## 基本

### 基本

#### ・*コメント*

// 一行コメント

(\*　複数行コメント　\*)

#### ・エントリーポイント

[<EntryPoint>]

let main hoge = …

・代入は = でなく <-　????

・メリットなど

自前でロジックをゴリゴリ作りこむプロジェクト向き

.NET ライブラリを参照するプロジェクトでは C# の方が楽？

・結合法則性

F# では左側に結合法則性がある

let F x y z = x y z

// ↓

let F x y z = (x y) z

### C# との対応表

|  |  |  |  |
| --- | --- | --- | --- |
| F# | C# | 意味 | 備考 |
| open | using | 外部モジュールの参照 | VSのF# では必要時に自動的に追加される |
| match | select case |  | [match](#_Match) |
| \_ | default | match 内で用いる |  |
| elif | else if |  |  |
|  |  |  |  |
| <> | != | bool の非等価演算子 |  |
| \*\* | ^ | べき乗 | long のみ |
| fun | func | ラムダ式 | [ラムダ式](#_ラムダ式)　※似過ぎて紛らわしい |
| not | ! | 論理値の否定 | -(マイナス) でも同じ |
|  |  |  |  |

static class に近いもの

[<AbstractClass; Sealed>]

type SampleClass private() =

static member SampleMethod(arg : string) : string = arg + arg

### 関連する予約語など

let 変数の宣言。　※F# は 変数も関数も let で宣言するらしい

use dispose を呼び出してリソースを解放する必要のある変数を宣言する

try, exception おなじみの例外処理

TRUE, FALSE boolean 型の値だが、F# ではどうやら大文字らしい

yield 繰り返し処理で値を返す。C#でも同機能（ほとんど使わないが）

let! 匿名宣言？

do 関数または値の定義を使用しないでコードを実行

実行するコードは unit を返す

### 注意点

.NET で呼び出す場合 F# 側で List を返す関数がある場合、これは　System.Collections.Generic.List<T>　ではなく、Microsoft.FSharp.Collections.List<T>　なので、そのままでは受け取れない。

→ 配列 (Array) であれば問題ないので、引数、戻り値は Array にする

### その他

C系の言語に慣れていると、各行末の ; がないのは気持ち悪い

()　の有無について

let reader = new System.IO.StringReader("hello");

let line1 = reader.ReadLine

printfn "The line is %s" line1 // →エラーとなる

let line2 = reader.ReadLine()

line1 はReadLine メソッドへのポインタ(delegate) で、line2 はメソッドの戻り値であるstring 型の値となる。

## Interactive

printfn で表示

;; + Enter で実行

## 特徴・独特な機能

### pipeline

|> のような表現の事

普段右から与える引数（変数）を左から与える。ただそれだけの事。

e.g. 1

let doSomething arg = printfn "%s" arg

"hoge" |> doSomething

e.g. 2

let result = [1..10] |> List.map (fun i -> i+1)

なお、こういった事も可能

let (|>) x f = f x

・リバース・パイプ （ <| の事）

引数を右から与える ⇒ なんの為にこんなものが？

括弧なしで引数を与える事ができ、見やすく、時間短縮になる？

printf "%i" 1+2 // error

printf "%i" (1+2) // using parens

printf "%i" <| 1+2 // using reverse pipe

・両側から同時に利用する事も可能　pseudo infix と言う？

let add x y = x + y

(1+2) add (3+4) // error

1+2 |> add <| 3+4 // pseudo infix

### composition

>> のような表現の事。関数合成演算子（ composition operator ）

関数の結果を直接次の関数に渡す

let add1Times2 x = times2(add1 x)

let add1Times2 = (>>) add1 times2

let add1Times2 = add1 >> times2

上３つは同じ意味。一番下が一番見やすい。

入力の引数と出力の引数の型が同じである限り、各関数はどの変数も使用可能。以下のような事も可能

let twice f = f >> f

・パイプと関数合成演算子の違い

|> 隣の変数（引数）をそのまま渡す。

>> 隣の関数の戻り値を隣に渡す。

関数合成演算子は関数にしか使えない。

### match

select case のようなもの

[0..10] |> List.map( fun a ->

match a % 2 with

| 1 -> "奇数"

| 0 -> "偶数"

| \_ -> "その他")

例文の場合、「その他」にはなり得ないが、 \_ は default のようなもの。

function

関数の最後の引数に対してパターンマッチする場合

例）List の要素数を返す count 関数

let rec count = function

| [] -> 0 // 配列の場合

| x::xs -> 1 + (count xs) // Seq の場合

この２つはほぼ同じだとか　：http://zecl.hatenablog.com/entry/20110512/p1

match a with

a |> function

### シャドウイング

特定の変数をそれ以降で参照されないように隠す事

let read path =

(\* 正規化前のpathをログ出力する関数 \*)

let logger () = printfn "%s" path

(\* シャドウィング。これ以降は正規化前のpathを参照する必要なし \*)

let path = normalize path

...

readFile path logger

### rec

再帰する関数を示す

let rec add1\_nTimes n =

if n<=0 then 0

else 1 + add1\_nTimes(n-1)

### mutable

値が変更されうる変数。の意。

通常の言語であれば、ほとんどの変数がそうだと思うが、つまりF#上では基本的に immutable な変数を使って設計をする、と言う事。

### do

C系の言語の {} のようにブロックを表す？

また、EntryPoint で使うと引数や戻り値を省略して書ける

[<EntryPoint>]

do

printfn "hello world"

### module

C# 的に書くと public static class といった感じ

### するべき事・しないべき事

・mutable　は**一度たりとも**使わない。変数を使わず、関数を用いる。

・forループ、if-then-else　を使わず、list 再帰、 match を使う　seqモジュール？

・.NET 言語などにありがちな、ドット表記を使わない ???

・クラスを使わない。C言語のようなシンプルな記述になる。

・デバッガを使用しない。一行ずつ試し、頭の中でデバッグを行う。（そもそもデバッグツールがない）

・小さな、構造体のような 型をいくつかつくることになる。

・pipe (|>) やcomposition (>>) を多用すべき。入れ子式で関数を呼び出すより

・対話画面によって、一段ずつ開発を進めるべき。

## List

### 基本

配列のようなもの。 List機能（ライブラリ）は F# で非常に重要な位置を占めるとの事。

let integerList = [ 1; 2; 3; 4; 5; 6; 7 ]

* [|1;2;3|] とすると Array (配列) になる

セミコロンはつけなくても可？？ 18.03.01 関数じゃないとか言われてエラーになったが？

let list = [ 1 2 3]

同じ基底クラスを継承した同系クラスであれば同じ list 内に含む事が可能

let myControlList : Control list = [ new Button(); new CheckBox() ]

連番（seq ?） 関数とみなされるらしい??

e.g.

let list = [ 1 .. 10 ]

要素の追加

let list1 = [ 1 2 3]

let list2 = 100 :: list1 => 100; 1; 2; 3;

List変数どうしの加算

let list3 = list1 @list2

要素の参照

let intList = [1;2;3;4;5] in intList.[0]

### プロパティ

型 説明

Head T 最初の要素

Empty T list 適切な型の空のリストを返す？

IsEmpty bool

Item T 指定 index のインスタンスを返す

Length int

Tail T list 最初の要素を除いたリスト

ｌist1.Tail.Head // ２つ目の要素

ｌist1.Tail.Tail.Head // ３つ目の要素

### 関数

・合成、分割

append

let list = List.append [1; 2; 3] [4; 5; 6]

→ val list : int list = [1; 2; 3; 4; 5; 6]

※ listA |> List. Append listB だと　listA は第二引数となり listB: listA となる

concat ‘T list list -> ‘T list など

Array.concat [ [|0..3|] ; [|4|] ]

→ val it : int [] = [|0; 1; 2; 3; 4|]

filter(f:bool, list): list 引数の list をf の条件に合う要素だけのlist にして返す

reduce(‘T[] -> ‘T) 例えばstring[] -> string にする。

partition 一次配列を二次配列に変換

zip(list1, list2) 要素数が同じ 2つのListを結合して多次元 Listを作る

let array1 = [| 1; 2; 3 |]

let array2 = [| -1; -2; -3 |]

let arrayZip = Array.zip array1 array2

→ val arrayZip : (int \* int) [] = [|(1, -1); (2, -2); (3, -3)|]

unzip

・検索、条件

forall(f) : bool 要素の全てが条件に一致するかどうか？

exists (f) : bool 引数となるラムダ式の条件にあうメンバが存在するかどうかを返す、

find(f:bool)

findIndex

tryFind

tryFindIndex

・並び替え

sort (list) : list 引数で与えられた list を並び替えた list を返す

sortBy(f) : list 引数となるラムダ式の条件で並び替える

rev =reverse 逆順で並び替える

・map

map(f, list):list List内のすべての要素に メソッド f を施して新しいlist を返す

e.g.1

let data = [1;2;3;4]

let added1List = data |> List.map (fun x -> x + 1)

e.g.2

let eachAdd1 = List.map (fun i -> i+1)

eachAdd1 [0;1;2;3]

map2(f, list1, list2) ２引数の関数f のそれぞれに List1, List2 を代入した結果を返す

e.g.

let list1 = [1; 2; 3]

let list2 = [4; 5; 6]

let sumList = List.map2 (fun x y -> x + y) list1 list2

→ val sumList : int list = [5; 7; 9]

map3

mapi

・iter

Iter(f):unit 各要素に対し、引数のf関数の実行のみを行う。

・統計、計算

sum

sumBy

average

averageBy

・よくわからない 18.03.06

pick

choose

fold(f(a,b),

foldBack

### Array

List とほぼ同じだが、ユニークなメンバをまとめる

メソッド

create(int) : 新しい配列を生成する

### 参考資料

[msdn：Collections.List モジュール (F#)](https://msdn.microsoft.com/ja-jp/library/ee353738(v=vs.120).aspx)

## 変数

### 型

|  |  |  |  |
| --- | --- | --- | --- |
| 型 | 説明 | 型推測 | 備考 |
| int | 32bit 数値 | b bin, o oct, x hex e.g. 10b |  |
| char | Unicode文字 |  |  |
| string |  |  | 読み取り専用  [System.Text.StringBuilderクラス](http://msdn.microsoft.com/ja-jp/library/system.text.stringbuilder.aspx) |
| byte |  | uy e.g. byte = 10uy |  |
| sbyte |  |  |  |
| float | 64bit 浮動小数 |  | または double ; IEEE 754規格 |
| float32 | 32bit 浮動小数 | f | またはsingle |
| bool |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

### フォーマット文字列

|  |  |
| --- | --- |
| %b | bool |
| %s | string |
| %d, %i | integer |
| %u | integer(as unsigned) |
| %x | hex (using lowercase) |
| %X | hex (using uppercase) |
| %o | oct |
| %e, %E | [-]d.dddde[sign]ddd  科学技術計算で使う 3.14 \* 10^0 っぽいやつ |
| %f | float |
| %g, %G | : f or e format |
| %M | decimal |
| %O | any value by using ToString() |
| %A | any reference |

### ジェネリック型

let onAStick x = x.ToString() + " on a stick"

'a -> string

このアポストロフィーがジェネリック型を示す。

C#風に書くと

string onAStick<a>();

以下のようにして実行する

onAStick 22

onAStick 3.14159

onAStick "hello"

リストや抽象構造体(abstract strucures) を扱う際に非常に重要になる

### その他の型

tuple 型

複数の型の変数を１セットとした型の事

F# に於いては , があったらタプル型とみなして良い？

string \* int // ("hello", 1)

コレクション型

int list // e.g. [1;2;3]

string list // e.g. ["a";"b";"c"]

seq<int> // e.g. seq{1..10}

int [] // e.g. [|1;2;3|]

option 型

独立単位型

レコード型

データベースの行（レコード）のような形

static 静的メンバ

type 型を示す。 C# の type と同じと思われる

val 特定の状況下で値やメンバである事を示す？

## メソッド

### 書き方

funあり：

let inc = fun x -> x + 1

letなし： let 関数名 引数１ 引数２…　= 処理内容

let inc x = x + 1

引数と戻り値の型指定

let stringLengthAsInt (arg:string) :int = arg.Length

※注意！

let stringLength arg:string = arg.Length

→ 戻り値の型を string に指定しているのに int を返す、と認識されるのでエラー

関数の範囲は、**同じインデントライン**で示す。

### 呼び出し

引数はスペースで区切る。

let ret = add 1

### 引数の型指定

推測不可能な場合

let stringLength (x:string) = x.Length

括弧がないと、戻り値の型指定だと思われてしまう。

下の例は引数、戻り値、両方指定している

let stringLength (x:string):in

### 高階関数 (higher-order function ;HOF)

他の関数を引数や戻り値にする関数の事

let evalWith5ThenAdd2 fn = fn 5 + 2

let add1 x = x + 1

evalWith5ThenAdd2 add1

→結果: 8 となる

※注意点　この場合　fn:(int -> int) -> int なので　(float -> int) の関数は使用できない（エラーになる）

その為、型の指定が必要になる場合がある

let evalWith5AsInt (fn:int->int) = fn 5

let evalWith5AsFloat (fn:int->float) = fn 5

map関数

引数1つの関数 f と、list を引数とする

let add1 x = x + 1

List.map add1 [1..10]

### inline

F# では自動汎化により、指定しなければ引数などはジェネリック型 (`a) になるが、add などは型推論で自動的に int になってしまう。

let add a b = a + b;;

val add : a:int -> b:int -> int

↓

> let inline add a b = a + b;;

但しinline の多用はコンパイラの最適化に対応や、ライブラリ内関数の事項が使用できなくなる場合があるので、可能な限り避ける。

また、パフォーマンスも向上するとは限らないので、測定を行うのが無難。

### ignore メソッド

引数としては何でも受け入れ、unit 型を返す特別メソッド。

F#では、通常関数の戻り値を必ず受け取るが、これを使うと関数の戻り値を「捨てる」事ができる。

|  |  |  |  |
| --- | --- | --- | --- |
|  | 通常 | 右向き | 左向き |
| 単発 | ignore(a()) | a() |> ignore | ignore <| a() |
| 連続 | foo(bar(baz())) | () |> baz |> bar |> foo | foo << bar << baz <| () |

do 1+1

上記の例はコンパイルエラーになるので

do (1+1 |> ignore)

とする。

### currying

関数型言語に多大な影響を与えたHaskell Curry（アメリカの数学者）にちなんでこう呼ぶらしい。

//　通常の引数２つの関数

let printTwoParameters x y =

printfn "x=%i y=%i" x y

↓内部ではこのようになる

//　明確な currying 版

let printTwoParameters x =

let subFunction y =

　printfn "x=%i y=%i" x y

subFunction

// このような実行形式となる

let intermediateFn = printTwoParameters 1 //　x が組み込まれた subFunction が返る

intermediateFn 2

→結局は全ての関数が、引数を一つしかとらない数学的関数となる

+ 関数

let x = 6

let y = 99

let intermediateFn = (+) x

let result = intermediateFn y

(+) キーワードが引数を取って加算する事を示している　→結果は 105 になる

### partial function application

関数型言語では広く用いられる手法で、非常に重要。

部分的な機能を実装し、何度も使いまわす事ができる。メソッドとメソッドを掛け合わせて、まるで継承関係のメソッドのような事を継承なしで実装できる。

（例）

let adderWithPluggableLogger logger x y = // 変数を３つとる関数

logger "x" x // この行で 第一引数 logger はstring, ‘a を引数-

logger "y" y // -とするmethod と判定できる→ logger メソッドを実行

let result = x + y // 加算結果を格納する変数 result

logger "x+y" result // logger メソッドを実行

result

// 引数を２つとり、Conlose 画面に{0}={1} 形式で表示するメソッド

let consoleLogger argName argValue =

printfn "%s=%A" argName argValue

// 引数を２つとり、DialogBox　に{0}={1} 形式で表示するメソッド

let popupLogger argName argValue =

let message = sprintf "%s=%A" argName argValue

System.Windows.Forms.MessageBox.Show(

text=message,caption="Logger")

|> ignore

//２つの各引数と、その加算結果をConlose表示する 画面にメソッド

let addWithConsoleLogger = adderWithPluggableLogger consoleLogger

//２つの各引数と、その加算結果をDialogBoxに表示する 画面にメソッド

let addWithPopupLogger = adderWithPluggableLogger popupLogger

partial function application の引数の順番に関するガイドライン

１．変化の可能性の低い変数を早めにもってくる　（原文では static なので誤訳かも）

２．データ型や list ,もしくは一番変化の多い変数を最後にもってくる

３．

1

基本的にすべてラムダ式？

fun 関数である事を示す。省略可能。匿名メソッドの表現

inline →　インライン関数

※inline関数

c言語のマクロのように、関数ではなく呼び出し元で展開して実行するもの

関数の呼び出しや引数の生成にかかる時間を節約するらしい

lazy 処理の結果だけが欲しい時？

rec 関数が何度も呼び出される事をしめす

## class

通常の書き方

type TestClass() = class

new() = {(\* フィールドの初期値を記述 \*)}

let mutable StringMember　= "hoge"

end

匿名クラス的な書き方

type Name = {

stringMember:string;

intMember:int

}

[<Class>]属性

class/end は不要なしで記述できるが、あまり出番はないらしい

primary constructor (default constructor)

type 型名 (引数リスト) =

クラス本体

* C# と違ってコンストラクタも記述しないとインスタンス生成すらできないらしい

### スコープ

let → private になる。要するにprimary constructor で値を代入しっぱなし。

val → public になる。

### Default Value

初期値の指定を省略するアトリビュート。

[<DefaultValue>]

### emun

type Animal =

| Dog = 0

| Elephant = 1

### static

### 継承

### interface

type iAnimal = interface

abstract Bark : unit -> unit

end

type Cat = class

new () = {}

interface iAnimal with

member x.Bark () = printfn "nya- nya-"

end

// ↓ このように個別に実装しないと外から呼び出せない。アホらし過ぎる 18.03.06

// ↓ こちらが優先的に呼ばれるらしいので、 上では意味のない実装にする手も

　member x.Bark () = printfn "nya- nya-"

end

### module

・F# 上で module で組むと C#から呼び出すときに static class として認識されている

・リフレクションで呼び出す時は全て type になる？？？

### 関連する予約語など

abstract 抽象クラス

interface インターフェイス

internal アセンブリ内

downto 下方変換

upcast 上方変換

## ラムダ式

fun を付ける必要がある

e.g. let add = fun x y -> x + y

なお、これは以下の例と全く同じになる

let add x y = x + y

以下の３つも同じ

let adderGenerator1 x y = x + y

let adderGenerator2 x = fun y -> x + y

let adderGenerator3 = fun x -> (fun y -> x + y)

## 冗長構文

(verbose)

code block

begin で始まり end で終わる？

## F#らしくない表現

プログラム言語では共通の概念であっても、F# らしくない為にあまり使わない方がよさそうな表現をまとめる。

・for ループ

[ for I in 1 .. 10 -> 処理 ]

但しこんなこともできるらしい

let squaresList = [ for i in 1 .. 10 -> i \* i ]

## 表示

printf コマンドラインに文字列を表示する

e.g.

printf "Hello World!"

printf "%d" (inc 1)

printfn 改行コード付きでコマンドラインに文字列を表示する

## XAML

F#独特

fixed ガベージコレクションから保護する

## よくわからないもの

match 算出値の branch = ノードの事？

discriminated union

## 参考資料

["関数的"思考を身に着ける（英語）](https://fsharpforfunandprofit.com/series/thinking-functionally.html)

英語で書かれている事を踏まえてもなお、最初に読むべき読み物

[MS公式: F# 言語リファレンス](https://msdn.microsoft.com/ja-jp/library/dd233181.aspx)

[Try F#](http://tryfs.net/) : ブラウザ上で F# コードの実行結果を確認できる

[F#入門](http://fsharpintro.net/)

C#ユーザーの為のリファレンス？

[仕事と F# と私](https://qiita.com/u_1roh/items/eb90672fc40fc7f2ea86)

３DCADとF#について少しだけ触れられている

[七誌の開発Wiki](http://seesaawiki.jp/w/n7shi/d/F%23)

クラス(C#)を基準にF# を解説

* [neue cc - F# TutorialをC#と比較しながらでF#を学ぶ](http://neue.cc/2009/11/09_214.html)

[F#の実例（英語）](http://fssnip.net/)

[標準的な.NETの規約に則ったAPIにする為のガイドライン](https://www.microsoft.com/en-us/research/project/f-at-microsoft-research/?from=http%3A%2F%2Fresearch.microsoft.com%2Fen-us%2Fum%2Fcambridge%2Fprojects%2Ffsharp%2Fmanual%2Ffsharp-component-design-guidelines.pdf)