(Basic PL terminology)

- grammar, BNF, (production) rule, non-terminals, terminals
- associativity
- precedence

Getting started

- First steps in VS and VS Code
  - F# project templates, .NET Core CLI
- Values and bindings, scoping
  - `let`, `let rec`
- Immutability vs. mutability
- Calling functions, partial function application
- Comments - `//` vs `///`, `(* ... *)`
- Properties, dot notation

Types and values

- Basic types and values/literals:
  - `char` (alias to `Char`): `'a'`, `'b'`, etc.
  - `int` (alias to `Int32`): `1`, `2`, etc.
  - `float` (alias to `Double`): `1.0`, `2.0e2`, etc.
  - `string` (alias to `String`)
    - `"Hello world!"`
    - `"""multi-line ...comment here"""`
    - `"multi-line\n\ comment\n\ .... here"`
  - `bool` (alias to `Boolean`),
  - decimal: `1m`, `2m`
- Records
  - Declaration: `type Person = { FirstName: string; LastName: string }`
  - Record values
    - `let john = { FirstName="John"; LastName="Smith" }`
    - `let mary = { john with FirstName="Mary" }`
- **Tuples**
  - (1, 2.0, "3.00")
- **Discriminated unions**
- Aggregate data types (collections):
  - `list` (immutable lists)
    - Use `ResizeArray` for C# lists
  - `array` (doc, alias to `Array`)
  - `array2D` (alias to `Array2D`)
    - Slicing arrays
      - `matrix[*,*]` - wildcard
      - `matrix[*, 2..4]` - lower/upper index (**Note:** indices are zero-based)
      - `matrix[*, ..4]` - upper

        - `matrix[*, 2..]` - lower
    - `set` (alias to `Set`)

Type signatures

- `int`: type
- `'T`: type variable
- `'T list` or `List<'T>`: lists/collections
- `'T1 * 'T2`: product type (tuple)
- `'T1 -> 'T2`: function type (lambda = anonymous function), `->` is right associative:
    - `'T1 -> 'T2 -> 'T3` <==> `'T -> ('T2 -> 'T3)`

Functional programming basics

- Functions as first-class values
- Currying vs tupled functions
    - `let add x y = x+y`
    - `let Add (x, y) = x+y`
- Partial application
    - `let add5 y = add 5 y` <==> `let add5 = add 5` (Eta-reduction)
- Equivalent forms for `let foo x y = ...`
    - `let foo = fun x y -> x+y`
    - `let foo = fun x -> fun y -> x+y`
    - `let foo x = fun y -> x+y`
    - `let foo x y = x+y`
- Higher-order functions
    - Functions taking other functions as arguments or returning them as values
- Aggregate operators
    - `map`
        - `List.map (fun i -> i*i) [1;2;3;4]`
    - `iter`
        - `List.iter (fun i -> printfn "%d") [1;2;3;4]`
        - `List.iter (printfn "%d") [1;2;3;4]`
    - `fold`
        - `List.fold (fun acc i -> acc+i) 0 [1;2;3;4]`