SMLコア言語入門

@no_maddo

September 26, 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！最低限SMLコードを理解しこの本の話題を楽しく読めるようになることです！

1　SML/NJ

1.1　SML/NJのインストール

1.2　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

1.3

Part II
基礎文法

2　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3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
```

3　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
  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。[2]

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
```

[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。

[2] 別に 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。

```
val it = () : unit
```

4 値の束縛・パターンマッチ

4.1 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 5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。

4.2 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。具体例を見て行きましょう。ソースコード 6に例を示します。

ソースコード 6: 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。[3]`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 7を見てみましょう。

ソースコード 7: 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

5 関数・リスト

5.1 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！

ソースコード 8: 単純な関数の定義

```
fun double a = a * 2
val double = fn : int -> int
```

5.2 再帰関数

TODO: fib, fact などを説明する帰納法的な考え方

5.3 リスト

リストは Immutable なデータ構造で、データの列を扱うときに配列よりも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。これを取り扱うための関数をいくつか定義していきましょう。

5.4 高階関数

TODO: map, filter, exists, forall などを説明する match 文を説明する匿名関数

5.5 カリー化・部分適用

コラム：演算子？

SML では演算子はただの関数です。TODO:続きを書く

[3]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。

6　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）

7　ユーザ定義データ型

7.1　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 9: 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
```

7.2　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

8　手続きプログラミング

8.1　参照型

8.2　例外

8.3　繰り返し

コラム：手続きとの付き合い方

9　モジュール

9.1　ライブラリの利用

9.2　モジュールの定義

9.3　シグニチャ

コラム：オブジェクトのカプセル化

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