1 Language Details

Minimal F# used in ??

```
(* Whitespace *)
whitespace = ' ' {' '}
newline = '\n' | '\r' '\n'
whitespace-or-newline = whitespace | newline

(* Literal *)
(* Literal: Digit *)
dDigit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
bDigit = "0" | "1"
oDigit  = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7"
xDigit  =
  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
  | "A" | "B" | "C" | "D" | "E" | "F" | "a" | "b" | "c" | "d" | "e" |
  "f"

(* Literal: Integer *)
int = dInt | xint
sbyte = (dInt | xInt) "y"
byte = ((dInt | xInt) "uy")
int32 = (dInt | xInt) ["l"]
uint32 = (dInt | xInt) ("u" | "ul")

dInt = dDigit {dDigit}
bitInt = "0" ("b" | "B") bDigit {bDigit}
octInt = "0" ("o" | "O") oDigit {oDigit}
hexInt = "0" ("x" | "X") xDigit {xDigit}
xint = bitInt | octInt | hexInt

(* Literal: float *)
float = dFloat | sFloat
dFloat = dInt "." {dDigit}
sFloat = (dInt | dFloat) ("e" | "E" ) ["+" | "-"] dInt
ieee64 = float | (xInt "LF")

(* Literal: char *)
char = "'" codePoint | escapeChar "'"
escapeChar =
  "\" ("b" | "n" | "r" | "t" | "\" | '"' | "'" | "a" | "f" | "v")
  | "\u" xDigit xDigit xDigit xDigit
  | "\U" xDigit xDigit xDigit xDigit xDigit xDigit xDigit xDigit
  | "\" dDigit dDigit dDigit

(* Literal: String *)
string = '"' { stringChar }  '"'
stringChar = char - '"'
verbatim-string = '@"' {(char - ('"' | '\"' )) | '""'} '"' |

(* Constant *)
const :=
  byte
  | sbyte
  | int32
  | uint32
  | int
  | ieee64
  | char
  | string
  | verbatim-string
  | "false"
  | "true"
  | "()"

(* Operators *)
infixOrPrefixOp := "+" | "-" | "+. " | "-. " | "%" | "&" | "&&"
tildes = "~" | "~" tildes
prefixOp = infixOrPrefixOp | tildes | (! {opChar} - "!=")
dots = "." | "." dots
infixOp =
  {dots} (
    infixOrPrefixOp
    | "-" {opChar}
    | "+" {opChar}
    | "||"
    | "<" {opChar}
    | ">" {opChar}
    | "="
    | " |" {opChar}
    | "&" {opChar}
    | "^" {opChar}
    | "*" {opChar}
    | "/" {opChar}
    | "%" {opChar}
    | "!=" )
  | ":=" | "::" | "$" | "?" (*$*)

(* Identifier *)
ident = (letter | "_") {letter | dDigit | specialChar}
letter =
  "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" |
   "L" | "M"
  | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "X" | "Y" |
    "Z"
  | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" |
   "l" | "m"
  | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "x" | "y" |
   "z"
specialChar = "_"

long-ident = ident | ident '.' long-ident  (* no space around '.' *)
long-ident-or-op = [long-ident '.'] ident-or-op (* no space around
   '.' *)
ident-or-op =
  ident
  | "(" infixOp | prefixOp ")"
  | "(*)"

(* Keywords *)
ident-keyword =
  "abstract" | "and" | "as" | "assert" | "base" | "begin" | "class" |
  "default"
  | "delegate" | "do" | "done" | "downcast" | "downto" | "elif" |
  "else" | "end"
  | "exception" | "extern" | "false" | "finally" | "for" | "fun" |
  "function"
  | "global" | "if" | "in" | "inherit" | "inline" | "interface" |
  "internal"
  | "lazy" | "let" | "match" | "member" | "module" | "mutable"
  | "namespace" | "new" | "null" | "of" | "open" | "or" | "override"
  | "private"
  | "public" | "rec" | "return" | "sig" | "static" | "struct" |
  "then" | "to"
  | "true" | "try" | "type" | "upcast" | "use" | "val" | "void" |
  "when"
  | "while" | "with" | "yield"

reserved-ident-keyword =
  "atomic" | "break" | "checked" | "component" | "const" |
  "constraint"
  | "constructor" | "continue" | "eager" | "fixed" | "fori" |
  "functor"
  | "include" "measure" | "method" | "mixin" | "object" | "parallel"
  | "params" | "process" | "protected" | "pure" | "recursive" |
  "sealed"
  | "tailcall" | "trait" | "virtual" | "volatile"

reserved-ident-formats = ident-text ( '!' | '#')

(* Symbolic Keywords *)
symbolic-keyword =
  "let!" | "use!" | "do!" | "yield!" | "return!" | "|" | "->" | "<-"
  | "." | ":"
  | "(" | ")" | "[" | "]" | "[<" | ">]" | "[|" | "|]" | "{" | "}" |
  "'" | "#"
  | ":?>" | ":?" | ":>" | ".." | "::" | ":=" | ";;" | ";" | "=" | "_"
  | "?"
  | "??" | "(*)" | "<@" | "@>" | "<@@" | "@@>"
reserved-symbolic-sequence =  "~" | "'"

(* Comments *)
blockComment = "(*" {codePoint} "*)"
lineComment = "//" {codePoint - newline} newline

(* Expressions *)
expr =
  | const (* a const value *)
  | "(" expr ")" (* block *)
  | long-ident-or-op (* identifier or operator *)
  | expr '.' long-ident-or-op (* dot lookup expression, no space
  around '.' *)
  | expr expr (* application *)
  | expr infix-op expr (* infix application *)
  | prefix-op expr (* prefix application *)
  | expr ".[" expr "]" (* index lookup, no space before '.' *)
  | expr ".[" slice-range "]" (* index lookup *)
  | expr "<-" expr (* assingment *)
  | exprTuple (* tuple *)
  | "[" (exprSeq | range-expr) "]" (* list *)
  | "[|" (exprSeq | range-expr) "|]" (* array *)
  | expr ":" type (* type annotation *)
  | expr; expr (* sequence of expressions *)
  | "let" valueDefn "in" expr  (* binding a value or variable *)
  | "let" ["rec"] functionDefn "in" expr (* binding a function or
  operator *)
  | "if" expr "then" expr {"elif" expr "then" expr} ["else" expr]  (*
  conditional *)
  | "while" expr "do" expr ["done"] (* while *)

exprTuple = expr | expr "," exprTuple
exprSeq =  expr | expr ";" exprSeq
range-expr = expr ".." expr [".." expr]
slice-range =
  expr
  | expr ".."  (* no space between expr and ".." *)
  | ".." expr  (* no space between expr and ".." *)
  | expr ".." expr  (* no space between expr and ".." *)
  | '*'

(* Types *)
type =
  | long-ident (* named such as "int" *)
  | "(" type ")" (* paranthesized *)
  | type "->" type (* function *)
  | typeTuple  (* tuple *)
  | typar (* variable *)
  | type long-ident (* named such as "int list" *)
  | type "[" typeArray "]" (* array, no spaces *)
typeTuple = type | type "*" typeTuple
typeArray = "," | "," typeArray

(* Pattern *)
pat =
  const (* constant *)
  | "_" (* wildcard *)
  | ident (* named *)
  | pat ":" type (* type constraint *)
  | "(" pat ")" (* paranthesized *)
  | patTuple (* tuple *)
  | patList (* list *)
  | patArray (* array *)

patTuple = pat | pat "," patTuple
patList := "[" [patSeq] "]"
patArray := "[|" [patSeq] "|]"
patSeq = pat | pat ";" patSeq

(* Value binding *)
valueDefn = ["mutable "] pat "=" expr

(* Function binding *)
functionDefn = ident-or-op argument-pats [":" type] "=" expr "
argument-pats = pat | pat argument-pats
```

Operator	Associativity	Description
`ident "<" types ">"`	Left	High-precedence type application
`ident "(" expr ")"`	Left	High-predence application
`"."`	Left	
`prefixOp`	Left	All prefix operators
`""` rule|	Left	Pattern matching rule
`ident expr`, `"lazy'' expr`, `"assert'' epxr`	Left	
`"**" opChar`	Right	Exponent like
`"*" opChar`, `"/" opChar`, `"%" opChar`	Left	Infix multiplication like
`"-" opChar`, `"+" opChar`	Left	Infix addition like
`":?''`	None	
`"::''`	Right	
`"^'' opChar`	Right	
`"!=" opChar`, `"<" opChar`, `">" opChar`, `"="`, `"|" opChar`, `"&" opChar`, `"$" opChar`	Left	Infix addition like
`":>"`, `":?>"`	Right	
`"&"`, `"&&"`	Left	Boolean and like
`"or"`, `"||"`	Left	Boolean or like
`","`	None	
`":="`	Right	
`"->"`	Right	
`"if"`	None	
`"function"`, `"fun"`, `"match"`, `"try"`	None	
`"let"`	None	
`";"`	Right	
`"|"`	Left	
`"when"`	Right	
`"as"`	Right	

Table 1.1: Precedence and associativity of operators. Operators in the same row has same precedence. See ?? for the definition of `prefixOp`