

SML コア言語入門

@no_maddo

October 19, 2015

Contents

I	Standard ML の世界へようこそ！	2
1	本チュートリアル の 読み方	2
2	SML/NJ	3
2.1	SML/NJ のインストール	3
2.2	対話環境について	3
II	基礎文法	4
3	式を組み合わせる	4
4	静的な型検査	4
5	値の束縛・パターンマッチ	6
5.1	val 文	6
5.2	val 文を用いたパターンマッチ	7
5.3	let 式	9
6	関数・リスト	10
6.1	関数を定義する	10
6.2	型注釈	11
6.3	演算子の定義	11
6.4	パラメトリック多相性	13
6.5	再帰関数	13
6.6	高階関数	14
6.7	local 文	15
7	リスト	16
7.1	リストの基本	16
7.2	case 式	17
7.3	リスト操作関数	18
7.4	List モジュール	19
8	レコード	21
8.1	型のエイリアス	23
9	ユーザ定義データ型	24
9.1	ヴァリエントの定義	24
9.2	再帰的ヴァリエント型	26

10 例外	30
11 手続きプログラミング	30
11.1 逐次実行	30
11.2 参照型	31
11.3 繰り返し	31
11.4 配列	32
11.5 値制約 (value restriction)	32
12 モジュール	33
12.1 モジュールの定義	34
12.2 シグニチャ	34
12.3 ファンクタ (functor)	36
III プログラムの例	38

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！

SML は非純粋（副作用、破壊的代入が出来る）な関数型言語の一種です。Strict な（C などと同じ評価順序）評価戦略・静的な（コンパイル時の）型付け・高階関数（関数を操作する関数）など静的型付けを行う関数型言語として標準的な機能を持つ、ML と呼ばれる歴史ある言語ファミリに属しています。

特徴としては Scheme のように言語の定義が存在します [?]. 言語の定義が存在すると、言語を処理系を叩いて理解するのではなく、より定式化されたものを対象に理解することが出来ます。

個人的な私見ですが、ML は関数型言語を学ぶための言語として最適です。わかりやすい言語設計で、特に型システムは関数型言語でも標準的で分かりやすいです。日本では関数型言語として名前が上がるのはまず Haskell, F# などかもしれませんが、これらの型システムもベースは ML で主に用いられる Hindley-Milner 型システムをベースにしています。

特に言語処理系に興味がある人にとっては、SML は様々な実験的な処理系が作られてきたので SML を学ぶ意義が大いにあります。CPS 変換によってすべての関数呼び出しが末尾再帰呼び出し化される”Standar ML of New Jersey”や Whole optimized compiler として有名な、分割コンパイルを諦めて強力な最適化がかかる”MLton”, 定理照明器 Isabelle/HOL の開発に用いられている Poly/ML など。Type Passing を全面的に利用しようとして夢破れた TIL など、実験的な論文レベルの処理系もたくさんあります。

ぜひこの同人誌を楽しく読むために、SML のコア言語を学んで下さい。

1 本チュートリアルを読み方

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が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。

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

基本的にはインストールの指示に従えば特に困難なくインストールすることが出来ます。

<http://www.smlnj.org/dist/working/index.html> で最新版を確認してください。

- Windows の場合

Windows 向けにインストーラが存在します。上で示した URL から最新版をクリックし、.msi ファイルをダウンロードし、指示に従って下さい。

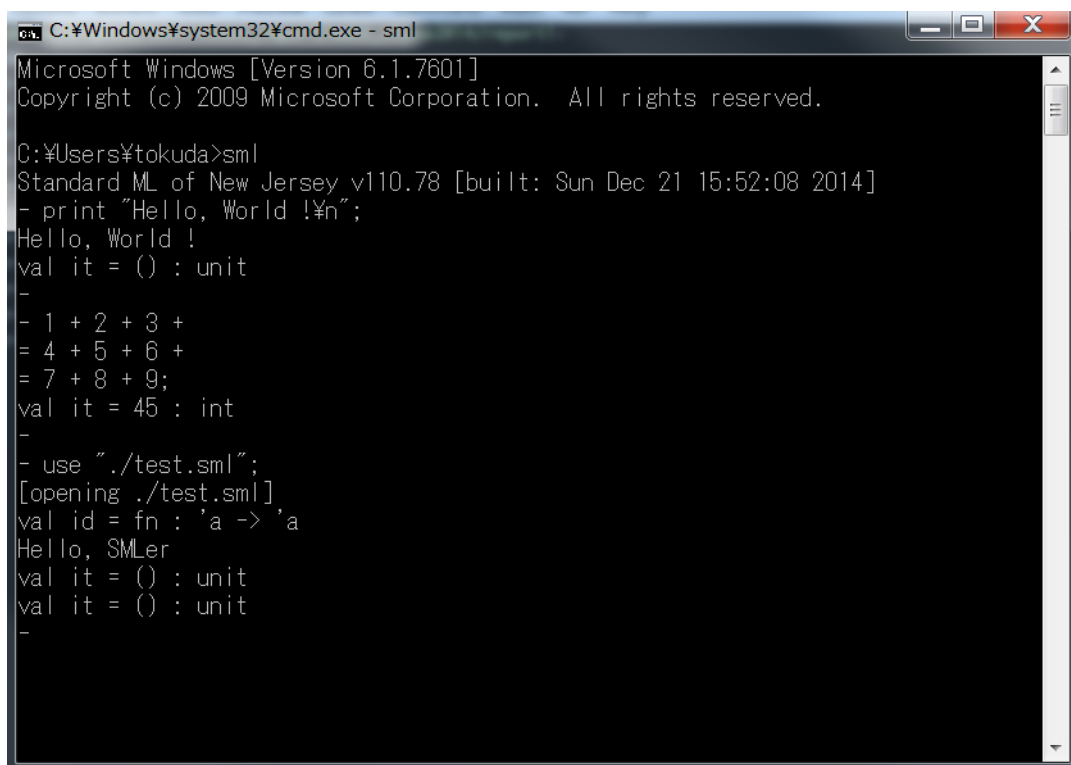
Windows ではコマンドプロンプト上で SML/NJ の対話環境が動くことになります。コマンドプロンプトではコピーアンドペーストや基本操作がしづらいと思うので Emacs など他のプログラム上でコマンドプロンプトを実行するようなプログラムが必要かもしれません。

- Linux 系 OS の場合

多くのメジャーディストリビューションではパッケージ管理ソフト経由でインストールすることが出来ます。執筆現在 (2015 年 10 月 12 日) で最新版は 110.78 です、パッケージ管理ソフト経由でインストールできるコンパイラは古いかもしれませんが基本的には問題ないはずです。ml-build などツールがうまく動かないようならばソースコードからコンパイルして下さい。

2.2 対話環境について

Windows を例に見方を説明します。



```
C:\Windows\system32\cmd.exe - sml
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\tokuda>sml
Standard ML of New Jersey v110.78 [built: Sun Dec 21 15:52:08 2014]
- print "Hello, World !\\n";
Hello, World !
val it = () : unit
-
- 1 + 2 + 3 +
= 4 + 5 + 6 +
= 7 + 8 + 9;
val it = 45 : int
-
- use "./test.sml";
[opening ./test.sml]
val id = fn : 'a -> 'a
Hello, SMLer
val it = () : unit
val it = () : unit
-
```

Figure 1: 対話環境

コマンドプロンプト上で sml と入力し、SML/NJ を起動しましょう。その後上の画像のように、`print "Hello, World !\\n";` と入力してみてください。セミコロンも必ず必要です。画面に正しくプリントされれば成功です！

対話環境の見方を説明します。

- 各行の最初のハイフン (-) は対話環境が挿入するプロンプトです。プログラムの一部ではないので気にして大丈夫です。

- プログラムが複数行に渡る時は、対話環境がイコール (=) 記号を自動的に挿入します。これもプログラムの一部ではないので気にして大丈夫です。
- 行の最後に必要なセミコロン (;) は対話環境に入力が終わったことを伝えるために必要な文字です。これがなければずっと対話環境は入力待ちをします。
- 画面で Hello, World ! の次の行にかかっているように、プログラムを実行（これ以降評価と呼びます）すると計算結果とその型がプリントされます。これによってユーザはインタラクティブに関数を定義し、計算結果を手早く確認しながらプログラミングを続けることが出来ます。
- 対話環境にいちいち打ち込むのは面倒かもしれません。ファイルのロードには `use "./test.sml";` のように、`use` 関数を使ってファイルを読み込みます。この場合はファイルはカレントディレクトリにあるため特に指定していませんが、ファイルのパスを入力して下さい。そうすると、ファイルに書かれた定義の読み込みやプログラムの評価が行われます。

Part II

基礎文法

3 式を組み合わせる

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 (*value*) を返します。値を返すものを式 (*expression*) と呼びましょう。それ以外のは文または宣言と呼びます。SML に限らず一般に関数型言語では式を組み合わせてプログラミングしていきます。以後、式と文は明確に区別されているので気をつけて下さい。

値とは直感的には整数・浮動小数点数・文字列・関数とそれを組み合わせて作ったデータ構造などこれ以上簡単に出来ないプログラムのことです。例えば `print "Hello, world!"` は関数と文字列の適用なので値ではありません。この評価結果 (計算結果) である `()` (ユニット) は値です。

同様に `if` 式も C の三項演算子のように値を返します。`if` 式の `else` 節は省略できません。`if` 式の構文は「**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。

SML では、プログラムの実行というのは式を評価して値にすることです。後述の静的な型検査によって評価途中にある種のエラーにならないことを保証できます。

ソースコード 1: `if` 文は式

```
- if true then 1 else 2;
val it = 1 : int
(* if式は式なので自由に組み合わせられる *)
- (if false then 1 + 2 else 3 + 4) * (if true then 4 else 6);
val it = 28 : 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。動的型付き (実行時に型の整合性を確認する) 言語と異なり、型が合わないものはこの時エラーになります。静的な型付けを行う言語の例として、C や Java を思い浮かべると静的な型付けというのは変数を宣言するたびに冗長な型を書かなくてはいけない、いちいち変更するのに面倒というイメージがあるかもしれません。SML には型推論 (*type inference*) と呼ばれる、型をプログラマが書かなくても処理系が補ってくれる、という機能があります。そのためソースコード 1 でも一切型を書かなくても返り値は `int` だな、と判断してくれました。

さて、型エラーになる例を見てみましょう。ソースコード 2 を見て下さい。

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型検査に成功しますが、SML ではこのような仕組みは存在しません。キャストする関数により明示的に変換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

その次に `if` 式に関する型エラーです。SML の `if` 式は、`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 2: 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。ソースコード 3 を見て下さい。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。関数の型は `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。

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 らしく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、`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 3: 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
```

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。

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 4: 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変更不可能 (*immutable*) 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ラ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。

変数が変更可能な言語でその変数を複数箇所で書き換えるプログラムと、変数が変更不可能な言語で書かれたプログラムをデバッグすることを想像して下さい。例えばある変数の不変条件をチェックするアサーションが失敗した時、その変数がどうしてその値になったのか調べる時に、1箇所定義部分だけを確認するのと変更している部分全てを見てなぜその値になったのか考えるのはどちらが楽でしょうか。

文献では“Effective Java”などでも変更不可能なオブジェクト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 Yegor Bugayenko の “Objects Should Be Immutable”² 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** (*tuple*) が現れました。タプルというのは組み型と呼ばれ、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。ちなみに `int * int * real` と `int * (int * real)` は違う型なので注意して下さい。前者は3つの要素が入ったタプル型ですが、後者は2つの要素が入ったタプル型で2番目の要素がタプル型と読むことが出来ます。

さて、タプルを作る方法はただ `(1, "2")` などと書くだけです。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、スコープの話をしてしまおう。ソースコード 5 を見て下さい。SML ではレキシカルスコープ (*lexical scope*) 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 5: 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
```

²<http://www.yegor256.com/2014/06/09/objects-should-be-immutable.html>

```
(* 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
```

コラム：識別子に使える文字

あまり意識する必要はありませんが、SMLの識別子には2つの区別があります。

- アルファベット英数 (*alphanumeric*)
文字 [a-z,A-Z] かプライム (') で始まり、2文字目以降が文字 [a-z,A-Z]・数字 [0-9]・プライム (')・アンダーバー (_) であるもの
- シンボリック (*symbolic*)
以下のシンボルのみで作られた列 (# | など基本機能と被るものを除く)
! % & \$ # + - / : < = > ? @ \ ~ ' ^ | *

シンボリックルールで作る識別子を定義するには **op** が必要です。

この識別子は変数・関数・モジュール名・シグニチャ名で共通です。**structure ## = struct ... end** などと出来るので、識別子を見ても例えばそれがモジュール名なのかコンストラクタなのか区別する事は見た目だけでは出来ません。

ヒント：関数の結合の強さ

関数 *f* があったときに、

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 (*pattern match*) を用います。具体例を見ていきましょう。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
```

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きこちらのほうがより一般的です。例えば **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。

タプルの中身の要素を取り出したい、でもタプル自体にも名前をつけてアクセスできるようにしたい場合には *as* パターンを使います。ソースコード7を見て下さい。

ソースコード 7: as パターン

```
- val t1 as (a, b) = (1 + 2, 3 * 4);
val t1 = (3,12) : int * int
```

```
val a = 3 : int
val b = 12 : int
```

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がミスマッチだと言っていますよね。

また、データの一部を名前を付ける必要がないときには、**val** (_, x) = (1, 2) のようにワイルドカード (_) を使用できます。

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（アトムックパターン）
          | <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 :=    ...                                  % ワイルドカード
          | 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
```

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！`val 1 = 2`みたいな定数パターンのパターンマッチで、正しいコードです。（ただし実行時エラーになります）。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。ほぼすべてのパターンのケースを用いて複雑なパターンマッチを書いてみました。

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 _, B : t] A 2, {r1,...}) =
  ([B, [] A 1, B] A 2, {r1=1,r2="fuga",r3=[]});
val x = B : t
val y = [] : t list
val r1 = 1 : 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、定義の中にその補助のための定義を書けるようにしましょう。

こ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、ユーザに公開する関数はグラフ構造を壊さない関数のみにしたいですね。処理の途中でのみ使う、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安全性のために利用者が使えないようにしたいです。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、より細かい単位である式の単位でローカルな定義ができればより安全性が高まると思いませんか。それだけではなく、公開する関数と並べて書くのではなく外に公開しない形で書くことによりこの関数・値はこの式の中でしか使わないんだな、といったプログラムの意図をコードの中に表しやすくなります。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（Lisp 系言語の let 式と殆ど一緒です）。**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。let 式はれっきとした式です。なので他の 1 や "Himawari" ^ "Sakurako" 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

(* 変数名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
```

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「**un** 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。

関数定義に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ていることに注意して下さい。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³は知っているため、`x`、`y`の型は `int` 型であると推論されます。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。

関数の定義の中に `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、この宣言を `fun` 文の中にも書くことも出来ます。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 型注釈

型推論によって SML では多くの場合型を書く必要がありません。しかし推論結果が理解できない、なぜ型エラーになるのかわからない時などに型の注釈を書くことができます。一般に `13 : int` など、「式 : 型」と書きます。多くの場合 `val (x : int) = 13` などと注釈を書いたならば括弧で括る方がわかりづらいパースエラーが起こらないため安全です。

6.3 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** をつけると通常関数として扱えます。

³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。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、それを前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を前置記法でかけるようにする宣言 *)
- infix 4 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前置記法で使えるようになります (infix の後の数字は演算子の結合の強さを指定。省略可)。前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。

さて、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同様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を前置記法でかけるようにする宣言 *)
- infix **;
infix **
- 2 ** 10;
val it = 1024 : int
```

infix 宣言により前置記法で書くこととなった関数は `nonfix` により普通の記法で書かれるように出来ます。また infix 宣言は宣言であるため、let 式の宣言が並ぶ部分に書くことが出来ます。利便性のため一部分でだけで前置記法を用いることが出来るようになります。⁴

⁴つまり SML ではプログラム中で演算子の結合の強さが変わったり前置記法になったりそうでなくなったりするわ

6.4 パラメトリック多相性

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 とそのタプル) 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わからない変数の型を決定すると説明しました。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のでしょうか？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 (instantiation) 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 (polymorphic function) 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
```

`'a` と `'b` は違う型変数であることに気をつけて下さい。もし `fst: 'a * 'a -> 'a` ならば、同じ型の要素が2つ入ったタプルしか受け取ることが出来ませんが、`fst: 'a * 'b -> 'a` なので、違う型の要素が2つ入ったタプルを受け取ることが出来ます。

このあたりの型の制約が書けることが、ただ C 言語の `void*` 型とは異なる部分です。`void*` に対しては型チェックをされませんが、型変数に対しては整合性が確認されます。

6.5 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 (recursive function) 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
```

けです。パーサーがどうなっているのか想像してみてください。SML Kit の Parsing の論文には "Syntactacally, SML is a nightmare"[3] と書かれています。

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 $n = 1$ 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。この考え方をういて後で何回も再帰関数の説明をします。

6.6 高階関数

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)` 型であるの、2つの引数を取りその引数を足す関数 `plus` を定義しています。これはプログラム中の `int -> int -> int` はカッコを省略した形式です。今後省略できる括弧はすべて省略していきます。この型に対して今まで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。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、一つ引数を取り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れる事が出来ます。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`

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

型の関数を引数にとり、それを2度使います。

このように関数がある他の `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。また関数を返す関数や、関数を実行する関数を指して「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、 $f' = \lim_{a \rightarrow 0} \frac{f(x+a) - f(x)}{a}$ 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。

ソースコード 22: 高階関数の例

```
- fun diff f x = let val a = 0.0000000001 in (f (x + a) - f x) / a end;
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。

6.7 local 文

関数定義のために補助的な定義を書きたいことは沢山あります。しかし `let` 文がネストするのは避けたい場合というのはあると思いますが、`local` 文を用いると解決します。`local` 文は「`local` 宣言 11 宣言 12 ... `in` 宣言 21 宣言 22 ... `end`」という文法です。`local` から `in` までの定義は `in` 以降からはアクセスできますが、外側からアクセスすることは出来ません。

ソースコード 23: local 文

```
- local
  val a = 0.0000000001
  fun diff f x = (f (x + a) - f x) / a
in
  fun myCos r = diff Math.sin r
  val f' = diff (fn x => x * x)
end;
val myCos = fn : real -> real
val f' = fn : real -> real
```

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。しかし、ソースコード 21 の関数 `plus` は `int -> int -> int` 型の関数であると推論されています。`real -> 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 -> int` 型がつくことは少し変です。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 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` などとそれを組み合わせたな具体型」だけです。そのため、SML ではプラス演算子のようにアドホックに演算子が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、「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。加算、減算などが可能な値の型は `Num` に属するようになっています。

ソースコード 24: 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#^[2] 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 25: 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`, `IntInf.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

リストは変更不可能なデータ構造で、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 26 を見て下さい。

ソースコード 26: 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
```


'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 27: 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 28: 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。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「入力为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書けるならば便利そうです。

そのために case 式を用います。case 式は「**case** 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 29 を見て下さい。

ソースコード 29: case 式

```
- case [1,2,3] 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。`case` 式の方岐を書くとき、順番に気をつける必要があります。例えば `case 1 of n => 0 | 1 => 1` というプログラムがあったときに整数は変数パターン `n` に必ずマッチするのでこのプログラムは絶対に 2 つ目の分岐部分に到達することはありません。書く順番が逆ならそんなことはありません。この場合対話環境が *Error: match redundant* とパターンマッチが冗長だ、と警告を出すはずです。

さて、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

パターンマッチでは、すべての場合を網羅できていないと処理系が警告します。例えば `case 1 of 1 => "Fuso" | 2 => "Yamashiro"` のように `int` 型の値をパターンマッチしてみると、`stdIn:16.1-16.26 Warning: match nonexhaustive` というメッセージが出るはずです。これはパターンの列挙に漏れがあることを警告しています。この警告は `case` 式を使った場合だけではなくあらゆるパターンマッチを行った時に機能します。

警告が出た時に修正が面倒だったり理由が分からなければ `any` パターンを用いることが出来ます。例えば `case 1 of _ => 1` のようにアンダーバーを用いればどんなパターンにもヒットします。しかし `any` パターンは何でもヒットしてしまうためにパターンマッチの網羅性検査を事実上機能させなくしてしまうので、多用するのは避けるべきです。

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 30: 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、空リストは空リストの逆順になっている
- 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⁶。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 31 を見てください。

⁶簡単のためにこのような実装になっていますが実はこれは推奨されない実装です。なぜならば@演算子はリストを全部操作するためコストが大きいためです。

`fun rev' acc [] = acc | rev' acc (x::xs) = rev' (x::acc) xs; fun rev l = rev' [] l` などと定義するほうが遥かに効率が良いです

ソースコード 31: 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空リストを返し、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 30 と 31 は挙動は全く一緒なので、好みで使い分けて下さい。この文章でも書きやすい方を選んでその都度違うスタイルを取ります。

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。⁷

ソースコード 32: 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 [] = []  
  | map f (x :: xs) = f x :: map f xs;  
val map = fn : ('a -> 'b) -> 'a list -> 'b list  
  
(* mapを試してみる *)  
- 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 足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だと思います。

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、`fn` 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⁸。

ソースコード 33: 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
```

匿名関数としてカーリー化されている、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あります。また匿名関数を使う例は次の節にたくさん現れますので確認してみてください。

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。テーブル 1 を見て下さい。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 [1]。ちなみに述語とは、`bool` 型の要素を返す 1 引数関数のことです。

⁷他の言語では `reduce`, `select` と呼ばれることもあります。

⁸実は `fun` 文は `val funName = fn pat1 => pat2 => ...` のシンタックスシュガーです

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
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
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 の結果で要素を 2 つに分ける
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

Table 1: List モジュールの主な関数

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 34: 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]];
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
    (* String.explode: string -> char list を使う *)
    val chars = String.explode str

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

コラム：ドキュメントとしての型

テーブル 1 を見てると、型がドキュメントとして有効に働いていると分かります。例えば `List.partition` の型は `('a -> bool) -> 'a list -> 'a list * 'a list` ですが、これと名前を見れば、述語を受け取って、述語が `true` になる要素と `false` になる要素を 2 つに分ける関数だということが分かります。

多くの ML 系言語はマイナーなので、あまりライブラリのドキュメントが整っていません。そのため関数の名前とその型が最大にして唯一のドキュメントであることはままあります。そのためユーザは強くあらねばなりません……。そこはいいところでもあり、悪いところでもあるのだと思います。

コラム：foldl と foldr

`List.foldl` は非常にプリミティブなリスト操作関数です。まずは再帰関数でリスト操作を考えるよりも、`foldl` で実装できないか考えるべきです。`foldl` を使って `List` モジュールに含まれる関数を実装してみましょう。なぜ正しく動くのか考えてみることは良い練習問題になります！ぜひ考えてみて下さい！

`foldl`、`foldr` に渡す関数は概ね `fn (x, acc) => ...` という形になります。`acc` は所謂アキュムレータで、今までの計算結果を引数として受け取るときによく用いられる名前です。つまり畳み込み関数に渡す関数は今処理する要素 `x` と今までの処理結果 `acc` をどう処理して `acc` と同じ型の要素を返すのか記述したものになっていなければいけません (コードから読み取れるでしょうか?)。 `fn (x, acc) => x::acc` を渡して初期値のリストに次々要素を繋げていけば最終的に逆順になりますよね (関数 `rev`)。 `fn (x, acc) => f x::acc` のように、関数 `f` を要素に適用してその結果を繋げていけば関数 `map` の出来上がりです。ただし、左の要素 (先頭の方) から処理して `acc` の先頭に足していくと自然に逆順になるので逆から処理していく必要があります。`foldl` と `foldr` の違いは先頭から処理していくか最後から処理していくのかの違いです。そのため下の `map` では `foldr` を使っています。

```
- fun foldl f acc [] = acc
  | foldl f acc (x :: xs) = foldl f (f (x, acc)) xs;
val foldl = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

- fun foldr f acc [] = acc
  | foldr f acc (x :: xs) = f (x, (foldr f acc xs));
val foldr = fn : ('a * 'b -> 'b) -> 'b -> 'a list -> 'b

- fun rev l = foldl (fn (x, acc) => x::acc) [] l;
val rev = fn : 'a list -> 'a list

- fun map f l = (List.foldr (fn (x, acc) => f x :: acc) [] l);
val map = fn : ('a -> 'b) -> 'a list -> 'b list

- fun forall f l = foldl (fn (x, acc) => f x andalso acc) true l;
val forall = fn : ('a -> bool) -> 'a list -> bool
```

8 レコード

構造体のような、複数の要素をまとめる方法をこの章では学びましょう。タプルは複数の値を 1 つにまとめる手段として既に登場しましたが、アクセスするときに順番 (例えば `(a, b, c)` という形のパターンなら `a` は 1 番目) でそれぞれの要素を区別していました。これは、たくさんの種類の構造が入り乱れるときには不便です。構造体の要素にアクセスするときのような、意味のある名前アクセスしたいですね。

そのような構造を SML ではレコード (*record*) が提供してくれます。ここではなぜ構造を作ると便利なのかという議論は、他の言語と同様なのでせず SML のレコードの特徴・使い方のみに注目します。早速例を見てみましょう。ソースコード 35 を見て下さい。

ソースコード 35: レコードの例

```
(* レコードを作る *)
- val r = {x=1, y=3.14};
val r = {x=1,y=3.14} : {x:int, y:real}

(* #フィールド名 を用いた、フィールドアクセス関数による要素取り出し *)
- #x r;
val it = 1 : int

(* パターンマッチによる要素取り出し、asパターンも使える *)
- val r' as {x=num, y=float} = r;
val r' = {x=1,y=3.14} : {x:int, y:real}
val num = 1 : int
val float = 3.14 : real

(* フィールドxの要素のみ取り出す時にワイルドカードパターンが使える *)
- val {x=num,...} = r;
val num = 1 : int

(* round: real -> int は四捨五入してintにキャストする関数 *)
- fun plus {x=num,y=float} = num + round float;
val plus = fn : {x:int, y:real} -> int
```

レコードは「{フィールド名 1=式 1, フィールド名 2=式 2, ...}」という形で書かれます。レコードから要素を取り出す方法は主に2つです。1つは#フィールド名 レコード という形でフィールドアクセス関数を経由して取り出す方法があり、もうひとつは val 文・fun 文におけるパターンマッチによる要素取り出しがあります。パターンマッチで取り出す際は、パターンには{フィールド名 1 = 束縛変数名 1, フィールド名 2 = 束縛変数名 2, ...}という風に書かれます。複数の要素があるレコードから一部のフィールド名の要素を取り出す時、...(ワイルドカードパターン)を使うことが出来ます。例えば上の例のように x, y のフィールドが存在するレコードに対して、フィールド x の要素のみを取り出す時、**val {x=num,...}**と書いて他のフィールドを省略できます。

レコード型の型はソースコード 35 の plus 関数のように、{フィールド名 1:型 1, フィールド名 2:型 2,...} という形です。これは直感的に分かりやすいと思います。型・レコードの束縛・パターンマッチにおいて、フィールドの順番はどんな順番でも構いません。

もしフィールド名=束縛変数名ならば、イコール以降は省略することが出来ます。ソースコード 36 を見て下さい。

ソースコード 36: レコードのパターンマッチ時の変数名の省略

```
- fun plus {x, y} = x + round y;
val plus = fn : {x:int, y:real} -> int
```

SMLにおいてレコードは特に宣言もなく使えますが、この扱いは気をつける点があります。それはレコードの型は、すべてのフィールド名が明らかになっていなければならないことです。例を見てみましょう。ソースコード 37 を見て下さい。

関数 plus' はレコード r を受け取り、フィールドアクセス関数 #x と #y を用いて要素を取り出しソースコード 35 の plus 関数と同じく y をキャストしてから x と加算しています。しかしこれはコンパイルが通りません。これはフィールド x, y がレコード r に含まれることは分かるけども、それ以外のフィールドが r に存在するかもしれず r の型を確定できないからです。

このコードままコンパイルを通すためには型注釈を書けば良いです。plus'' を見て下さい。関数 idX はレコードを受け取り、その中に含まれるフィールド x の要素を返す関数です。これも関数 plus' と同じ理由でコンパイルが通りません。idX' の定義のように、フィールド名をすべて列挙すれば型が決まるためコンパイルできます。

また、#から始まるフィールドアクセス関数はれっきとした関数で、他の式と組み合わせることが出来ます。あまり困ることはないと思いますが、#x のようにそれ単体で評価しても型が決まりません。文脈で決まる場合は大丈夫ですが、そうでない場合はレコードやアクセス関数に型注釈を

書けば通ります。

ソースコード 37: フィールド名が解決出来ない例

```
- fun plus' r = #x r + round (#y r);
stdIn:25.1-25.34 Error: unresolved flex record (need to know the names of ALL the fields
in this context)
  type: {x:int, y:real; 'Z}

(* rの型を書けば通る *)
- fun plus'' (r: {x:int,y:real}) = #x r + round (#y r);
val plus'' = fn : {x:int, y:real} -> int

- fun idX {x,...} = x;
stdIn:1.2-10.5 Error: unresolved flex record
  (can't tell what fields there are besides #x)

(* フィールド名をすべて列挙すれば型が決まる *)
- fun idX' {x, y, z} = x;
val idX' = fn : {x:'a, y:'b, z:'c} -> 'a
```

最後に、タプルはレコードの表現のシンタックスシュガーであることを紹介します。タプルは、レコードのフィールド名が1,2,... と整数で表されたただのレコードです。そのため、フィールドアクセス関数を利用できます。例えばタプルの第一要素を取り出したければ#1 を用いればいいわけです。

ソースコード 38: タプルはレコード

```
- {1="hoge", 2="null"};;
val it = ("hoge", "null") : string * string

- #1 ("hoge", "null");
val it = "hoge" : string
```

8.1 型のエイリアス

レコード型は長いため、もし型を何回も書くとするとても不便です。型に名前を付けるためにtype文を使いましょう。ソースコード 39 を見て下さい。type文は「**type** <パラメータ> 型名 = 型」という構文です。

ソースコード 39: 型のエイリアス

```
- type t = {x:int, y:real};
type t = {x:int, y:real}
- val r : t = {x=12, y=3.14};
val r = {x=12,y=3.14} : t
```

後述の datatype 文とは異なることに注意して下さい。datatype 文は新しい、主にヴァリアント型を作るときに用いますが type 文は主に型のエイリアスを作るときに用います。

8章で挙げた問題は、私はコードを書いていると非常に不便に感じます。これを解決する方法は主に2つあります。

一つはレコードを自由に使えることを放棄することです。ML系言語のOCaml[4]では、レコードは必ず宣言してから使います。そしてレコードの要素へのアクセスのためのアクセス関数は存在せず、レコードアクセスのためにドット記号を用いてレコード.フィールド名のように書きます。フィールド名は、すべてレコード型を利用する前にすべて宣言されているおかげでフィールド名を見ればどのレコード型のフィールド名なのか区別がつきます。

例を見てみましょう。ソースコード 40 を見て下さい。

ソースコード 40: OCaml におけるレコード

```
(* レコードは予め宣言する *)
type t1 = {x:int; y:float};;

(* ドット(.)でレコードのフィールドへアクセス
   OCamlではletで関数も変数も宣言する *)
let plus r = r.x + int_of_float r.y;;
# val plus : t1 -> int = <fun>
```

この方式では複数のレコード型で同じフィールド名を使いまわすことが面倒ですが、私見ではSMLの方式よりも面倒な事を防ぐことが出来ると思います。

もうひとつはSML#[2]で実装されている多相レコードです。多相レコードでは、8章で議論した「レコード型のフィールドはすべて明らかにならなければならない」という制約はなくなり、型が与える情報はレコードが持つべき最低限のフィールドとその型をあたえるものになっています。ソースコード 41 では、関数 *f* の型をみると、「型変数 'b、ただし 'b はレコード型で少なくとも 'a 型のフィールド *name* は含まれる」という制約を書き下しています。

ソースコード 41: SML# の多相レコード

```
# fun f x = #name x;
val f = fn : ['a, 'b#{name:'a}. 'b -> 'a]
# f {name = "Joe", age = 21};
val it = "Joe" : string
```

OCaml のオブジェクトも同じ事が出来ます。このあたりの歴史的背景はSML#プロジェクトの記事^aを見て下さい。

ソースコード 42: OCaml のオブジェクト

```
utop[0]> let f x = x # name;;
val f : < name : 'a; .. > -> 'a = <fun>
utop[1]> f (object method name = "joe" method age = 21 end);;
- : bytes = "joe"
```

^a<http://www.pllab.riec.tohoku.ac.jp/smlsharp/ja/?Foundations%2F010>

9 ユーザ定義データ型

9.1 ヴァリアントの定義

今まではレコード・タプルなど複数の値を1つにまとめる方法を学びました。ここでは新しく複数の型の値を1種類の型にまとめる、**ヴァリアント** (*variant*) の使い方を学びましょう。

早速例を見てみましょう。ソースコード 43 を見て下さい。

ソースコード 43: 単純なヴァリアント

```
(* t型を新しくつくる *)
- datatype t = A of int | B of real | C of string;
```



```
datatype t = A of int | B of real | C of string
```

```
(* t型のリストを作る *)
- val l = [A 12, B 3.14, C "ruri"];
val l = [A 12,B 3.14,C "ruri"] : t list

(* t型の要素をstringに変換する関数を作る *)
- fun toString t = case t of
    A i => Int.toString i
  | B f => Real.toString f
  | C s => s;
val toString = fn : t -> string

(* toStringを使ってみよう *)
- List.map toString l;
val it = ["12","3.14","ruri"] : string list
```

この例ではまず **datatype** 宣言により新しい型 *t* を作っています。この書き方は **datatype** <パラメータ> 型名 = コンストラクタ名1 **of** 要素の型1 | コンストラクタ名2 **of** 要素の型2 | ...」です。<パラメータ>の部分はオプションで、*list* のように型の一部をパラメータ化するために使います。宣言の際に列挙した、ヴァリエントを作るための関数(上の例ではA,B,Cのこと)をコンストラクタと呼びます。またヴァリエントの要素の型が *unit* 型である場合、この **of** 以降を省略できます。

t 型の要素を作る方法は「コンストラクタ 要素」と並べるだけです。コンストラクタは実は関数なので、ただの関数適用です。例えばソースコード 43 では *A 12* と並べることで *t* 型の値を作っています。

次にヴァリエントを分解するためにパターンマッチを使います。上の例で *toString* 関数の定義を見て下さい。この定義で、**case t of A i => ...** の部分がもし引数が *A* のケースであった場合の処理です。*t* が *A* のケースの場合、中身を *i* と名前をつけて処理を行います(リストを扱った時と変わりありません)。

ヴァリエントは、「要素にラベルがついたもの」です。これにより複数の異なる型の値を実行時に区別し、安全に混ぜることが出来るようになっています。新しく作る型の要素が宣言の時に列挙されているという意味で定義の見た目は似ていても、C や Java の Enum とは扱いが全く異なります。

ヴァリエントはあまり ML 系言語以外には見られない機能です。これができると何故嬉しいのか詳しく見て行きましょう。ソースコード 44 では、標準ライブラリで提供されている '*a option* '型を使用しています。*option* は *list* と同様にそれ単体では型ではなく、*int option* や *string option* などパラメータを持つことで初めて型として扱われます。型パラメータを持つヴァリエントを宣言するためには、**type 'a t = C of 'a** などと **type** の後にパラメータを書きます。

ソースコード 44: オプション型

```
(* option型の定義は
   datatype 'a option = NONE | SOME of 'a *)

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

(* 3で割ってあまりが0になる要素を取り出す *)
List.find (fn x => x mod 3 = 0) l;
val it = SOME 10 : int option

(* 100以上の要素を取り出すが、lには存在しないのでNONEが返ってくる *)
- List.find (fn x => x >= 100) l;
val it = NONE : int option
```

上の例ではまず *int list* 型の変数 *l* をつくり、*List.find*: ('a -> bool) -> 'a list -> 'a option に匿名関数とこのリストを渡しています。'*a option* '型の要素は2つあります。一つは要素が存在

しないことを表す **NONE**、もうひとつは要素が存在することを表す **SOME of 'a** です。List.find を使った最初の例では、3 で割ってあまりが 0 になる最初の要素を取り出していますが、これはこのような要素が 1 に存在するので **SOME 3** を返しています。一方でその下の例では、100 以上の要素を探して取り出していますが、そのような要素は存在しないので、存在しなかったことを表すために **NONE** が返ってきます。

このように、計算が失敗するかもしれないという事を手軽に、さらに型にもそれを表明することが出来ます。プログラマは例えば List.find の返り値部分を見て 'a ではなく 'a option であることを見れば、ああ、この処理というのは失敗する場合もあるんだなということが分かります。また上の例で int option 型の値を使うためには、必ずパターンマッチにより SOME のケースと NONE のケースを記述しなければなりません。ここが重要なのですが、書き漏らしたケースがあれば「コンパイラが警告します」。ここが他の失敗をエンコードする手段と全く違うところです。失敗の場合のケースを書き漏しを機械がチェック出来るという部分がオプション型の良い部分です。

オプション型以外にも、例えば成功した時 'a 型の要素を返し、失敗した時 'b 型の要素を返す ('a, 'b) result 型のようなものも考えられます。このような構成要素をユーザが作れる、それがヴァリエントの嬉しい部分です。

コラム：ヴァリエント無しで失敗をエンコードするには

ヴァリエントが存在しない言語ではこのような失敗するかもしれない要素はどのように表せるでしょうか？考えられる可能性としては、

1. null など計算に失敗した事を表す特別な定数を返す
2. 例外を計算が失敗した時に投げる
3. オブジェクトのサブタイピングを用いて SOME, NONE のスーパータイプに該当するようなインターフェースを作り、それを継承する形で SOME, NONE を実装する

1. の null を用いる場合だと、コードのあちこちで null チェックが必要になります。この悲惨さは私よりも職業プログラマの皆様のほうがよくご存知だと思います。

2. の失敗した時に例外を投げることにして、List.find を例にすると返り値を 'a option ではなく 'a にしてしまうのは null を用いるよりはあり得る設計です。しかし、例外の存在はプログラムを危険にします。プログラマが例外をキャッチすることを忘れれば、プログラムはキャッチされない例外によって終了してしまいます。そしてキャッチし忘れをコンパイル時に h 発見することは中々難しいです

3. の同じような解決策は Play フレームワークの Java インターフェースにも実装されています。しかし、SML ではオプション型の定義は 1 行で書けますが、Java のこの実装では冗長なたくさんのコードにより実装されているようです^a。もちろんユーザとして利用するだけならパターンマッチできない等の制限はありますが便利に使えんと思います。

^a<https://www.playframework.com/documentation/2.0/api/java/play/libs/F.Option.html>

9.2 再帰的ヴァリエント型

コンストラクタの要素として、自分自身と同じ型の要素を持つようなヴァリエント型のことを再帰的ヴァリエント型と呼びましょう。これはすでに登場しています。リストの定義を考えてみましょう。リストの定義は List モジュールをオープンしてみるとプリントされます。モジュールをオープンする、とはモジュールの関数や変数にはドット記号をつけてアクセスしていましたが、モジュール名を省略できるようになります。例えば open List; のあとでは List.find ではなく find とかければこの関数を指すようになります。

ソースコード 45: リストの定義を確認する

```
- open List;
opening List
datatype 'a list = :: of 'a * 'a list | nil
exception Empty
```

```

val null : 'a list -> bool
... 以下省略 ...
- op ::;
val it = fn : 'a * 'a list -> 'a list

```

これを見るとリストは **datatype** `'a list = :: of 'a * 'a list | nil` と定義されていますね。リストというのはコンス (`::`) の場合と空リストの場合から成り立っています。`::` の場合の要素の部分を見てみると、今 `'a list` が存在しないから作ろうとしているのに、既に `'a list` が存在しているかのようにこれを使っています。再帰の構造をしていますね。

ちなみにコンスのケースで要素の型に含まれる型変数が `'b` ではなく `'a` だということも重要な情報です。`'a list` の要素の型はすべて `'a` 型である、他の型は含まれないということを表明しているわけです。`'a` 以外の型変数、例えば `'b` を使ってしまうとそんなものはバインドされない！と怒られてしまうわけですが。

最後に応用例として 2 分探索木を利用した辞書を実装してみましょう。ソースコード 46 を見て下さい。ここでは対話環境の応答やプロンプトは省略しています。

ソースコード 46: 2 分探索木による辞書の実装

```

(* keyとして文字列でアクセスし、'a型の要素を取り出す辞書の実装
破壊的変更はせずに辞書に要素を追加したら、新しい辞書を返すようにする *)
datatype 'a dict =
  LEAF
| NODE of {left: 'a dict, key:string, value: 'a, right: 'a dict}
val empty = LEAF

(* insert: string -> 'a -> 'a dict -> 'a dict *)
fun insert k v LEAF = NODE {left= LEAF, key= k, value= v, right= LEAF}
  | insert k v (NODE {left, right, key, value}) =
    if String.< (k, key)
    then NODE {left= insert k v left, right=right, key= key, value= value}
    else NODE {left= left, right= insert k v right, key= key, value= value}

(* find: string -> 'a dict -> 'a option *)
fun find k LEAF = NONE
  | find k (NODE {left, right, key, value}) =
    if k = key then SOME value else
    if String.< (k, key) then find k left else find k right

(* toString: (string -> 'a -> string) -> 'a dict -> string *)
fun toString f LEAF = "LEAF"
  | toString f (NODE {left, right, key, value}) =
    "(" ^ toString f left ^ ", " ^ f key value ^ ", " ^ toString f right ^ ")"

(* printDict: (string -> 'a -> string) -> 'a dict -> unit *)
fun printDict f dict = print (toString f dict)

```

気をつける点は多相的なイコールを多用していることです。イコールの型付けは特殊で、`'a * 'a -> bool` という型を持ちます⁹。色々な型に対して等しいかどうかを判定できる、ユーザが書けない関数です。

ソースコード 47: イコールを乱用する

```

- (1, 2) = (3, 4);
val it = false : bool
- {x=12, y=13} = {x=12, y=13};
val it = true : bool
- ref 0 = ref 0; (* 参照の場合はアドレスが等しいかどうかのチェック (後の章参照) *)
val it = false : bool

```

⁹ちなみに `'a` は特殊な型変数で、浮動小数点数型や関数型に単一化できない型変数です。浮動小数点数の等しさの判定はしちゃいけないし、関数同士の比較は普通は出来ませんよね。

また `string` 型の比較を行う `String.<`関数を利用しています。これは `infix` 宣言されていないので普通の関数と同じスタイルで関数適用しています。また、一部対話環境がネストが深かったりデータが長すぎて省略している部分がありますね。

ソースコード 48: dict 型の利用

```
(* パイプライン演算子 次のヒントコラム参照 *)
- fun op |> (x, f) = f x;
val |> = fn : 'a * ('a -> 'b) -> 'b
- infix 1 |>
infix 1 |>

- val dic = insert "hibiki" 1 LEAF
              |> insert "sakimori" 2
              |> insert "kanade" 3
              |> insert "chris" 4
              |> insert "shirabe" 5
              |> insert "maria" 6;

NODE
  {key="hibiki",left=NODE {key="chris",left=LEAF,right=LEAF,value=4},
   right=NODE {key="sakimori",left=NODE #,right=NODE #,value=2},value=1}
: int dict

- find "chris" dic;
val it = SOME 4 : int option

- find "sugita" dic;
val it = NONE : int option
```

ヒント：パイプライン演算子

ソースコード 48 でパイプライン演算子を定義して使用しました。これは関数なのですが一種の制御構造みたいな役割を果たす、多用される演算子なので紹介します。

`fun op |> (x, f) = f x` がパイプライン演算子 `|>` の定義です。次の行の `infix` 宣言により中置演算子として扱われます（結合の強さは弱い `1` を選びました）。定義を見ると、ふつう `f x` とかかる関数適用を逆順に書くためのオペレータなのだということが分かります。これでパイプライン演算子がどのように働くのかというと、例えば `x |> f |> g |> h` というコードは `h (g (f x))` というコードと同じ挙動をします。

パイプライン演算子が並んでいる部分は、計算した結果を次のパイプラインの末尾の部分に渡して処理して、それをさらに次のパイプラインに渡していく、と理解することが出来ます。下の2つのソースコードはどちらが読みやすいでしょうか？

```
insert "hibiki" 1 LEAF
|> insert "sakimori" 2
|> insert "kanade" 3
...
```

```
... (insert "kanade" 3 (insert "sakimori" 2 (insert "hibiki" 1 LEAF))) ...
(* 更に外側にネストする *)
```

SMLではヴァリアントはすべて予め宣言されなければ使えませんでした。また後で定義を拡張してヴァリアントを増やしたり、コンストラクタの名前を他のヴァリアント型のコンストラクタの名前と同じに出来ませんでした。OCaml[4]には宣言せずに使える、多相ヴァリアントというものがあります。多相ヴァリアントは'が最初につき、大文字で始まります。例を見てみましょう。

ソースコード 49: 多相ヴァリアント

```
utop[0]> let f x = match x with 'A y -> y | 'B -> 2 | _ -> 3;;
val f : [> 'A of int | 'B ] -> int = <fun>

utop[25]> let g x = match x with 'A y -> y | 'B -> 2;;
val g : [< 'A of int | 'B ] -> int = <fun>
```

多相ヴァリアントの型は少し特殊です。まず関数 `f` の型を見てみると、`[> 'A of int | 'B]` というのは多相ヴァリアントを表す型で、少なくとも `'A of int` と `'B` というヴァリアントを含む多相ヴァリアント型ということを表しています。関数 `g` に現れる `[< 'A of int | 'B]` はその逆で、高々 `'A of int` か `'B` というヴァリアントしか含まない多相ヴァリアント型だということを表しています。型の大小記号 (`>` と `<`) が異なっている部分に注目して下さい。

ちなみに `let f x = if x = 0 then 'A "hoge" else 'A 13` のように多相ヴァリアントの要素の型が異なるものを混ぜることは出来ません。

多相ヴァリアントを解説すると沢山記事が書いてしまうのですが、面白い挙動の一例として `'Cons` と `'Nil` を用いてリスト構造を作ってみます。

ソースコード 50: 多相ヴァリアントを用いたリスト構造 (ocaml コード)

10 例外

SMLには他の言語と同じような例外機構があります。例外の作成には **exception** 例外名 <of 型>、例外を投げるときには **raise** 例外名、それをキャッチするためには 式 **handle** パターン1 => 式1 | パターン2 => 式2 | ... という構文を用います。

例外の型付けは少し特別です。作られた例外はすべて `exn` 型を持ちます。¹⁰ また **raise** 例外は多相型 `'a` を持ちます。例外が投げられれば処理を続けることもないので、任意の型をもって問題ありません。

ソースコード 51: 例外の型付け

```
- exception Exn of int;
exception Exn of int
- Exn 100;
val it = Exn(-) : exn
- fn () => raise Exn 1;
val it = fn : unit -> 'a
```

ソースコード 52: 例外を用いたプログラム例

```
exception Break
(* int listを受け取ってすべて掛け合わせる関数
途中に0があれば例外を投げて大域脱出 *)
fun mul l =
  let fun mul' [] = 1
        | mul' (0 :: _) = raise Break
        | mul' (x :: xs) = x * mul' xs
  in mul' l handle Break => 0 end
```

11 手続きプログラミング

ここまでの章では、プリントを除きピュアな（破壊的代入などの副作用がない）要素だけを扱ってきました。SMLにおける副作用のある計算を行う仕組みを見て行きましょう。他の言事対して変わらないのでさらっといきます。

主にここで出てくるのは、返り値に `unit` 型の値を返す関数や式です。`unit` 型はヴァリアントで、値が `()` 1つしかない型です。`unit` 型はC言語の `void` 型のように、プリントや破壊的代入など副作用を主に期待している処理が返すものがない時に返す事が殆どです。型を解釈するときに、返り値が `unit` 型になっていれば「この返り値には意味はなくて、副作用が重要なのだな」と考えて下さい。

```
- ();
val it = () : unit
(* 対話環境ではいらないですが、宣言列のなかで副作用だけを
起こしたいときには定数パターンのパターンマッチを使います *)
- val () = print "hello, impure world !\n";
hello, impure world !
```

11.1 逐次実行

順番に処理を実行していくことを逐次実行と呼びます。SMLの逐次実行は「(式1; 式2; ...)」という構文です。逐次実行の式は式1を評価して、その返り値を捨てて次に式2を評価する、というのを続け最後に評価した値を返します。必ず括弧で括らないと行けないことに気をつけて下さい。

ソースコード 53: 逐次実行

```
- fun p () = (print "1"; print "2"; print "3"; print "\n");
val p = fn : unit -> unit
- p ();
123
val it = () : unit
```

¹⁰例外型を悪用(?)すると面白いことが出来、実行時に型によって分岐するプログラムなんかも書けてしまいます[5]!

逐次実行は let 式の **let** から **in** までの宣言列でも **val () = ...** を用いて行うことができます。また **in** から **end** までに逐次実行を書く場合のみ括弧が必要ありません。

```
let val a = 12
    val () = print "rakudai"
    val b = "*"
in print 1; print 2 end
```

11.2 参照型

SML では変数は基本的にイミュータブルでした。多くの場合、再帰関数などでループ処理を行えばあまり困ることはないのですが、効率のために変更できる変数を使いたくなる時があります。そのために参照型と呼ばれる型の扱い方を習得しましょう。例をみれば使い方がわかると思うので、詳細はソースコード 54 を見て下さい。

ソースコード 54: ref 型

```
(* 参照型を作るのにrefを用いる *)
- val r = ref 0;
val r = ref 0 : int ref

- val r' = r;
val r' = ref 0 : int ref

(* 参照型から値を取り出すには!を使う *)
- op !;
val it = fn : 'a ref -> 'a

(* 代入には:=を使う op := : 'a ref * 'a -> unit *)
- r := !r + 1;
val it = () : unit

(* 書き換わってる! *)
- (!r, !r');
val it = (1,1) : int * int
```

ref 0 という値は、まさにメモリー上にセルを確保して 0 を書き込むプログラムです。**val r = ref 0** という文で、そのポインタを **r** に束縛していると思って下さい。**val r' = r** はポインタのコピーです。そのため **r'** が指す先は **r** と共通です。**r** を書き換えれば、**r'** の指す先の内容もアップデートされます。

11.3 繰り返し

あまり SML を書いていると使いたくなる事はないのですが、一応 while 式が存在します。while 式は unit を返します。大体は List.app や List.appi、Array.appi などと済みます。

ソースコード 55: while 式

```
- fun incr r = r := !r + 1;
val incr = fn : int ref -> unit
- let val r = ref 0 in
    while !r <= 10 do (print (Int.toString (!r)); incr r)
end;
012345678910val it = () : unit

- let val l = List.tabulate (11, (fn x => x)) in
    List.app (fn n => print (Int.toString n)) l end
012345678910val it = () : unit
```

11.4 配列

配列は多相的で `'a array` 型と表現されます。配列を扱うための特別な構文は存在せず、すべて関数を通してアクセスやアップデートをします。Array モジュールの詳細はマニュアルを見ましょう [1]。

ソースコード 56: 配列の扱い

```
(* 関数を使って int array をつくる *)
- val arr = Array.tabulate (100, (fn x => x + 1));
val arr = [|1,2,3,4,5,6,7,8,9,10,11,12,...|] : int array

(* 配列を書き換える *)
- Array.update (arr, 0, 200);
val it = () : unit

(* 配列にアクセスする。たしかに置き換わってますね *)
- Array.sub (arr, 0);
val it = 200 : int
```

11.5 値制約 (value restriction)

実は SML おいて多相的な型がつくケースは制限されています。不拡張な式 (Non-expansive expression, 値式と訳している文献もある) と呼ばれる式にだけ多相的な型がつきます。それは直感的には、例外・副作用を起こし得ないと明らかに分かる式のことです。より具体的には「定数、関数、コンストラクタの適用 (ref は除く) とそれらの組み合わせだけで作られるデータ構造」のことです。

最も値制約の害がある例は関数適用の結果が多相的にならないことが挙げられます。

ソースコード 57: 値制約

```
(* 上のルールにはコンストラクタでない関数適用の返り値は多相的にならない *)
- fun id x = x;
val id = fn : 'a -> 'a

- id id;
stdIn:3.1-3.6 Warning: type vars not generalized because of
  value restriction are instantiated to dummy types (X1,X2,...)
val it = fn : ?X1 -> ?X1
```

副作用のない言語 (例えば Haskell) ではこのような制約は必要ありません。副作用があるときこの制約が必要な場合を見てみましょう。ソースコード 58 を見て下さい。一行目のコードが多相的になるのは不正なコードで、他の行の型付けは正しいです。

ソースコード 58: 値制約が必要なケース (不正なコード、動かない)

```
val polyRef : ('a -> 'a) ref = ref (fn x => x)
val () = polyRef := (fn n => n * n)
val str = (!polyRef) "This operation is illegal !"
```

`polyRef` がもし多相的な型がつくとすると、一度決まった型が変わることはないので、当然型変数に `int` を代入しても (2 行目) `string` を代入しても (3 行目) 問題ないはずですが、しかし 3 行目で `!polyRef` の中身は `int -> int` の関数になっており、このコードは不正な動作をしてしまいます。原因は `polyRef` に多相的な型がつくことです。またこのような事が起こる事が分かると、関数適用の結果も多相的になってはいけなことが分かります。コンストラクタの適用は副作用は起こし得ないので許されていますが関数適用には副作用があるかもしれないためその返り値には多相型をもたせてはいけません。

ソースコード 59: 関数の例


```
- fun f () = ref NONE;
val f = fn : unit -> 'a option ref
- val r = f ();
stdIn:13.1-13.5 Warning: type vars not generalized because of
value restriction are instantiated to dummy types (X1,X2,...)
val it = ref NONE : ?.X1 option ref
```

実用的には、例えば `val duplicate = List.map (fn x => (x, x))` のようなケースで多相的になってほしいにも関わらず多相的にならない場合があります。多くの場合、 η 展開と呼ばれる手法で解決できます。この手法は、先程の例ならば `fun duplicate l = List.map (fn x => (x, x)) l` と右辺を関数にしてしまうことです（`fun` 文は `val` 文のシンタックスシュガーで、デシュガーすると `val duplicate = fn l => ...` になります）。 η 展開についてはラムダ計算の用語なので深追いしませんが、多相的にならないと困った時には少し表現を変えるだけで多相的な型がつくことが多いということは覚えておいて下さい。

コラム：手続きとの付き合い方

ML 系言語を書いていると中々手続きとの付き合いというのは切っても切れません。せっかく他のピュアな（副作用のない）言語ではなく手軽に副作用を書ける言語を使っているわけですから、必要な場面で使わない手はありません。そこで私見ですが手続きとの付き合い方について書いておきます。私の方針は以下の通りです。

1. 基本的には副作用は使わずにピュアに書く
2. 副作用で記述が簡潔になる・凄く高速化できる時は積極的に利用する
ただし、副作用が1つ（または少数）の関数で閉じており、外側からはピュアに見えるようにする
3. グローバルなテーブルでアルゴリズムが綺麗になるなら使うが、書き込む部分・読み込む部分を少なくする
基本的にはテーブルは `let` 式・`local` 文を用いて他からアクセスされないことを保証する

純粋性 (*purity*) はとても良い性質です。入力と同じであれば出力が同じになることが保証され、テストもしやすくなります。副作用を用いる時も、ある関数の外側から見ればこの性質が保たれるようにするべきです。

よくプログラムを書いているとグローバルなテーブルを用いて高速化したくなります。その場合は書き込み・読み込みが無節操に行われるとデバッグが地獄です。その場合スコープを制限する仕組みを用いてテーブルの影響範囲を限定しましょう。

ソースコード 60: `let` 式でリファレンスが他からアクセスされないことを保証する

```
val (reader, writer) =
  let val r = ref 0 in (fn () => !r, fn n => r := n) end
```

12 モジュール

この章では名前空間を区切るためのモジュールシステムについて説明します。SML にはオブジェクト指向プログラミング言語のようなクラスは存在ませんが、C++ の名前空間のような機能があります。違いはモジュールを受け取って新たなモジュールを返すファンクター (Functor) やシグニチャ (モジュールの型のようなもの)、シグニチャによる型隠蔽の存在です。モジュールシステムに関しては綺麗な理論がないため、少し理解しづらい・使いづらい部分があるかもしれません。SML で大規模開発をする上でモジュールの理解は不可欠です。他の言語と比較しながら読んでみて下さい。

12.1 モジュールの定義

モジュールの定義は方法は以下のように行うことができます。ソースコード 61 では対話環境の応答は省略しています。

ソースコード 61: モジュールの定義

```
structure M1 = struct
  (* トップレベルと同じ定義が書ける *)
  val pi = 3.14
  fun id x = x
  datatype t = A of int | B of real
  exception Exit
  structure M2 = struct val pipi = pi * pi end
end

(* モジュールの要素には モジュール名.変数名 でアクセスする *)
fun area r = M.pi * r * r
val l = [M1.A 12, M1.B M.pi]

fun mul [] = 1
  | mul (x::xs) = if x = 0 then raise M1.Exit else x * mul xs

val r1 = M1.M2.pipi * M1.M2.pipi
val r2 = let open M1 in M2.pipi end

structure M3 = List (* モジュールのエイリアスも書ける *)
val l = M3.map M.id [1,2,3]
```

モジュール定義をオープンして定義をトップレベルに展開するためには `open` 文を用います。`val r2 = ...` の部分ではモジュール `M` をオープンして `M.` という記述を省略できるようになります。モジュール名が長い場合や、トップレベルに見えているモジュールを上書きする場合に有効です。例えば強化標準ライブラリを作る際に、ソースコード 62 のように、モジュールを定義しておいて、使う部分でモジュール `PoweredStd` をオープンすれば既にある `List` モジュールを上書きして `List` という記述は `PoweredStd.List` を指すようになります。これはプログラムの挙動を一気に変えるときにとても便利です。

ソースコード 62: `open` の使いどころ

```
structure PoweredStd = struct
  structure List = ...
  structure Array = ...
  ...
end
```

12.2 シグニチャ

さてモジュールの型のようなものである、シグニチャの使い方を学びましょう。シグニチャの役割は主に 2 つあります。

- モジュールにシグニチャが要求する変数・関数・型が宣言されていることを強制する
- シグニチャに書かれた関数・変数・型以外にはアクセスできなくする

ソースコード 46 で 2 分探索木を用いた辞書の定義を行いました、それをモジュール内で定義した時のシグニチャを定義してみましょう。ソースコード 63 を見て下さい (`'a dict` を `'a t` に名前を変えています)。

ソースコード 63: シグニチャの定義

```
(* シグニチャの定義は signature ID = sig ... end で行う *)
signature DICT = sig
  type 'a t
  val empty : 'a t
  val insert : string -> 'a -> 'a t -> 'a t
  val find : string -> 'a t -> 'a option
  val printDict : (string -> 'a -> string) -> 'a t -> unit
end
```

シグニチャ宣言の文法は **signature** モジュール名 = **sig** 宣言 **end** です。シグニチャに含まれる変数の宣言には **val** 変数名 : 型、ヴァリエント型には普通のトップレベルでの定義のように **datatype** 型名 = 定義を書き、型隠蔽 (後述) のためには上の定義のように **type** <パラメータ> 型名を宣言します。型隠蔽で **type** を用いる場合には定義を書かないところに注意して下さい。

ソースコード 64: モジュール Dictにシグニチャを付ける

```
structure Dict :> DICT = struct
  datatype 'a t =
    LEAF
  | NODE of {left: 'a t, key:string, value: 'a, right: 'a t}

  val empty = LEAF

  fun insert k v LEAF = NODE {left= LEAF, key= k, value= v, right= LEAF}
  | insert k v (NODE {left, right, key, value}) =
    if String.< (k, key)
    then NODE {left= insert k v left, right=right, key= key, value= value}
    else NODE {left= left, right= insert k v right, key= key, value= value}

  fun find k LEAF = NONE
  | find k (NODE {left, right, key, value}) =
    if k = key then SOME value else
    if String.< (k, key) then find k left else find k right

  fun toString f LEAF = "LEAF"
  | toString f (NODE {left,right,key,value}) =
    "(" ^ toString f left ^ ", " ^ f key value ^ ", " ^ toString f right ^ ")"

  fun printDict f dict = print (toString f dict)
end
```

ソースコード 64 では、上で定義したシグニチャDICTの制約を Dict モジュールに Dict :> DICT という記述によって課しています。これがなければ何の制約もないただのモジュール定義になります。シグニチャDICTの 'a t 型の宣言を見ると **type** を用いており定義が書いてないため、この詳細にアクセスできなくしています。またシグニチャDICTには関数 toString のことは書いてないので、シグニチャDICTの制約が課されたモジュール Dict ではこの関数にアクセスできなくしています。ソースコード 65 を見て下さい。

ちなみにここでシグニチャ名をすべて大文字で DICT、モジュール名を Dict としています。別にこのような名前でなければならない理由はありません。ただ、SML/NJ の習慣ではモジュール名は最初の文字を大文字にして他は小文字、シグニチャ名には全て大文字を用いるようです。ただモジュールとシグニチャで関数などの名前は一致してなければなりません。

ソースコード 65: Dictを使う

```
fun op |> (x, f) = f x
infix 1 |>

val dict = Dict.insert "wakaba" 1 Dict.empty
          |> Dict.insert "saikyo-no-method" 2
          |> Dict.insert "red-dragon" 3
```

```
(* 'a Dict.t 型の実装は型隠蔽により隠されているので中身がプリントされない！ *)
val dict = - : int Dict.t

val str = Dict.toString dict
(* toStringはシグニチャDICTによって隠したのでアクセスできない *)
stdIn:44.11-44.24 Error: unbound variable or constructor: toString in path Dict.toString
```

もしモジュール Dict に printDict が無い時はどうなるのでしょうか？この定義を消して対話環境に入力してみると、stdIn:45.1-67.4 Error: unmatched value specification: printDict と表示され、関数 printDict がモジュールの中にないぞ！と怒られてしまいます。

一般にシグニチャによる型の隠蔽はどんどんやるべきです。その理由は

- コードの安全性のため
'a Dict.t 型の値のヴァリエーションに直接触れると 2 分探索木になっていない木もユーザが作れてしまいます。このような事を禁止できれば安全性は高まります。
- モジュール定義の柔軟性のため
'a Dict.t 型の定義を非公開にしておけば、この内部実装に依存したコードが書けなくなります。あとで内部実装を配列・ハッシュテーブルを用いたより効率の良いものに交換することが容易になります。

ソースコード 65 を見ると、辞書の実装が 2 分探索木であることに依存している記述はどこにもありません。だとすれば内部の実装が例えばリストでも良いはずですが。ソースコード 66 ではリストを用いた辞書の定義を行いました。これもシグニチャ DICT の誓約を満たします。リストの探索ではリストの探索にはリストの長さ n に対して線形時間がかかりますが、2 分探索木による実装では対数時間で済みます。しかし 2 分木による実装のほうがコード量も多いし大変であれば、まずはリストを使って実装して後で差し替えればいいですね。このようなことが容易に出来ることをシグニチャによる実装の隠蔽により保証できます。

ソースコード 66: リストによる辞書の実装

```
structure Dict2 :> DICT = struct
  type 'a t = (string * 'a) list

  val empty = []
  fun insert k v l = (k, v) :: l
  fun find k l = Option.map (fn (k, v) => v) (List.find (fn (k', v) => k = k') l)
  fun toString f [] = ""
    | toString f ((k, v) :: tl) = "(" ^ f k v ^ ")" ^ toString f tl
  fun printDict f l = (print (toString f l); print "\n")
end
```

もしシグニチャ制約は課すけども、型情報を一部公開したかったら `where type` 文を用いることが出来ます。例えば `structure Dict2 :> DICT where type 'a t = (string * 'a) list = ...` と書くところここで書いた定義が外部に公開されます。

12.3 ファンクタ (functor)

ファンクタは定義 (の列) を受け取ってモジュールを返す関数のようなものです。例として 2 分探索木を提供するファンクタを定義してみましょう。2 分探索木ではその性質上ノードのラベル同士で大小判定出来ないといけません。SML には `compare: 'a -> 'a -> bool` の型を持つ関数はイコールしか存在しないため、多相的な 'a tree のような型の値は作れませんでした。

そこで、ここでいう 'a に相当する型とそれの大小判定関数を渡してやり、渡された型が格納できる 2 分探索木を扱うモジュールを返すファンクタを定義してみましょう。ファンタは **functor** ファンクタ名 (宣言の列) = **struct** ... **end** という構文により定義されます。宣言の列としてよく用いられるのは以下のとおりです。

- **val** 変数名 : 型

- **type** 型名
- **datatype** 型名 = ...
- **structure** モジュール名 : シグニチャ名
- **exception** 例外名 **of** 型名

ソースコード 67: ファンクタの例

```
(* 2分探索木を返すファンタBinTreeの定義 *)
functor BinTree (type elem
                val eq : (elem * elem) -> bool
                val compare : (elem * elem) -> bool) = struct
  type elem = elem
  datatype t = LEAF | NODE of {elem: elem, left: t, right: t}
  fun insert v LEAF = NODE {elem=v, left = LEAF, right = LEAF}
    | insert v (t as NODE {elem, left, right}) =
      if eq (v, elem) then t else
      if compare (v, elem) then NODE {elem=elem, left=left, right=insert v right} else
      NODE {elem=elem, left=insert v left, right=right}
  fun mem v LEAF = false
    | mem v (NODE {elem, left, right}) =
      if eq (v, elem) then true else
      if compare (v, elem) then mem v right else mem v left
end

(* ファンクタを使ってモジュールをつくる *)
structure StringTree =
  BinTree(type t = string val eq = (op =) val compare = String.<=)

val tree = let open StringTree in insert "3" LEAF |> insert "1" |> insert "2" end;
val tree =
  NODE {elem="3",left=LEAF,right=NODE {elem="1",left=NODE #,right=LEAF}}
  : StringTree.tree

(StringTree.mem "3" tree, StringTree.mem "4" tree);;
val it = (true,false) : bool * bool
```

ここではファンクタが返すモジュール自体の tree 型の実装は隠蔽されていませんが、ファンクタがモジュールにもシグニチャの制約を課すことが出来き、型隠蔽を行うことも出来ます。

ソースコード 68: 2分木のシグニチャ

```
signature BINTREE = sig
  type elem
  type t
  val insert : elem -> tree -> tree
  val mem : elem -> tree -> bool
end
```

ソースコード 69: BinTree の再定義

```
functor BinTree
  (type elem
   val eq : (elem * elem) -> bool
   val compare : (elem * elem) -> bool)
  :> BINTREE where type elem=elem = struct
  ...
end
```

ソースコード 69 では、ファンクタ BinTree を再定義しています。定義は一緒なのですが、増えた部分は:> BINTREE **where** **type** elem=elemの部分です。:> BINTREE はファンクタが返すモジュールのシグニチャ制約で、**where** 以降はシグニチャ制約によって隠す型 elem を公開することを宣言しています。where type 文がなければ型 elem の実装を隠してしまい、このファンクタが作るモジュールの関数の使い道が一切なくなってしまうです。

OCaml[4] ではモジュールを動的に作ることが出来ます。そのため例えば CUI ツールを作った時に、全く同じモジュールで同じ関数呼び出しをしているようにコードを書いてもコマンドオプションによってモジュールの中身を切り替えてしまっ行って動作を切り変える、みたいな事を簡単に行うことが出来ます。

詳細は ocaml のマニュアルを読んで欲しいのですが^a、ざっくり概要だけ説明します。以下のコードのポイントは、モジュールを普通の値として使うために pack している部分と、pack されたモジュールを unpack している部分があることです。pack されたモジュール (小文字の `m`) は普通の値のように使えますがモジュールの要素に直接アクセスできません。unpack された普通のモジュール (大文字の `M`) には要素にアクセスできますがモジュール自体は値ではないので操作するために pack する必要があります。その pack にあたるのが `(module M : S)` で、unpack にあたるのが `val m : S` です。これが何故必要なのかは、型システム入門などで存在型について調べてみて下さい。

ソースコード 70: 第一級モジュール モジュールを作る関数と使う関数

```
module type S = sig type t val init : t val to_string : t -> string end;;

(* シグニチャ制約Sを満たすモジュールを作って返す関数 *)
let make_mod x = let module M = struct
  type t = int
  let init = x
  let to_string t = string_of_int t end in (module M : S);;
val make_mod : int -> (module S) = <fun>

(* シグニチャ制約Sを満たすモジュールを受け取ってそれを使う関数 *)
let use_mod m = let module M = (val m : S) in M.to_string M.init;;
val use_mod : (module S) -> string = <fun>
```

^a<http://ocaml.jp/refman/ch07s14.html>

Part III

プログラムの例

References

- [1] The Standard ML Basis Library
<http://sml-family.org/Basis/index.html>
- [2] 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/>
- [3] Nick Rothwell, “Parsing in the SML Kit”
<http://www.lfcs.inf.ed.ac.uk/reports/92/ECS-LFCS-92-236/ECS-LFCS-92-236.ps>
- [4] “ocaml” <http://ocaml.org/>
- [5] “Scrap Your Boilerplate using Dynamic Types in Ordinary SML”
<https://www.cs.hmc.edu/~oneill/syb/syb-ml-oneill.pdf>