```
(*
   "And whatever you do, whether in word or deed,
   do it all in the name of the Lord Jesus, giving thanks to
   God the Father through him."
   Collossians 3:17 NIV

   !!! Warning !!! This document is currently in a draft status.
                   it might change a bit.  If you actually do cite it,
                   be sure and let everyone know that you are citing
                   a draft document.

   Title     : Recursive Decendent Parser in F#
   Filename  : N/A (could easly be main.fs)
   Author    : Shawn Eary
   Date      : 15-JUN-2010
   Revision  : dft-1.00
   Copyright : 2010
   License   : Free Christian Document License (FCDL)
               http://sites.google.com/site/shawneary/fcdl
   Warranty  : None (See FCDL Terms)
   Purpose   : To serve as a potential tutorial for the new
               F# language and to recognize that all "good"
               things come from the father in heaven.

   Notes: The program is covered under the FCDL (see above) as such it
          is treated more as a paper than a program.  Do not confuse
          the FCDL License with other "Open Source" Licenses as this
          license is very different.  The FCDL is a "good faith"
          liecnes that basically gives you the
          right to:
          a) Make as many copies of covered documents that you want as
             long as the entire covered documents including spelling
             errors, biblical references, citations and author credits
             are kept intact.
          b) Use portions of this document to create your own
             religiously neutral or Christian products provided you
             properly cite this document.
             (Overall URL of document and Author at tail or beginning
              of work with (filename:functionname) citations inline)
          c) Charge for the distribution of covered documents or
             for products that were created by using portions of covered
             documents with no obligation to reimburse the authors of
             the covered documents.

   Resources
   [1] - "Programming Languages Concepts and Constructs" 2nd Edition
         Ravi Sethi
         Addison-Wesley 1996
         ISBN#0-201-59065-4

   [2] - CS 3721 Programming Languges Recursive Descendent Parsing
         Dr. Neal Wagner
         http://www.cs.utsa.edu/~wagner/CS3723/rdparse/rdparser.html
         (URL Last Checked on 15-JUN-2010)

   [3] - http://en.wikipedia.org/wiki/Operator_associativity
         (URL Last Checked on 15-JUN-2010)

   [4] - http://diditwith.net/2007/10/26/ImReallyDiggingF.aspx
```

```
         (URL Last Checked on 15-JUN-2010)

   [5] - http://stackoverflow.com/questions/398316/f-how-to-chop-a-string-to-
substrings-of-given-length
         (URL Last Checked on 15-JUN-2010)

   [6] - http://cs.hubfs.net/forums/13978/ShowThread.aspx#13978
         (URL Last Checked on 15-JUN-2010)

   [7] - "Compile Contstuction: Principles and Practice"
         Kenneth Louden
         Course Technology; 1 edition (January 24, 1997)
         ISBN-10: 0534939724
         ISBN-13: 978-0534939724
         http://www.amazon.com/Compiler-Construction-Principles-Kenneth-
Louden/dp/0534939724
         (URL Last Checked on 15-JUN-2010)
   *)

(* Enumeration of the tokens that are recogned by this program *)
type tokenType =
    | garbage = 0
    | nothing = 1
    | plus = 2
    | minus = 3
    | mul = 4
    | div = 5
    | integer = 6
    | exp = 7
    | leftParen = 8
    | rightParen = 9

(* A class to hold a token *)
type token(tt : tokenType, ?value : bigint) =
    member x.tokenType with get() = tt
    member x.value with get() = defaultArg value 0I
    override x.ToString() =
        if (x.tokenType = tokenType.integer) then
            x.value.ToString()
        else
            x.tokenType.ToString()

(* This function has been modified some, but I got the basic idea from
   StackOverflow [5] *)
let rec explode str =
    let len = String.length str in
    if len=0 then [] else
    (str.[0] :: explode (str.[1..(len-1)]))

(* Returns true if an integer is on the inputted character list and
   false in all other conditions *)
let rec hasInteger l =
    match l with
    | '~' :: c :: _ -> System.Char.IsDigit(c)
    | c :: _ -> System.Char.IsDigit(c)
    | [] -> false

(* Returns a copy of the inputted character list 'l' with the next
   integer removed from the front of the list *)
```

```
let rec skipBigint l =
    match l with
    | '~' :: rest -> skipBigint(rest)
    | c :: rest ->
    if (System.Char.IsDigit(c)) then skipBigint(rest) else c::rest
    | [] -> []

(* Reads a BigInt from a list of characters.  This customized function
   works better for me than BigInt.Parse becuase it allows me to handle
   the ~ negation operator that is popular in SML/NJ and it also
   supports garbage whereas the BigInt.Parse does not support these
   features

   I got some initial help with the design of this from HubFS [6] *)
let rec readBigPositive (l : char list) (curVal : bigint) =
    match l with
    | [] -> 0I
    | c :: rest ->
    if System.Char.IsDigit(c) then
        let curDigitVal = bigint.Parse(c.ToString())
        let curIncrement = curVal + curDigitVal
        if rest.IsEmpty then
            curIncrement
        else
            if System.Char.IsDigit(rest.Head) then
                readBigPositive rest (curIncrement*10I)
            else
                curIncrement
    else
        0I

(* Wrapper function around readBigPositive for convenience *)
let rec readBigint (l : char list) =
    match l with
    | '~' :: rest -> -1I * (readBigPositive rest 0I)
    | list -> readBigPositive list 0I

(* Takes a list of characters and tries to return a string list of
   token names *)
let rec tokens s =
    match s with
    | '+' :: rest -> token(tokenType.plus) :: tokens(rest)
    | '-' :: rest -> token(tokenType.minus) :: tokens(rest)
    | '*' :: rest -> token(tokenType.mul) :: tokens(rest)
    | '/' :: rest -> token(tokenType.div) :: tokens(rest)
    | '^' :: rest -> token(tokenType.exp) :: tokens(rest)
    | '(' :: rest -> token(tokenType.leftParen) :: tokens(rest)
    | ')' :: rest -> token(tokenType.rightParen) :: tokens(rest)
    | [] -> []
    | l ->
    if hasInteger(l) then
        token(tokenType.integer,readBigint(l)) :: tokens(skipBigint(l)
    )
    else
        if System.Char.IsWhiteSpace(l.Head) then
            tokens(l.Tail)
        else
            token(tokenType.garbage) :: tokens(l)
```

```
(* The parsing routine is based on the Arithmetic Expression Grammar
   on Page 45 of the Sethi Text [1] and the Wagner Tutorials [2].  I
   originally had plans to expand the grammar to allow more interesting
   operations such as complex numbers, but I felt that the operations
   weren't so important when I learned that F# already supports these
   operations via the PowerPack or other options.  Another interesting
   thought is that the F# PowerPack already supports Lex and Yacc which
   bascially makes this whole program an academic expercies, but I felt
   the need to finish it anyway...

   E -> E + T | E - T | T
   T -> T * B | T / B | B
   B -> F ^ B | F
   F -> (E) | integer

   From Above note also that exponentiation is usually right
   associative [2][3]  *)

(* See initial pages of Compiler Contruction Principles and
   Practices by Ken Lauden [7]
   Sorry, I don't have the text with me right now it is at work.
   Mr. Lauden uses functions to recognize bundles of terms.
   For example he might use mulOpp to recognize * and / and AddOpp
   to recognize + and -   I assume his technique is commion practice
   but I put a citation here in case someone is looking for further
   reading *)
let mulOpp (t : token) =
     (t.tokenType = tokenType.mul) ||
     (t.tokenType = tokenType.div)

(* Recognized addition operatiors *)
let addOpp (t : token) =
     (t.tokenType = tokenType.plus) ||
     (t.tokenType = tokenType.minus)

(* Recognized the single exponention operator *)
let expOpp (t : token) =
     (t.tokenType = tokenType.exp)

(* Used to split a list of tokens into two halfs to make it easy
   to execute recursive descendent parsing *)
let rec iSplit
     (l : token list)
     (f : token -> bool)
     (u : token list)
     (parenNestCount : int) =
     match l with
     | []  -> (token(tokenType.nothing), (u, []))
     | _   ->
             (* Define the augmented right list *)
             let arl = l.Head::u
             if l.Head.tokenType = tokenType.leftParen then
                 iSplit l.Tail f arl (parenNestCount + 1)
             else
                 if l.Head.tokenType = tokenType.rightParen then
                     iSplit l.Tail f arl (parenNestCount - 1)
                 else
                     if (f (l.Head)) && (parenNestCount = 0)  then
                        (l.Head, (u, l.Tail))
```

```
                    else
                        iSplit
                            l.Tail
                            f
                            arl
                            parenNestCount

(* Wrapper function around iSplit to be used with right associative
   productions like B -> F ^ B *)
let splitRightAssociative (l : token list) (f : token -> bool) =
    let splitRes = iSplit l f [] 0
    let splitResToken = fst splitRes
    let splitResPair = snd splitRes
    (splitResToken, (List.rev(fst splitResPair), (snd splitResPair)))

(* Wrapper function around iSplit to be used with left associative
   productions like E -> E + T | E - T | T *)
let splitLeftAssociative (l : token list) (f : token -> bool) =
    let splitRes = iSplit (List.rev l) f [] 0
    let splitResToken = fst splitRes
    let splitResPair = snd splitRes
    (splitResToken, (List.rev(snd splitResPair), (fst splitResPair)))

(* "Basically" multiple integer n by itself e times *)
let rec pow (n : bigint) (e : bigint) =
    if e = 0I then
        1I
    else
        if e = 1I then
            n
        else
            n * (pow n (e - 1I))

(* Definition of the mutually recursive functions evalExpression,
   evalTerm, evalBase, evalFactor *)
let rec
    (* Evaluates the production "E -> E + T | E - T | T" *)
    evalExpression (l : token list) =
    let theSplit = splitLeftAssociative l addOpp
    let theToken = fst theSplit
    let splitPair = snd theSplit
    if theToken.tokenType = tokenType.nothing then
        // Did not find an addOpp; therefore this expression must be
        // a term
        evalTerm l
    else
        let left = fst splitPair
        let right = snd splitPair
        if theToken.tokenType = tokenType.plus then
            (evalExpression left) + (evalTerm right)
        else
            (evalExpression left) - (evalTerm right)
and
    (* Evaluates the production "F -> (E) | integer" *)
    evalFactor (l : token list) =
    if l.Head.tokenType = tokenType.leftParen then
        let lRev = List.rev l
        if lRev.Head.tokenType <> tokenType.rightParen then
            (* The factor is not valid since the leftParen was not
```

```
                matched with a rightParen *)
            failwith "Unmatched parenthesis"
        else
            (* Remove the leading and trailing parenthesis from the
               factor.  This can be done by:
               a) taking the tail of the reversed list
               b) reversing it back to the original order
                  and
               c) taking the tail or the original order list *)
            let lChop = (List.rev lRev.Tail).Tail
            evalExpression lChop
    else
        (* The head of this factor is than a leftParen; therefore, it
           has to be an integer *)
        if l.Head.tokenType = tokenType.integer then
            (* The factor is an integer.  Simply return the integer's
               value *)
            l.Head.value
        else
            (* The factor doesn't start with a leftParen and it isn't
               and integer; therefore, the factor is erroneous *)
            (failwith
            ("Invalid Token Found " + l.Head.tokenType.ToString()))
and
    (* Evaluates the production "B -> F ^ B | F" *)
    evalBase (l : token list) =
    let theSplit = splitRightAssociative l expOpp
    let theToken = fst theSplit
    let splitPair = snd theSplit
    if theToken.tokenType = tokenType.nothing then
        (* Did not find an expOpp; therefore,  this expression must
           be a factor *)
        evalFactor l
    else
        let left = fst splitPair
        let right = snd splitPair
        pow (evalFactor left) (evalBase right)
and
    (* Evaluates the production "T -> T * B | T / B | B" *)
    evalTerm (l : token list) =
    let theSplit = splitLeftAssociative l mulOpp
    let theToken = fst theSplit
    let splitPair = snd theSplit
    if theToken.tokenType = tokenType.nothing then
        // Did not find an mulOpp; therefore this expression must be
        // the subject of exponentiation
        evalBase l
    else
        let left = fst splitPair
        let right = snd splitPair
        if theToken.tokenType = tokenType.mul then
            (evalTerm left) * (evalBase right)
        else
            (evalTerm left) / (evalBase right)

(* Beginning prompt of program with instructions for exit *)
printfn "Grammar:"
printfn "E -> E + T | E - T | T"
printfn "T -> T * B | T / B | B"
```

```
printfn "B -> F ^ B | F"
printfn "F -> (E) | integer\n"
printfn "Floating Point values are not allowed and Division truncates."
printfn "Use ~ to represent negative integers.\n"
printfn "Example:"
printfn "Input Expression >>\n"
printfn "~3 + 90 / 2"
printfn "Press [Enter] on a blank line to terminate.\n"

(* Use of a mutable datatype to process lines of user input.  In
   general, it is best to avoid mutable data, but in this particular
   case it makes sense to make "e" mutable *)
let mutable e = "anything" (* e must be anything other than "" *)
while e <> "" do
    printfn "Input Expression >>\n"
    e <- System.Console.ReadLine()
    (* For some weird reasons bigints have to be output in
       %A format...
       http://diditwith.net/2007/10/26/ImReallyDiggingF.aspx *)
    if e <> "" then
        printfn "ans = %A " (evalExpression (tokens (explode e)))
    else
        printfn "exiting"
```