SMLコア言語入門

@no_maddo

September 30, 2015

Contents

Part I
Standard MLの世界へようこそ！

Standard ML（以下SML）の世界へようこそ！
TODO: SMLがどんな言語か説明する

1 本チュートリアルの読み方

このチュートリアルは足早に他のプログラミング言語を何か1つ以上知っている人向けにSMLのコア言語の書き方をSML/NJを題材に説明します。関数型言語の知識はあると理解が簡単かと思いますが必要としません。その都度説明します。

セクションごとにプログラム例が登場するので、ぜひ自分のマシンにSML処理系をインストールして試しながら読んでみて下さい。

途中のコラムでは、プログラミング言語マニア向けに言語デザイン・SMLの考え方・他の言語との比較に触れます。コラムはそこまでの知識では理解できない用語や概念が含まれますので、分からなければ後で読んでもらうのが良いと思います。

2 SML/NJ

2.1 SML/NJのインストール

2.2 対話環境について

TODO:

- ハローワールド
- 対話環境の見方を説明する
- ファイルのロード

2.3

Part II
基礎文法

3　全ては式

例えばC言語では値を返さないものありますが、SMLでは（関数宣言・型宣言などの宣言は除いて）全ての要素は値を返します。SMLに限らず一般に関数型言語では値を組み合わせてプログラミングしていきます。

ソースコード 1: 全ては値！

```
- 1
val it = 1 : int
- "god is god";
val it = "god is god" : string
```

同様にif文もCの三項演算子のように値を返します。if文のelse節は省略できません。if文の構文は **if** 式1 **then** 式2 **else** 式3（ただし式1の型はbool, 式2の型=式3の型）です。型付けに関しては4章で詳しく説明します。

ソースコード 2: if文は値

```
- if true then 1 else 2;
val it = 1 : int
- (if true then 1 else 2) + 1;
val it = 2 : int
```

4　静的な型検査

SMLではプログラムは静的に（コンパイル時に）型付けされます。動的型付け言語などと異なり型が合わないものはこの時エラーになります。

最初の例では `int` 型の値と `real` 型（Cの Double のこと）の値を加算しようとして型エラーになりました。C言語などでは暗黙のキャストによりコンパイルが通りますが、SMLではこのような仕組みは存在せず、自分でキャスト関数を挿入する必要があります。これは書くのには面倒ですが、プログラムを安全にします。[1]

次にif文に関する型エラーです。エラーメッセージを読めばなんとなく何がエラーになっているのか分かると思います。SMLのif文は、then節・else節で同じ型の値を返さなければなりません。下の例でthen節は `int` 型の値を返していますが、else節では `bool` 型の値を返しています。そのため型エラーが発生しました。

ソースコード 3: 型エラー1

```
- 1 + 2.0;
stdIn:5.1–5.8 Error: operator and operand don't agree [literal]
  operator domain: int * int
  operand:         int * real
  in expression:
    1 + 2.0
- if 1 > 2 then 1 else false;
stdIn:21.1–21.27 Error: types of if branches do not agree [literal]
  then branch: int
  else branch: bool
```

[1] 暗黙のキャストはプログラムを危険にします。例えば浮動小数点数演算の精度・ポインタのキャストなどがプログラマの意識外で行われてしまいます。

```
  in expression:
    if 1 > 2 then 1 else false
```

次に関数が登場する例を見てみましょう。下の例では、まず関数 print の型を確認し、それを使おうとしています。関数の方は T1 -> T2 などと、アロー（->）を用いて表されます。今はこの読み方は「T1 型の値を受け取ったら T2 型の値を返す」という理解で OK です。**??**章でより詳しく確認します。

関数適用の書式ですが 「関数名 引数 1 引数 2 ...」というような記法で書きます。C 言語で関数適用を書くのとは違いカッコが必要ありません。print("12") と書いてもエラーではありませんが ML らしいスタイルではありません。省略できるカッコは省略していきましょう。

さて、int 型の値である 12 をプリントしようとしていますが、print 関数の型は string -> unit で int を受け取るようにはできておらず、型エラーになります。

ソースコード 4: 型エラー 2

```
(* print関数の型を確認しよう *)
- print;
val it = fn : string -> unit

(* print型はstring型の値を受け取ってunit型の値を返すので、
   intを受け取ったら型エラー *)
- print 12;
stdIn:4.1-4.9 Error: operator and operand don't agree [literal]
  operator domain: string
  operand:         int
  in expression:
    print 12

(* int型の値をstring型の値にキャストする関数をはさもう
   ^ は文字列の結合のための演算子 *)
- print (Int.toString 12 ^ "\n");
12
val it = () : unit
```

5 値の束縛・パターンマッチ

5.1 val 文

さて、変数・関数を定義する方法を学びましょう。まずは変数から。

ソースコード 5: val 文

```
- val x = 12;
val x = 12 : int

- val y =
    if x > 100
    then Bool.toString true
    else Real.toString 3.14;
val y = "3.14" : string

(* 新しい要素：タプル *)
- val t1 = (1, 3.14);;
val t1 = (1,3.14) : int * real

- val t2 = (1 + 2, t1);;
val t2 = (3,(1,3.14)) : int * (int * real)
```

val 文は **val** 変数名 = 式という形で書かれます。注意すべきなのは、デフォルトで変数は immutable(変更不可能) であることです。そのため一度定義した値が変わらないことをコンパイルが

保証してくれるのでプログラムを安全に開発することができます！これはC などの言語を書いている人には違和感があるかもしれませんが、”Effective Java” などでも Immutable objects はプログラムをシンプルにする・デフォルトでスレッドセーフである、など沢山の利点があることが主張されています。興味があれば Effective Java や [1] を読んでみてください。

さて、ソースコード 6の最後の例で新しい型、タプルが現れました。タプルというのは組み型と呼ばれで、複数の型の異なる要素をまとめた構造を作ることができます。例えば `int` と `real` 型の値を 1 つにしているときには、`int * real` のようにこの型は表現されます。タプルの要素は任意の型の要素が許されるので、`int * (int * real)` などとタプルがネストすることも考えられます。

さて、タプルを作る方法はただ `(1, "2")` などと書くだけです。論理学などをやっている方はお分かりになると思いますが、導入規則には対で除去規則を導入しなければならないですよね？作り方がわかったところで、この構造を破壊する方法を学びましょう。

その前に、スコープの話をしましょう。ソースコード 6を見て下さい。SML では静的スコープを採用しています。そのため、一度参照したものの参照先が変わることはありません。

ソースコード 6: val のスコープ

```
- val name = "nadesico";
val name = "nadesico" : string
(* 変数nameを使ってタプルt1を定義 *)
- val t1 = (name, 1996);
val t1 = ("nadesico",1996) : string * int

(* 変数nameをもう一度定義
   この先nameと書いた時に指されるものはこっちの定義になる *)
- val name = "bebop";
val name = "bebop" : string
(* t2を定義しようとする時、name="bebop"となっている *)
- val t2 = (name, 1998);
val t2 = ("bebop",1998) : string * int

(* 変数nameを再定義しても、t1でnameが指しているものが変わるわけではない *)
- t1;
val it = ("nadesico",1996) : string * int
```

5.2 val 文を用いたパターンマッチ

タプルから値を取り出すには、主にパターンマッチを用います。具体例を見ていきましょう。ソースコード 7に例を示します。

ソースコード 7: val 文でパターンマッチ

```
- val t1 = ("nj", "poly", "alice");
val t1 = ("nj","poly","alice") : string * string * string
- val (a, b, c) = t1;
val a = "nj" : string
val b = "poly" : string
val c = "alice" : string

- a ^ " " ^ b ^ " " ^ c;
val it = "nj poly alice" : string
```

さて、上の例ではまずタプル `t1` を定義し、それをパターンマッチによって分解し中身の要素を取り出しました。val 文には **`val`** 変数名 `=` 式という形以外に **`val`** パターン `=` 式という形を書くことが出来ます。むしろ val のあとに来るものが変数名だけのケースは単純なパターンの 1 つで、「val パターン = 式」と覚えるのがより一般的です。**`val`** `(a,b,c) = t1` と書いた時、`t1` の第一要素が `a` に束縛・第二要素が `b` に束縛・第三要素が `c` に束縛されます。

パターンマッチは、val 式の左辺（パターンの部分）と右辺（式の部分）の形が一致していなければなりません。ソースコード 8を見てみましょう。

ソースコード 8: 型が合わないパターンマッチ

```
- val (a, b) = (1, 2, 3);
stdIn:20.5–20.23 Error:
pattern and expression in val dec don't agree [tycon mismatch]
  pattern:    'Z * 'Y
  expression:    int * int * int
  in declaration:
    (a,b) = (1,2,3)
```

上の例では val 文の左辺のパターンは 2 つの要素を受け取る形をしていますが、右辺の式の部分は 3 つの要素を持つタプルです。エラーメッセージでもパターンと式（expression）がミスマッチだと言っていますよね。

コラム：どんなパターンが存在するの

パターンは以下の BNF で表されます。慣れが必要ですがぜひ読んでみてください。<>でくくられたものはオプション（省略可能）です。

```
pat :=    atpat                          % 単純な場合（アトミックパターン）
        | <op> longvid atpat             % ヴァリアントパターン
        | pat vid pat                    % ヴァリアントの中置形式
        | pat : ty                       % 型注釈付き
        | <op> pat <:ty> as pat          % as パターン
atpat :=  sconst                         % 定数
        | <op> longvid                   % 変数
        | {<patrow>}                     % レコード
        | ()                             % unit 定数
        | (pat,...,pat)                  % タプルパターン
        | [pat,...,pat]                  % リストパターン
        | (pat)                          % カッコ
patrow := ...                            % ワイルドカード
        | label = pat <, patrow>         % フィールドパターン
        | vid<:ty> <as pat> <,patrow>    % レイヤード（layered）

longvid % ヴァリアント名
vid     % 変数名
ty      % 型
op      % 演算子
```

これを見て重箱の隅をつついてみましょう:)。
まず定数はパターンです！そのため `val 1 = 2` みたいなコードは許されます（パターンマッチに失敗したと言って実行時エラーになります）。
atpat に (pat) があるので、意味もなくカッコがネストした `val ((((((x))))))  = 1` のようなケースも許されます。
ヴァリアントを中置演算子にした形はパターンマッチすることが許されるので、

ソースコード 9: 中置形式のヴァリアントのパターンマッチ

```
datatype t = A of int * int
infix A

val x A y = ...
```

みたいなコードもかけます！
最後に、無駄に複雑なパターンを書いて終わりましょう。

ソースコード 10: 複雑なパターン

```
datatype t = A of t list * int | B;
infix A;

val a as [x, y A z, B] A b = [B, [] A 1, B] A 2;

val a = [B,[] A 1,B] A 2 : t
val x = B : t
val y = [] : t list
val z = 1 : int
val b = 2 : int
```

5.3 let 式

これまではトップレベルに定義を並べるだけでしたが、より複雑な定義をする時にはこれでは不便な時があります。

ネストした定義、定義の中にそれの補助のための定義を書けるようにしましょう。

これの必要性を理解するために、ライブラリを作ることを想像して下さい。例えばグラフ操作ライブラリを作っている時、ユーザに触らせる関数はどんな組み合わせをしてもグラフ構造が壊れないようにしたいけれども、処理の途中で使う受け取ったグラフを不正なグラフにして返すような関数は利用者が使えないようにしたいですよね。

そのために、カプセル化などの仕組みを用いて途中で用いる操作関数や途中状態の値はアクセス出来ないようにすることが多いですね。これをライブラリやクラス単位ではなく、式単位で行っていけばより安全性が高まります。それだけではなく、この関数・値はこの式の中でしか使わないんだな、などと言ったプログラマの意図をコードの中に表しやすくなります。[2] このような用途のために let 式を導入しましょう。

let 式の書き方は **let** 宣言 1　宣言 2　... **in** 式 **end** です。[3]**end** は忘れやすいですが必須なので気をつけて下さい。

例を見てみましょう。ソースコード 11を見て下さい。

ソースコード 11: let 式

```
(* 半径2の円の面積を計算してみる *)
- val area =
    let
      val pi = 3.14
      val r = 2.0
    in
      pi * r * r
    end;
val area = 12.56 : real

(* let式は式なので自由に組み合わせられる *)
- (let val r = 2 in r * r end) + 1;
val it = 5 : int
```

上の例では、まず半径 2 の円の面積 (area) を計算するために円周率 `pi` と `r` を定義して in 以降で使用しています。

また let 式はれっきとした式です。なので他の `1` や`"hoge" ^ "fuga"`といった式と同様に使うことが出来ます。

また let 式はスコープを作ります。`area` を定義するために定義した変数である `pi` や `r` には let 式の外側からはアクセスすることが出来ません。

ソースコード 12: let 式のスコープ

```
(* let式の中ではrにアクセスできるが、外側からはアクセスできない *)
- (let val r = 2 in r * r end) + r;
stdIn:19.6 Error: unbound variable or constructor: r

- val name = "Akito";
val name = "Akito" : string

(* 変数nameをlet式内で定義して使っているが、外側ではname="Akito"として使える *)
- (let val name = "Yurika" in name ^ ", " end) ^ name;
val it = "Yurika, Akito" : string
```

[2] また val 文しか導入していませんが、すぐに関数定義が登場して定義の中に関数定義が書けるようになります。

[3] Lisp 系言語の let 式と殆ど一緒です。

6　関数・リスト

6.1　関数を定義する

いよいよ関数型言語の味噌である関数について扱っていきます！関数は主に **fun** 変数名 仮引数1 仮引数2 ... = 式という fun 宣言により定義されます。ソースコード 13を見てみましょう。

ソースコード 13: 単純な関数の定義

```
- fun double a = a * 2;
val double = fn : int -> int
- val x = double 12;
val x = 24 : int

(* 関数の仮引数部分でもval文同様のパターンマッチが使える *)
- fun plus (x, y) = x + y;;
val plus = fn : int * int -> int
- plus (100, 200);
val it = 300 : int
```

関数適用は以前説明したとおり、「関数名 引数1 引数2 ...」という形で書かれます。C 言語のようにカッコは必要ありません。

関数には一切型を書いていないのにも関わらず、関数の型が推論されチェックされることに気をつけて下さい。関数 plus の中でプラス演算子を使っていますが、これは int 型の要素を 2 つ取ることをコンパイラ[4] は知っているため、x・y の型は int 型であると推論します。引数の型も推論でき、x + y の返り値も int であるので、関数 plus の型は (int * int) -> int となります。

fun 文の引数部分のパターンに対しても型推論が働きます。関数 plus では、引数部分にタプルパターンを書いています。この場合関数 plus は引数は 1 つで、それはタプルであると推論されます。

その後仮引数のタプルの要素である x, y の型は int 型であることが使われ方からわかるので、関数 plus は (int * int) -> int 型を持つことがわかります。

関数の定義に let 式を使うことが出来ます。let 式の書き方は **let** 宣言1 宣言2 ... **in** 式 **end** と書きましたが、この宣言に fun 文も書くことが出来ます。これによって関数宣言の中でしか有効でない、定義に必要な補助関数を定義することが出来ます。

ソースコード 14: ネストした関数宣言

```
- fun printPow x n =
    let
      fun pow x n =
        if n = 0 then 1 else x * pow x (n - 1)
      fun printWithBreak str =
        print (str ^ "\n")
    in
      printWithBreak (Int.toString (pow x n))
    end;
val printPow = fn : int -> int -> unit@/

- printPow 3 3;
@/27
val it = () : unit
```

6.2　演算子の定義

今まで演算子というものが存在するとしてきましたが、演算子にパラメータを渡す事は、単に関数適用のシンタックスシュガーに過ぎません。

演算子の型を確認してみましょう。演算子には、**op** をつけると通常の関数として扱えます。

[4] 正確にはこの場合対話環境ですが :)

ソースコード 15: 演算子を評価する

```
(* 型を確認する *)
- op +;
val it = fn : int * int -> int

(* 普通の関数みたいに使ってみよう *)
- op + (1, 2);
val it = 3 : int
```

演算子というのは記法は異なりますがただの関数だということが分かりましたが、反対に関数も演算子にすることが出来ます。

例を見てみましょう。ソースコード 16を見て下さい。

このコードでは、べき乗関数 power を定義し、それを中置記法で使えるようにしています。

ソースコード 16: 演算子の定義

```
- fun power (x, n) =
    if n = 0 then 1 else x * power (x, n - 1);
val power = fn : (int * int) -> int

(* powerを中置記法でかけるようにする宣言 *)
- infix power;
nonfix power

(* 使ってみよう *)
- 2 power 10;
val it = 1024 : int
```

power 関数は普通の数学で扱うようなべき乗の定義通りです。infix 宣言をするとこれを中置記法で使えるようになります。

中置記法でかけるようにするためには、関数は型が ('a * 'b) -> ... という形をしている必要があります。そうでないと演算子を定義できても、その演算子にどんな入力を与えても型エラーになります。

演算子にはやはり記号を使いたいですよね。記号で構成された関数も他の関数と動揺の方法で定義できます。ソースコード 17では power と書かずに**で住むように新たな演算子を定義しています。

ソースコード 17: 演算子の定義

```
- fun power (x, n) =
    if n = 0 then 1 else x * power (x, n - 1);
val power = fn : (int * int) -> int

(* 記号から始まる関数はopをつけてから関数名を書く *)
- fun op ** (x, n) = power (x, n);
val ** = fn : int * int -> int

(* 普通の関数と同じように使えるよー *)
- ** (2, 10);
val it = 1024 : int

(* powerを中置記法でかけるようにする宣言 *)
- infix **;
infix **
- 2 ** 10;
val it = 1024 : int
```

6.3 パラメトリック多相性 (Parametric Polymorphism)

今までの関数はすべての型の構成要素が具体的な型（int や real）になっていました。型推論の説明でも、定義の中に現れる演算子や関数の型から引数や関数の型を決定していると説明しました。

さて、もし関数を定義するとき、型の制約が何もなかったらどんな型を付ければよいでしょうか？具体的には `fun id x = x` のように関数 `id` を定義した時、`x` にはなんの型の制約はありません。この関数はどんな型を持つべきでしょうか。

正解は「どんな型でもいい、型変数'a」を用いて、`'a -> 'a` という型がつきます！ソースコード 18を見て下さい。

ソースコード 18: 多相関数

```
- fun id x = x;
val id = fn : 'a -> 'a

(* id関数を使ってみる *)
- (id 1, id 3.0, id "hoge");
val it = (1,3.0,"hoge") : int * real * string
```

`'a -> 'a` の`'a` はどんな型にもなれます。例えば `id 1` というのを計算するとき、`'a` は `int` に単一化されています。そのため、`id 1` の返り値も `int` であることがわかります。

このような、型変数が含まれる関数のことを「多相関数」と呼びます。

単純ですがよく使われる多相関数をいくつか定義してみましょう。

ソースコード 19: 多相関数たち

```
- fun fst (x, y) = x;
val fst = fn : 'a * 'b -> 'a
- fun snd (x, y) = y;
val snd = fn : 'a * 'b -> 'b

(* 使ってみる *)
- val t1 = (1, "hoge");;
val t1 = (1,"hoge") : int * string
- fst t1 + 2;
val it = 3 : int

- fun revApp x f = f x;
val revApp = fn : 'a -> ('a -> 'b) -> 'b
```

6.4 再帰関数

さて、いよいよ繰り返しを含むプログラムを書いていきましょう。関数型言語ではよく繰り返しを書くために、再帰関数を用います。

例を見てきましょう。ソースコード 20をみてください。基本的には他の言語で書き下したのと同様です。

ソースコード 20: 再帰関数

```
- fun fact n = if n = 1 then 1 else n * fact (n - 1);
val fact = fn : int -> int
- fact 5;
val it = 120 : int
```

再帰関数で気をつけることは特にありません。せいぜい、再帰呼出しのために今定義しようとしている関数名が fun 文の右辺（fun 文のイコール以降の式）に現れる、くらいでしょうか。

関数 `fact` を見てみましょう。ご存知のように、階乗は以下のように定義されます。

$$fact\ n = \begin{cases} 1 & (if\ n = 1) \\ n * fact(n-1) & otherwise \end{cases}$$

階乗の定義は入力が $n = 1$ であれば 1 を返し、そうでなければ $n * fact(n-1)$ というものです。この定義を素直に書き下しています。

再帰関数は帰納法的な考え方で構築することが出来ます。というのは関数 `fact` の場合、

- ベースケースを記述する（この場合 $n = 1$ のとき）

- 帰納法の仮定（`fact (n - 1)` は正しく計算される）をつかって、どうすれば `fact n` の計算式は正しくなるのか考える

というものです。

「$fact(n-1)$ は正しく $n-1$ の階乗として計算されるので、$n * fact(n-1)$ も n の階乗を計算する式として正しい」と考えることができれば完璧です。この考え方を用いて後で何回も再帰関数の説明をします。

6.5 高階関数

さて、関数型（アロー型）の表現は `T1 -> T2` という形で表されます。この意味は T1 型の値を引数に取り、T2 型の値を返す関数という意味でした

この例で T1 や T2 はアロー型であることも許されます。そのような関数はどんな性質を持ちうるのでしょうか？例を見てみましょう。

ソースコード 21: 第一級関数

```
(* T1->T2のT2が関数型である場合 *)
- fun plus x y = x + y;
val plus = fn : int -> int -> int

- val plusOne = plus 1;
val plusOne = fn : int -> int
- val plusTwo = plus 2;
val plusTwo = fn : int -> int
- plusTwo (plusOne 1);
val it = 4 : int

(* T1->T2のT1が関数型である場合 *)
- fun twice f = f (f 1);
val twice = fn : (int -> int) -> int
- twice plusOne;
val it = 3 : int
```

まずは T2 が関数型の例から見てみます。ソースコード 21の上の例では、`int -> (int -> int)` 型の 2 つ引数を取り、その引数を足すだけの関数 `plus` を定義しています（プログラム中の `int -> int -> int` はカッコを省略した形式です）。

この関数に対しても今までの読み方と同じ解釈をすることが出来ます。すなわち、関数 `plus` は `int` 型の値を 1 つ受けると `int -> int` 型の値を返す、関数を返す関数なのです！

関数を返す関数は、同じような関数をたくさん定義する場合に便利です。（この便利さは 6.5節を見てみると実感できると思います）。例では同じような形をするだろう、関数 `plusOne` と `plusTwo` を定義しています。関数 `plusOne` は関数で、一つ引数をとり 1 足す関数になっています（関数 `plus` の仮引数の x にすでに 1 が入っていると考えて下さい）。関数 `plusTwo` も同様です。

T1 が関数型であるような関数どういう解釈が出来るかというと、関数を受け取る関数であると考えらる事が出来ます。関数 `twice` は 1 に対して同じ関数を 2 回適用する関数です。[5]`int -> int` 型の関数を引数にとり、それを 2 度使います。

このように関数が他の `int` 型や `string` 型の値と同じように扱えることを指して「関数が第一級である」と言うことがあります。また関数を操作する関数を指して「高階関数」と呼びます。

さて、最後にこれらの要素を使った例として微分関数 `diff` を定義してみましょう。ソースコード 22を見て下さい。

微分演算子 ′（この例では `diff` 関数）はまず `real -> real` 型の関数 f を受け取り、一引数関数を返す関数です。

微分演算の定義を確認してみると、$f' = \lim_{a \to 0} \frac{f(x+a)-f(x)}{a}$ でしたね。今回は極限は近似して $a = 0.0000000001$ として計算してみます。

[5] ちなみに「関数を引数に適用する」という言葉は関数と引数の順番をよく間違えられています。私は英語で覚えています。apply A to B は、意味は A を B 当てはめるという意味です。ジェネリックな存在である関数を実際の引数に当てはめる、という覚え方が簡単かなぁと思います

ソースコード 22: 高階関数の例

```
- fun diff f x = (f (x + 0.0000000001) - f x) / 0.0000000001;
val diff = fn : (real -> real) -> real -> real

(* Math.sinはMathモジュールに含まれるsin関数にアクセスする記法 *)
- val myCos = diff Math.sin;
val myCos = fn : real -> real

(* 正しく動いているか確認 *)
- (myCos Math.pi, Math.cos Math.pi);
val it = (~1.00000008274,~1.0) : real * real
- (myCos (Math.pi / 2.0), Math.cos (Math.pi / 2.0));
val it = (0.0,6.12303176911E~17) : real * real
- (myCos (Math.pi / 3.0), Math.cos (Math.pi / 3.0));
val it = (0.50000004137,0.5) : real * real
```

関数 diff を定義した後、確認のため標準ライブラリの Math モジュールに含まれる sin 関数を微分して myCos を定義しました。

その後、ライブラリに含まれる cos 関数と myCos に同じ引数を与え、出力結果を比べられるようにしました。

誤差はあるものの、概ね cos 関数として動作しているように見えますね。

コラム：アドホック多相性

関数 plus の型を見て不自然に思った方もいるかもしれません。
プラス演算子は 3.14 + 1.1 のように、real 型の値の演算にも用いられます（対話環境で評価してみて下さい）。
しかし、ソースコード 21の関数 plus は int -> int 型の関数であると推論されています。real -> real 型の plusReal を定義するためには **fun** plusReal (x:real) y = x + y のように型注釈を書く必要があります。

関数 plus に int -> int 型がつくことは少し変です。なぜならプラス演算子は real 型にも使えるはずなので、より一般的な型がついてもいいはずです。
これは SML の型システムに由来します。SML では（TODO:SML には、というのは主語が大きすぎる？）Haskell の型クラスのような「型パラメータ'a、ただし'a の動く範囲は制限されている」というような事がかけません。
SML の型システムで表せるのは「何でも受け取る'a」もしくは「int、real のような具体型」だけです。そのため、SML ではプラス演算子にようにアドホックに（場当たり的に）演算子は多相的になっているものがありますが、それを型として表すことが出来ません。

この問題に対する解決策はいくつか考えられています。
一つは型クラスです。型クラスでは、上で述べたような「型パラメータ'a、ただし'a の動く範囲は制限されている」という事がかけます。
Haskell では例えば、加算演算子は Num a => a -> a -> a という型を持ちます。Num a の意味は「型パラメータ a は Num に属している型に限られる」という意味です。

ソースコード 23: Haskell での plus の型付け

```
Prelude> :t (+)
(+) :: Num a => a -> a -> a

Prelude> let plus x y = x + y
Prelude> :t plus
plus :: Num a => a -> a -> a
```

東北大学の大堀研で開発されている SML#[3] では、第一級オーバーローディングが実装されています。例えば、関数 plus の型は以下のように表されます。

ソースコード 24: SML#での plus の型付け

```
fun plus x = x + x;
val plus = fn
  : ['a::int, IntInf.int, real, Real32.real, word, Word8.word. 'a -> 'a]
```

'a::int, IntInf.int, real, Real32.real, word, Word8.word. は型変数'a は int, IntInt.int, real, ... のどれかであることを表しています。このような型を持つ関数もユーザが書くことも出来ます。

7 リスト

7.1 リストの基本

リストは Immutable なデータ構造で、データの列を扱うときに関数型プログラミングでは良く用いられる重要なデータ構造です。

リストは今までの型とは少し異なり、list だけでは型を表しません。型の一部をパラメータ化しており、int list や string list など、「何かのリスト」であることを型の上で表現します。

まずは例を見てみましょう。ソースコード 25を見て下さい。

ソースコード 25: 色々なリスト

```
- [1,2,3];
val it = [1,2,3] : int list
```

```
- ["sml", "ocaml", "haskell", "fsharp"];
val it = ["sml","ocaml","haskell","fsharp"] : string list
- [[(1, 2)],[(3, 4)]];
val it = [[(1,2)],[(3,4)]] : (int * int) list list

- [];
val it = [] : 'a list
```

'a list の'a 部分が色々な型に変わっていますね。'a は何が代入されてもいいので、リストがネストしたり中にタプルが入っても構いません。

空のリストなど、'a list の'a 部分が単相化（具体的な型に置き換わってない）されていないリストも考えられます。そのため、多相的なリストというものも考えられます。

ソースコード 26: 多相的なリスト

```
- val empty = [];;
val empty = [] : 'a list

- 1 :: 2 :: empty;
val it = [1,2] : int list
- "hoge" :: empty;;
val it = ["hoge"] : string list
- [] :: empty;;
val it = [[]] : 'a list list
```

リストを扱うためにプリミティブな演算子を紹介します。

- ::演算子は、リストの先頭に要素を追加して新しいリストを作る演算子です。
- @演算子は、リスト同士を結合する演算子です。

ソースコード 27: リスト操作演算子の型

```
- op ::;
val it = fn : 'a * 'a list -> 'a list
- op @;
val it = fn : 'a list * 'a list -> 'a list
```

注意することは、::は右結合的だということです。例えば1 :: 2 :: [] と書いた時には1 :: (2 :: [])のように演算子が結合します。(1 :: 2) :: [] という結合順序では型エラーになりますよね？

さて、これらの演算子を使って簡単なリスト操作をする関数を書いてみましょう。その前に val 文・fun 文以外のパターンマッチについて触れます。

7.2 case 式

val 文などのパターンが書ける部分でパターンマッチが出来ることは既に触れましたが、パターンマッチをしてその結果によって実行結果を分岐する構文があればリスト操作では「入力が空リストかそうでないかで処理を分ける」みたいな事が簡単にかけたら便利そうです。

そのために case 式を用います。case 式は **case** パターンマッチする式 **of** パターン 1 => 式 1 | パターン 2 => 式 2 ... のように使います。

例を見てみましょう。ソースコード 28を見て下さい。

ソースコード 28: case 式

```
- val l = [1,2,3];
val l = [1,2,3] : int list

- case l of
    [] => "empty"
  | x :: [] => "one"
  | x :: y :: [] => "two"
```

```
  | x :: y :: xs => "more";
val it = "more" : string

- fun sum l = case l of
    [] => 0
  | x :: xs => x + sum xs;
val sum = fn : int list -> int
- sum [1,2,3];
val it = 6 : int
```

前半部分では cases 式を評価しています。この case 式はリストが空の時、要素が 1 だけの時、要素が 2 つだけの時、それ以上の時で分岐しています。

次の関数 sum では、仮引数 l が空リストかそうでないかで分岐しています。これはリスト操作をする関数では典型的なパターンです。

関数 sum は以下の帰納法のような考え方に基づいて作られています。

- ベースケース：入力 $l = []$ のとき和は 0
- 入力 $l = x :: xs$ のとき、$sum\ xs$ は正しく計算されていると仮定する。
  この時 $x + sum\ xs$ はリスト l の和として正しい

7.3 リスト操作関数

やっと準備が整いました。リスト操作をする実用的な関数を定義していきましょう。

ソースコード 29: リストを逆順にする関数 rev を定義する

```
- fun rev l = case l of
    [] -> []
  | x :: xs = rev xs @ [x];
val rev = fn : 'a list -> 'a list

(* revを使ってみる *)
- rev [1,2,3];
val it = [3,2,1] : int list
```

関数 rev の作り方が正しいことを、また帰納法のような考え方でこれを説明してみましょう。

- ベースケース：入力 $l = []$ のとき、空リストは逆順に既にしてある
- 入力 $l = x\ ::\ xs$ のとき、$rev\ xs$ は xs を正しく逆順にすると仮定する。
  この時 $rev\ xs$ の後ろに x を付ければ $l = x\ ::\ xs$ を正しく逆順にする

関数の引数に対して rev と同様に引数を case 式でパターンマッチ出来ました。ここで、fun 文でタプルのパターンマッチをしたことを思い出しましょう。これと同様なことがリストに対してもできれば便利そうです。

関数 rev をその形で書きなおしてみます。ソースコード 30を見てください。

ソースコード 30: 引数部分でのパターンマッチ

```
- fun rev [] = []
    | rev (x :: xs) = rev xs @ [x];
val rev = fn : 'a list -> 'a list
```

上の書き方では、引数部分に直接定数（この場合空リスト）を書いています。実は定数は一種のパターンです。定数パターン空リストの時は食うリストを返し、そうでなければその下の行の処理を行います。

ソースコード 29と 30は挙動は全く一緒なので、好みで使い分けて下さい。この文章では一貫してあとで導入した記法は使わず、case 式を用います。

次に関数 map を定義してみましょう。map 関数は関数をひとつ受け取り、それを更に受け取ったリストのすべての要素に適用する関数です。[6]

[6]他の言語では reduce と呼ばれることもあります。

ソースコード 31: 関数 f とリスト l を受け取り、l の要素全てに f を適用する関数

```
- fun map f l =
    case l of
      [] => []
    | x :: xs => f x :: map f xs;
val map = fn : ('a -> 'b) -> 'a list -> 'b list

(* mapを使ってみる *)
- fun add1 n = n + 1;
val add1 = fn : int -> int
- map add1 [1,2,3];
val it = [2,3,4] : int list

- map Int.toString [1,2,3];
val it = ["1","2","3"] : string list
```

上の例では map を定義した後、受け取った引数に 1 足すだけの関数関数 add1 を定義して map 関数に渡しています。map add1 [1,2,3] の実行結果を見ると [2,3,4] となっており、すべての要素が 1 たされていることが分かります。

また関数 map の型を見ると、('a -> 'b) -> 'a list -> 'b list となっています。この型と関数名から概ね動作を予想できます。なぜなら今与えられたのは f : 'a -> 'b と l : 'a list だけです。これから'b 型の値を手に入れるには、リストの要素に f を適用するしか方法がありません。それでいて名前が map です。名前と型で動作が予想できることが分かっていただけたと思います。

ここでわざわざ関数 map に渡すためだけに関数 add1 を定義するのは面倒です。その場で関数がささっと書ける方法があると便利そうです。

名前のない関数である「匿名関数 (Anonymous function)」は、**fun** パターン => 式という形で書くことが出来ます。これも他の関数同様第一級です。

ソースコード 32: 匿名関数

```
- (fn x => x + 1) 3;
val it = 4 : int

- List.map (fn x => x * x) [1,2,3];
val it = [1,4,9] : int list
```

匿名関数としてカリー化されている、複数引数を受け取る関数を作るには **fn** x => **fn** y => ... のように書く必要ががります。

7.4 List モジュール

最後に SML/NJ が提供する標準ライブラリの List モジュールに含まれる関数を見ていきます。全ての関数は取り上げないので、詳しくはリファレンスを見て下さい [2]。

リストは本当によく使うので、List モジュールに含まれる関数はほぼすべて覚えるくらいに勢いで使い方を覚えると良いと思います。

さて、いくつかの関数を実際に使ってみましょう。

ソースコード 33: List モジュール

```
- val l = List.tabulate (10, fn x => x);
val l = [0,1,2,3,4,5,6,7,8,9] : int list

- List.take (l, (List.length l div 2));
val it = [0,1,2,3,4] : int list

- List.drop (l, (List.length l div 2));
val it = [5,6,7,8,9] : int list

- List.concat [[1,2,3], [4,5,6], [7,8,9]];
```

関数名	型	説明
hd	'a list -> 'a	先頭要素を取り出す
tl	'a list -> 'a list	先頭要素を取り除いたリストを返す
last	'a list -> 'a	最後の要素を返す
nth	'a list * int -> 'a	n 番目の要素を取り出す
take	'a list * int -> 'a list	先頭 n 番目までのリストを返す
drop	'a list * int -> 'a list	先頭 n 番目まで捨て残りのリストを返す
length	'a list -> int	リストの長さを返す
rev	'a list -> 'a list	リストを逆順にして返す
concat	'a list list -> 'a list	ネストしたリストを平にする
map	('a -> 'b) -> 'a list -> 'b list	入力 f を入力リストの全てに適用する
find	('a -> bool) -> 'a list -> 'a option	述語 f が true になった値を返す
filter	('a -> bool) -> 'a list -> 'a list	述語 f が true になった値を集めて返す
partition	('a -> bool) -> 'a list -> 'a list * 'a list	述語 f が true になった要素と false になった要素に分ける
foldr	('a * 'b -> 'b) -> 'b -> 'a list -> 'b	後述
foldl	('a * 'b -> 'b) -> 'b -> 'a list -> 'b	後述
exists	('a -> bool) -> 'a list -> bool	述語 f が true になる要素が存在するか
all	('a -> bool) -> 'a list -> bool	すべての要素が述語 f を満たすか
tabulate	int * (int -> 'a) -> 'a list	0 から n-1 を関数 f を適用した結果を返す

```
val it = [1,2,3,4,5,6,7,8,9] : int list

- List.filter (fn x => x mod 2 = 0) l;
val it = [0,2,4,6,8] : int list

- fun capitalize str =
    let
      val chars = String.explode str (* String.explode: string -> char list *)
      val capitalized = List.map Char.toUpper chars (* Char.toUpper: char -> char *)
    in
      String.implode capitalized (* String.implode: char list -> string *)
    end;
val capitalize = fn : string -> string
- capitalize "ryoko, hikaru, izumi";
val it = "RYOKO, HIKARU, IZUMI" : string
```

8 レコード

TODO: 構造の作り方・壊し方（fun 文・case 文でのパターンマッチ）

9 ユーザ定義データ型

9.1 ヴァリアントの定義

TODO:

- case 文で分解する
- enum との比較

ソースコード 34: 単純なヴァリアント

```
datatype inputs = NOTHING | ARG of string | FLAG of string * bool;
datatype inputs = ARG of string | FLAG of string * bool | NOTHING
```

```
val l = [FLAG ("-c", false), FLAG ("-i", true), ARG "a.ml", ARG "b.ml"];
val l = [FLAG ("-c",false),FLAG ("-i",true),ARG "a.ml",ARG "b.ml"]
  : inputs list
```

9.2 再帰データ構造

TODO:

- 上の定義の問題 [NOTHING, NOTHING] みたいなのがかけることを指摘
- リストっぽい構造にする

コラム：多相ヴァリアント

TODO: OCaml で多相ヴァリアントを説明

10 手続きプログラミング

10.1 参照型

10.2 例外

10.3 繰り返し

コラム：手続きとの付き合い方

11 モジュール

11.1 ライブラリの利用

11.2 モジュールの定義

11.3 シグニチャ

コラム：オブジェクトのカプセル化

11.4 ファンクタ

コラム：第一級モジュール

Part III
プログララムの例

References

[1] Yegor Bugayenko, “Objects Should Be Immutable”
http://www.yegor256.com/2014/06/09/objects-should-be-immutable.html

[2] The Standard ML Basis Library
http://sml-family.org/Basis/index.html

[3] 東北大学電気通信研究所 大堀研究室, “SML # プロジェクト”
http://www.pllab.riec.tohoku.ac.jp/smlsharp/ja/