map.ofSeq

let paramRetrieve (m:Map<string,_>) (p:string) =
  if Map.containsKey p m
  then Some(m.[p])
  else None
```

This is the main logic of parameter passing. Note that we give precedence to the -l and -indent parameters, if present.

This is a function that goes from the map of command line parameters to the input file name, output file name and options. With that we can finally define the original parseCommandLine.

```
let safeHead errMsg s = if s |> Seq.isEmpty then failwith errMsg else s |> Seq.head

let paramsToInputs paramsMap =
    let single p er     = match paramRetrieve paramsMap p with | Some(k) -> Some(k |> safeHe
                                                               | None -> None
    let get p s         = match paramRetrieve paramsMap p with |Some(k) -> k |> safeHead s
                                                               | None -> failwith s
    let defaultP p q er = match paramRetrieve paramsMap p with | Some(k) -> k |> safeHead er
                                                               | None -> q

    let inputFile       = get "" "You need to pass an input file"
    let outputFile      = defaultP  "o"
                                    (System.IO.Path.GetFileNameWithoutExtension(inputFile) +
                                    "You must pass a parameter to -o"

    let no, nc          = match single "l" "You must pass a language parameter to -l" with
                          | Some(l) -> getLangNoNC l
                          | None    ->
                                get "no" "The no (narrative open) parameter is mandatory, if
                                get "nc" "The nc (narrative close) parameter is mandatory, i

    let codeSymbs       = match single "indent" "You must pass a whitespace indentation numb
                          | Some(n)     ->
                                let success, value = System.Int32.TryParse n
                                if success
                                    then Indent(value)
                                    else failwith "-i accepts just an integer value as param
                          | None        ->
                                Surrounded(
                                    get "co" "The co (code open) parameter is mandatory, if
                                    get "cc" "The cc (code close) parameter is mandatory")
    inputFile, outputFile, {
        startNarrative  = no
        endNarrative    = nc
        codeSymbols     = codeSymbs
        }

parseCommandLine := parseArgs >> paramsToInputs
```

2.4 Main method

We can then write main as the only side effect function in the program. Here is where the IO monad would live ...

```
let banner  = "LLite : language friendly literate programming\n"

let myMain args =
    try
        printfn "%s" banner

        let inputFile, outputFile, options = !parseCommandLine args
        let input       = System.IO.File.ReadAllText inputFile
        let output      = !translate options input
        System.IO.File.WriteAllText (outputFile, output)
        0
    with
    | e ->
        printfn "%s" "Failure"
        printfn "%s" e.Message
        printfn "%s" usage
#if DEBUG
        printfn "\nDetailed Error Below:\n%A" e
#endif
        -1
```

3 An aside: forward declaring functions in F#

3.1 A simple solution

You can achieve something somehow similar to forward declaration by the 'declare 'dirty trick used in this program. Whenever you want to do a forward declaration of a function , or variable, you can type:

```
let testDeclare() =

    let add = declare<float -> float>

    let ``function where I want to use add without having defined it`` nums = nums |> Seq.ma
```

This creates a ref to a function from float to float. It looks a bit like an Haskell type declaration. You can then use such function as if it were actually define and delay his definition to a later point in time when you are ready to explain it.

When you are ready to talk about it, you can then define it with:

```
add := fun x -> x + 1.
```

The syntax is not too bad. You get that often-sought Haskell like explicit type declaration and you can regex the codebase to create an index at the end of the program (maybe).

But is it too slow? After all, there is one more indirection call for each function call.

Let's test it: enable #time in F# interactive and execute timeNormalF and timeIndirectF varying sleepTime and howManyIter until you convince yourself that it is ok (or not).

```
let sleepTime   = 50
let howManyIter = 100
let normalF x   = System.Threading.Thread.Sleep sleepTime
let indirectF   = declare<int -> unit>
indirectF      := fun x -> System.Threading.Thread.Sleep sleepTime

let timeNormalF     = [1..howManyIter] |> List.iter normalF
let timeIndirectF   = [1..howManyIter] |> List.iter !indirectF
()
```

3.2 A correct solution (but ugly)

Unfortunately, there is a big problem with all of the above: it doesn't work with generic functions and curried function invocations. The code below works in all cases, but it is ugly for the user to use. In this program I've used the beautiful, but incorrect, syntax.

```
type Literate() =
    static member Declare<'a, 'b>  (ref : obj ref) (x : 'a) : 'b =
        unbox <| (unbox<obj -> obj> !ref) x
    static member Define<'a, 'b> (func : 'a -> 'b) (ref : obj ref) (f : 'a -> 'b) =
        ref := box (unbox<'a> >> f >> box)

// Declaration
let rec id (x : 'a) : 'a = Literate.Declare idImpl x
and idImpl = ref null

// Usage
let f () = id 100 + id 200

// Definition
Literate.Define id idImpl (fun x -> x)
```

3.3 The traditional way

The traditional way of doing something like this is by passing the function as a parameter in a manner similar to the below.

```
// Declaration and usage intermingled
let calculate' (add: int -> int -> int) x y = add x y * add x y

// Definition
let add x y = x + y

let calculate = calculate' add
```

To my way of seeing, this is the most cumbersome solution and conceptually dishonest because you are not parametrizing your function for technical reasons, but just for the sake of explaining things. In a way, you are changing the signature of your functions for the sake of writing a book. That can't be right ...