**〇F#の基本**

・整数 int 0, 123, -456 など

・浮動小数点数 float 0.0, 1.23, -45.6 など

・真理値 bool true, false

・文字列 string "Taro", "Hello\n"など

・型をC言語のように明記しなくてよい、Pythonみたいに勝手に判断してくれる

・関数を使うときは「関数 引数」と書く

sqrt 4.0 → 2.0 (sqrtはintに対応していないので「.0」をつけないといけない)

・型変換は自動でやってくれないので自分で型変換を行う必要がある

4.2 + 2 → エラー

4.2 + float 2 → 6.2

・文字列同士を+演算子で足せる。”abc” + “def” → “abcdef”

・文字列の長さを返す String.length “文字列” → 3

・「// コメント」とすると「//」の後から行末までコメントアウト

・「(\* コメント \*)」とすると間がコメントアウトされる

・「#nowarn "62"」「#light "off"」とプログラムの最初に書くことで冗長構文で認識される

**〇VSCodeの使い方**

・対話型のF#を使うときは「表示」→「その他のウインドウ」→「インタラクティブF#」

・インタラクティブF#で1つの対話を修了させるときは「;;」を最後につける

・対話型（スクリプトファイル）とコンパイル型（コードファイル）の2種類のファイルがある。対話型は「ファイル名.fsx」でコンパイル型は「ファイル名.fs」

・対話型の実行は「#load "fsx ファイルの絶対パスまたは相対パス.fsx";;」とする。パスはファイルを右クリックすればコピーできる

・「open ファイル名;;」→「main;;」で実行される

#load “Program.fsx”;;

open Program;;

main;; # またはプログラムの中で使っている変数名

・プログラムの中の実行したいところを範囲選択して右クリック→「対話形式で実行」するとその部分だけ実行してくれる

**○変数**

・変数定義は「let 変数名 = 値」とする

・C言語のように変数を後から変えることはできない。

let x = 0

x = 1 // この「=」は比較演算子扱いになり、falseになる（x ≠ 1より）

let x = 1 // これは再定義なのでx = 1になる

・変数の再定義によって関数の中の変数は書き換わることはない

let r = 1;;

let area = r \* r;;

area;; // 1 \* 1

let r = 2;;

area;; // 1 \* 1のまま

**○関数**

・関数の定義は「let 関数名 引数名 = 式」

例、半径rを引数として円の面積を返す

let area r = r\*r\*3.14;;

// 関数型 area : r:float -> float (引数 -> 出力)

・関数を使うときは「関数名 引数」とする

area 1.0;; // 3.14

・標準で定義されている関数を次のように書くこともできる「let f = sqrt」

・複数の引数を使う場合は2通りある

1.「let 関数名 引数名1 引数2 = 式」

let average1 x y = (x + y) / 2.0;;

// val average1 : x:float -> y:float -> float

average1 2.0 3.0 // 2.5

2.「let 関数名 (引数名1, 引数名2) = 式」

let average2 (x, y) = (x + y) / 2.0;;

// val average2 : x:float \* y:float -> float

こちらは引数がタプル（組）となっている。使うときも(x, y)の形にする

average2 (2.0, 3.0) // 2.5

・再起関数を定義するときは「let rec 関数名 引数名 = 式」として関数名の前にrecを必ずつけること

例、階乗

let rec f n = if n = 0 then 1 else n \* f (n - 1);;

・関数の引数の型を指定するときは「(引数 : 型)」とする

let rec debt (r : float) d m n =

if n = 0 then m

else int (1.0 + r / 100.0 \* 30.0 / 365.0) \* m – d

・関数の中で新しい変数を定義する「局所定義」ができる。これをすると見やすかったり、同じ計算を複数回することを防いでくれるので計算量が少なくなったりする

「let 関数名 引数 = let 変数 = 式1 in 式2」式2で変数を複数回使うときに有用

let rec newton a n = if n = 0 then a

else let x = newton a (n-1) in (x + a / x) / 2.0;;

・無名関数は「fun 引数名 -> 式」と定義して、関数を引数とする関数の中で使える

let dx = 0.00001;;

let diff f x = (f (x – dx) – f x) / dx;; // 導関数、fが関数引数となっている

let g = diff (fun x -> x \* x);; // diffの第2引数xがgの引数となる

g 3.0;; // f’(x) = (x^2)’ = 2x, f’(3.0) = 6.00001

diff (fun x -> x \* x) 3.0 // 6.00001 上と同じ計算

・無名関数の引数の値が確定しないと「’a」という型になる。これを多相型関数という。このときにそのまま引数に代入しようとするとエラーメッセージがでるときがある。そのときは「let 関数名 = 式」を「let 関数名 = fun 新引数 -> 式 新引数」とする

let abc = fun x -> x;;

let a = fun y -> abc 1 y;;

**〇if文**

・if文は以下の形式で書く。「if 式1 then 式2 else 式3」ここで、式2と式3の型は同じでないといけない

・比較演算子

⟨ 式 ⟩ = ⟨ 式 ⟩ 2 つの式の値が等しければ true

⟨ 式 ⟩ <> ⟨ 式 ⟩ 2 つの式の値が等しくなければ true

⟨ 式 ⟩ > ⟨ 式 ⟩ 左の値が右より大きければ true

⟨ 式 ⟩ < ⟨ 式 ⟩ 左の値が右より小さければ true

⟨ 式 ⟩ >= ⟨ 式 ⟩ 左の値が右より大きいか等しければ true

⟨ 式 ⟩ <= ⟨ 式 ⟩ 左の値が右より小さいか等しければ true

⟨ 式 ⟩ && ⟨ 式 ⟩ 「かつ」を計算する:両方の式が true ならば true

⟨ 式 ⟩ || ⟨ 式 ⟩ 「または」を計算する:少なくとも一方の式が true ならば true

・if文を式の中の部分式として記述することができる

“Hello, ” + if 1 > 0 then “Yuto” else “Nakano” → “Hello, Yuto”

**○組（タプル）**

・順番のある複数の要素を組み合わせたもの。()で囲む

・「let 名前 = (値1, 値2, …, 値n);;」とかく

・組の中から取り出すときは組の大きさが2ならば、1つめなら「fst 名前」、2つめなら「snd 名前」とする。

let a = (“Nakano”, “Yuto”);;

fst a;; // “Nakano”

snd a;; // “Yuto”

大きさが3以上のときは自分で関数を作る必要がある。「let 関数名 = (x, y, z) = z;;」なら3つめを取り出すことができるので名前は適当に「third」とかにしておくとよい

let b = ("Nakano", "Yuto", 20, "Sendai");;

let third (x, y, z, w) = z;;

third b;;

**○レコード（辞書）**

・レコードは順番がない複数の要素について、ラベルをつけて管理することができる

・{}で囲む

・最初に型の定義をする必要がある。ラベルごとの区切りは「;」を使う。

「type 名前 = {ラベル1 : 型; ラベル2 : 型; …; ラベルn : 型;}」

・レコードの定義はラベルと値を「=」で結び、定義した型と同じものをつくる

「let 関数名 = {ラベル1 = 値1; ラベル2 = 値2; …; ラベルn = 値n};;」

・レコードの中身を取り出すときは組と同じように関数を定義する

「let 関数名 = {ラベル1 = 値1; ラベル2 = 値2; …; ラベルn = 値n} = 値1;;」

・関数を作る以外にも参照の構文がある。（便利!）「名前.ラベル」で値が取り出せる

type maibo = {family : string; first : string; age : int};; // 型の定義

let dic = {family = "Nakano"; first = "Yuto"; age = 20};; // レコードの定義

dic.family;; // 値の取り出し

**○リスト**

・スタック構造を作っていて配列(array)とは異なる。

・[]で囲む

・[]は空リストを表し、「h :: t」はhが先頭でtがそれ以降のリストを指す。tは全体の長さから1小さいリストである

・リストの作り方は「1 :: 2 :: 3 :: []」のように間に「::」を差し込み、最後に[]をつける。この「::」をconsと呼ぶ

・リストの中身は全て同じ型である必要がある。

【リストのモジュール】

・「let リスト名 = [1 .. 5]」とすると[1; 2; 3; 4; 5]になる

・「List.filter (fun x -> 式) リスト名」とすると無名関数のパターンのときのみ取り出したリストを作れる。

a = [1 .. 10];;

List.filter (fun x -> x % 3 = 0) a;; // [3; 6; 9]

・「List.concat リストの中に入っているリスト」でリストを1つの大きいリストにできる

List.concat [[1;]; [2; 3]; []; [4; 5; 6]];; // [1; 2; 3; 4; 5; 6]

・「List.length リスト名」でリストの長さ

・「List.map 無名関数 リスト」で前から順に取り出してリストに処理をする

List.map (fun x -> 2 \* x) [1; 2; 3];; // [2; 4; 6]

・「List.rev リスト」でリストを逆順にする

・「List.append リスト1 リスト2」で2つのリストをつなげられる。「リスト1 @ リスト2」と中置演算子@を使ってもよい

【List.fold】

・リストの要素を取り出してなんらかの処理をする

・再起関数を1行で書ける

・「List.fold 関数 終わったときの値 リスト」

関数では「fun x y ->」で始めてxには累積値、yにはリストの要素が入る。「fun acc x -> acc + string x」としてするとリストを書き並べる

例、リストの長さを返す関数（再起関数）

let rec length alist = match alist with

| [] -> 0

| h :: t -> 1 + length t;

例、リストの長さを返す関数（List.fold）

let length alist = List.fold (fun acc x -> 1 + acc) 0 alist;;

例、リストの要素を1つの文字列

"数列: " + List.fold (fun acc x -> string x + ", " + acc) "終わり" [1;2;3;4;5];;

// “数列: 5, 4, 3, 2, 1, 終わり

【リストのパターンマッチ】

・以下の書式で書くことでリストの中身を先頭から順にif文のように処理できる。2行目はリストが空[]のとき、3行目はリストの中身があるときでhは先頭、tはそれ以降のリストを指している。

match 式1 with

[] -> 式2

| h :: t -> 式3

例、リストが空かどうかの判断

let isEmpty l = match with

[] -> true

| h :: t -> false;;

isEmpty [];; // true

isEmpty [1; 2; 3];; // false

・リストのパターンマッチングを用いて再起関数を書くことが多い

例、リストの和をとる

let rec sumList l = match l with

[] -> 0 // 空のときは0を返して終了

| h :: t -> h + sumList t;; // 残っているときは先頭を足して残りを再起する

例、リストをn倍する

let rec scalarMultL n l = match l with

[] -> [] // 空のときは最後に空リストをつける

| h :: t -> (h \* n) :: scalarMultL n t;; // 残っているときは後ろにリストを伸ばす

**○直和型**

・組（タプル）やレコード（辞書）のように複数のデータが1つになっていることを直積型という。それに対して、AとBのデータについて、「AまたはBのいずれかである」を表すのを直和型という。bool、int、listが直和型である

・bool型はtrueまたはfalse、intは ..., -2, -1, 0, 0, 2, ..., のいずれか、listは::または[]

・列挙型の作り方は「type 型名 = データ1 | データ2 | データ3」とする。型の名前は小文字から始め、データ名は大文字から始め、どちらも自分で新しく作る

例、曜日を表す型 wday

type wday = Sun | Mon | Tue | Wed | Thu | Fri | Sat;;

・パターンマッチを使って関数を書くことが多い

例、次の曜日を表す

let wdaynext d = match with Sun -> Mon | Mon -> Tue | Tue -> Wed | Wed -> Thu

| Thu -> Fri | Fri -> Sat | Sat -> Sun;;

・データ構成子に引数を持たせることができる「データ構成子 of 型」とする。データ構成子の名前は大文字から始めること

例、intかfloatを持つ型 number

type number = Int of int | Float of float;;

例、number型をstring型に変換する

int num2str n = match n with Int x -> string x

Float x -> string x;;

・引数をもつデータ構成子と引数を持たないデータ構成子を混在させた型を作ることができる。「Undef」と書いて、未定義を表すことが多い。

例、整数か未定義かを表す型

type intOrUndef = Int of int | Undef;;

例、割り算をする関数

type divInt n m = match (n, m) with

(Int x, Int 0) -> Undef // 0では割れないので未定義

(Int x, int y) -> Int (x / y) // int同士のときのみ定義する

(Int x, Undef) -> Undef

(Undef, int y) -> Undef

(Undef, Undef) -> Undef;;

・同じ形で異なる型を扱いたいときは多相データ型にする。例えば「intか未定義か」を扱うintOrUndefと「floatか未定義か」を扱うfloatOrUndefを’aを用いて一緒に定義することができる

例、値があるかないかを表す型

type ‘a option = Some of ‘a | None;;

例、値がないときにtrueを返す関数

let isNone x = match x with None -> true

| Some x -> false;;

・上の例の「option型」はF#で標準に定義されている。Noneは値がないことを示し、Some xは値があることを示す。xの型はなんでもよい

例、リストの先頭を取り出す関数

let headOpt l = match l with [] -> None

| h :: t -> Some h;;

・多相型は「type ‘型変数名 型名 = 型」とする。型変数名の前に「’」をつける

・再帰データ型は定義の中に定義したものが含まれている形の型である。リストや木にこの構造が用いられる

例、リストを自分で定義する(リストは :: または [] がくるはず)

type ‘a mylist = Cons of ‘a \* ‘a mylist | Nil

例、mylistの使い方

let a = Cons (4, Cons (-1, (Cons (1, Nil))));; // 4 :: (-1 :: (1 :: [])) を表している

例、mylistの長さを返す関数

let rec length l = match l with Nil -> 0

Cons (h, t) -> 1 + length t;;

例、二分木を表す型（二分木は左右に木、真ん中に値が入っているはず）

type ‘a tree = Leaf | Node of ‘a tree \* ‘a \* ‘a tree

例、二分木の歯の数を返す関数

let numLeaves t = match t with Leaf -> 1

Node of (left, n, right) -> numLeaves left + numLeaves right;;

**○代数的データ型とパターンマッチ**

・代数的データ型とは直積型、直和型、再帰型、多層型を組み合わせて作られたデータ型のことである

・代数的データ型はその要素から上から順にパターンマッチすることができるのが強い

・ワイルドカードパターンとしてアンダーバー「\_」を使うと、何にでもマッチするジョーカーのような扱いができる

例、リストが空かどうかを判断する関数

let isEmpty l = match l with [] -> true

| \_ -> false;

・「->」の左に「when bool式」を置くことでif文のように条件分岐ができる

例、リストの先頭が正なら先頭を返し、負かそれ以外ならNoneを返す関数

let positiveHead l = match l with h :: \_ when h > 0 -> Some h

| \_ -> None

例、以前に定義した二分木において最も左の値を返す関数

let rec leftMost t = match t with

Leaf -> None

| Node (Leaf, n, \_) -> Some n

| Node (l, \_, \_) -> leftMost l

**○入出力**

【出力】

・「stdout.WriteLine 値」とすると値を出力して改行する。値はstringでもintでもよい。

・「stdout.Write 値」とすると値を出力して改行しない

・出力はunit型として返る。unitは()のみからなる型である

・出力を続けて行うときは末尾に「;」をつける

・改行は「\n」で表す

例、続けて書くとき

stdout.WriteLine “今日の天気は？”; stdout.WriteLine “晴れ！”

今日の天気は？

晴れ！

例、カウントダウン（int型）

let rec countDown n = match n with

0 -> () // unit型は()を返すと終了

| n -> stdout.WriteLine n; countDown (n-1);;

countDown 3;; // int型でもいける

3

2

1

【入力】

・「stdin.ReadLine ();;」で1行分の入力を受け付けてstring型にする

例、空文字列を読むまで入力を受け続け、そのリストを最後に返す関数

let rec readLines () = // 引数にunit型をもってくるので「()」を入れる

match stdin.ReadLine () with // ここに処理を埋め込む

"" -> []

| x -> x :: readLines ();;

**○fsファイルのビルド**

・fsxファイルでは対話的にコンパイルしていたが、fsファイルはまとめてコンパイルする

**○その他**

・「変数名 = ref 値」と変数を定義すると変更可能な変数を定義できる（参照）型は元がint型ならref int型になる

・使うときには「!変数名」とする

・書き換えるときは「変数名 := 新しい値」

・「exception 例外名;;」として例外を自分で作ることができる。引数をつけるときは「exception 例外名 of 型;;」のようにする

・例外を投げるには「raise 例外」とする

・例外の補足にはtry式を用いる。match式のようにパターンマッチする。パターンには例外の種類を入れる。例外が発生しなければ式がそのまま評価される

try 式 with

| パターン1 -> 式1

| パターン2 -> 式2

例

exception MyError of int list;;

try raise (MyError [3]) with

MyError [] -> 0

| MyError (h :: t) -> h;;

・ライブラリで定義されている例外については「:? 例外名」と書く

type intOrUndef = Int of int | Undef

let divInt n m = match (n, m) with

| (Int x, Int y) -> (try Int (x / y) with

| :? System.DivideByZeroException -> Undef)

| \_ -> Undef