プログラミング言語Standard ML入門
— Introduction to Standard ML —

大堀 淳

http://www.pllab.riec.tohoku.ac.jp/~ohori/

ML とはどんな言語?

幾つかの答え

- LCF システムのメタ言語（Meta Language）.
- 現在では汎用のプログラミング言語.
- 関数型言語の一つ.
- 静的に型付けられた言語.
- 堅牢な理論的基礎を持つ言語.

補足：Metaとは？

- 「後に」の意味．もともとは単なる位置の概念
- アリストテレスのmetaphysica（形而上学）により「超越」との解釈が加わる
  - もともとはmetaphysica（ta meta ta physika）「自然学（physica）の後に置かれた書」の意味.
  - その内容から，metaは自然学を越えた，との意味となる.
- カントの超越論的観念論（純粋理性批判）によって，自然学の「超越」が,「認識の地平を越える」との解釈に洗練される.
- 言語に関する文脈では，メタとは，言語活動を分析反省する立場．メタ言語：(対象）言語活動を記述するための言語.
- 例：Cで書かれたJAVAのコンパイラでは，CはJAVAを翻訳するためのメタ言語.

MLの特徴は?

幾つかの重要な特徴：

- ユーザ定義のデータ型
- パターンマッチング
- 自動的なメモリー割り当て（ごみ集め付きヒープ管理）
- 第1級のデータとしての関数
- 言語と一体となったモジュールシステム
- 多相型言語
- 自動型推論機構を装備

MLはハッカー[1] のためのクールなプログラミング言語.

[1] A person with an enthusiasm for programming or using computers as an end in itself.

MLの歴史

- 60's: LCF システムのメタ言語として誕生
- 1972: Milnerの多相型型推論アルゴリズム.
- early 80'S: CardelliによるFAMを使った実装
- 80's: "Polymorphism" 誌へのStandard MLの提案
- 80's: Standard MLの開発（Edinburgh大）
- 80's: Camlの開発（INRIA）
- late 80's: LeroyによるCaml-lightの開発（INRIA）
- early 90's: AppelとMacQueenによるStandard ML of New Jerseyの開発（Bell研/Princeton）
- 90's: LeroyによるObjective Camlの開発（INRIA）
- 2008: SML#を開発（東北大学電気通信研究所）

現在のMLの主な方言

1. Standard ML
   「公式」版．その仕様が以下の本として出版されている．

   R. Milner, M. Tofte, R. Harper, and D. MacQueen
   The Definition of Standard ML (revised), MIT Press, 1997.

2. Objective Caml
   フランスの方言．フランスINRIA研究所で開発．言語の名前であるとともに，処理系の名前でもある．
   高性能な処理系だが，上記の仕様との互換性はない．

3. SML♯
   東北大学電気通信研究所堀研究室で開発中．alpha版をリリースずみ．Standard MLの拡張版；The Definition of Standard MLと完全な上位互換性．

MLの教科書

Our lecture will roughly be based on

プログラミング言語Standard ML入門，大堀 淳，共立出版.

Other resources

- L. Paulson, "ML for the Working Programmer, 2nd Edition", Cambridge University Press, 1996.
  (多くのトピックが扱われた良書)
- R. Harper, "Programming in Standard ML".
  http://www.cs.cmu.edu/People/rwh/introsml/
  (on-lineで入手できる．)
- M. Tofte, "Four Lectures on Standard ML" (SML'90)
  ftp://ftp.diku.dk/pub/diku/users/tofte/FourLectures/sml/
  (短いがよく書けたチュートリアル)

MLのFAQ，或は，MLに関する種々の偏見・誤解

- Q. MLは理論に基づく言語なので，学ぶのが難しいのでは？
  A. 実際は逆です.

  堅牢な基礎の基に良く設計された機械は，理解しやすく操作も簡単.

  MLプログラミングをマスターするには，いくつかの数少ない原理を身を理解すればよい.

  – 式とその評価,
  – 関数 (再起関数，第1級のデータとしての関数),
  – 型,
  – データ型とパターンマッチング.

  これらMLの構造は，手続き型言語より遥かに単純.

- Q. そうはいっても，関数型プログラムはむずかしいのでは？
  A. いえ，手続き型プログラムより単純かつ自然です.
  例：

$$0! = 1$$
$$n! = n \times !(n-1)$$

```
fun f 0 = 1
  | f n = n * f (n - 1)
```

```
f(int n){
    int r;
    while (n != 0) {
        r = r * n;
        n = n - 1; }
    return(n);}
```

一般にMLのコードはより簡単でエラーが少ない.

- Q. コンパイルエラーが多く，開発の能率が悪いのでは？
  A. いえ，コンパイルエラーのおかげで開発効率が高いのです.
  エラーを含むコードの例：

```
fun f L nil = L
  | f L (h::t) =
        f (h@L) t

Type Error:
circularity
'Z list -> 'Z list
'Z -> 'Z list
```

```
(defun f (l1 l2)
  (if (null l1) l2
      (cons (cdr l1)
            (f (car l1) l2))))

f
```

  MLはこのように総ての型エラーをコンパイル時に報告する.
  これにより，MLのプログラム開発効率は他を圧倒して高い.

- Q. MLで書かれたプログラムは遅いのでは？
- MLは所詮研究者のおもちゃでしょう.

⋮

研究者／ハッカーとして，実際のソフトウエア開発現場で使い物になるML言語を開発し，この２つに応えたい ....

以降の内容

1. MLを使ってみよう
2. 値の束縛と関数定義
3. 再帰的関数と高階の関数
4. 静的型システム
5. リスト

参考資料

さらに興味のある方は，本特別講義の参考資料のページから

⇒ 「MLに関する種々の情報」（参考）

⇒ 「プログラミング言語Standdard ML入門に準拠した解説スライド」（英語版）

を参照．本資料の内容以外に，以下のような内容が含まれます．

6. MLの組み込みデータ型
7. 再帰的データ型の定義
8. 手続き型プログラミングの機能
9. モジュールシステム
10. 基本ライブラリ
11. プログラミングプロジェクト

1. MLを使ってみよう

MLをインストール

1．SML of NJのホームページ

http://www.smlnj.org/software.html

に行き，公式リリース版である110.0.7-4版をダウンロード.

2．SML＃のホームページ

http://www.pllab.riec.tohoku.ac.jp/smlsharp/ja/

に行き,「ダウンロード」のページからSML♯アルファリ版をダウンロード

MLの起動と使い方

対話型モードの実行を開始

```
% smlsharp                          (SML#システムの起動)
restoring static environment...     (起動情報)
#                                   (入力プロンプト)
```

この後，以下のような対話をしながらプログラムする．

1. ユーザによるプログラムの入力,
2. システムによるプログラムのコンパイルと実行,
3. システムによる結果の表示.

セッションの終了

- 対話型セッションの終了にはトップレベルで`^D`を入力する．
- プログラムを中断するには`^C`を入力する

式とその評価

関数型言語の基本原理

プログラムは一つの式．式の評価がプログラムの実行．

対話型セッションで以下のように式を入力:

`#` $expr$ `;`	(an expression)
$val\ it = value : type$	(the result)

ここで

- ";" はコンパイル単位の終了記号．
- $value$ はプログラムの実行結果を表す式の値．
- $type$ はMLコンパイラが推論した式の型．
- `val it` は，システムが結果に "`it`" という名前を付け記録していることを表す．

原子式

MLでは，“hello world” プログラムは単に以下のように書けばよい:

```
% smlsharp
# "Hi, there!";
val it = "Hi, there!" : string
```

MLでは簡単なプログラムは実際に簡単！
このプログラムをCのプログラムと比較してみよ.

注意:

- "Hi, there!" は一つの式．従って完全なプログラム.
- これは，原子式（定数リテラル式）の例.
- 原子式の評価結果はそれ自身（の計算機での表現）.
- MLは式の型を推論する．この場合，stringが推論されている.

幾つかの原子式の例：

```
# 21;
val it = 21 : int
# 3.14;
val it = 3.14 : real
# 1E2;
val it = 100.0 : real
# true;
```

```
# true;
val it = true : bool
# false;
val it = false : bool
# #"A";
val it = #"A" : char
```

ここで

- *int*：整数型
- *real*：浮動小数点数の型
- *bool*：真理値型
- *char*：文字データ型

組み込み演算式

簡単な算術式の例

```
# ~2;
val it = ~2 : int
# 3 + ~2;
val it = 1 : int
# 22 - (23 mod 3);
val it = 20 : int
# 3.14 * 6.0 * 6.0 * (60.0/360.0);
val it = 18.84 : real
```

- ~nは負の数の表現．-はマイナス１倍する演算.
- modは剰余を求める演算子

式はいくら長くてもよい.

```
# 2 * 4 div (5 - 3)
> * 3 + (10 - 7) + 6;
val it = 21 : int
```

最初の `>` は継続行のプロンプト.

条件文と真理値を持つ式

MLではプログラムは一つの式（基本原理（１））．よって，

条件分岐するプログラムもやはり１つの式

```
# if true then 1 else 2;
val it = 1 : int
# if false then 1 else 2;
val it = 2 : int
```

ここで`true`と`false`はそれぞれ真と偽を表す`bool`型の定数．

注:

- `if` E_1 `then` E_2 `else` E_3は式であり，従って値をもつ．
- E_1は任意の真理型の式でよい．

型付き言語の基本原理:

式は，型が正しい限りどのように組み合わせてもよい.

したがって，以下のようなことが可能

```
# (7 mod 2) = 2;
val it = false : bool
# (7 mod 2) = (if false then 1 else 2);
val it = false : bool
# if (7 mod 2) = 0 then 7 else 7 - 1;
val it = 6 : int
# 6 * 10;
val it = 60 : int
# (if (7 mod 2) = 0 then 7 else 7 - 1) * 10;
val it = 60 : int
```

“it” 式

システムは it という式に最後に評価された式の値を保持.

```
# 31;
val it = 31 : int
# it;
val it = 31 : int
# it + 1;
val it = 32 : int
# it;
val it = 32 : int
```

文字と文字列

```
# if #"A" > #"a" then ord #"A" else ord #"a";
val it = 97 : int
# chr 97;
val it = #"a" : char
# str it;
val it = "a" : string
# "SML" > "Lisp" ;
val it = true : bool
# "Standard " ^ "ML";
val it = "Standard ML" :  string
```

- `ord` e は文字eのASCIIコードを返す.
- `chr` e は整数eに対応する文字を返す.

ファイルからのプログラムの読み込み

ファイルに格納されたプログラムを実行するためには，以下の特殊構文を用いる．

`use "`*file*`" ;`

システムはこの文を以下のように処理する．

1. ファイルを開く．
2. “;” を区切りとして式を読み込み，コンパイル，実行する．
3. ファイルを閉じる．
4. トップレベルに戻る．

構文エラーと型エラー

他の言語同様，構文に誤りがあればエラーを報告する．

```
# (2 +2] + 4);
```

stdIn:1.7 Error: syntax error found at RBRACKET

さらに，型の不整合があれば型エラーを報告する．

```
# 33 + "cat";
```

stdIn:21.1-21.11 Error: operator and operand don't agree [literal]
*operator domain: int * int*
*operand: int * string*
in expression:
33 + "cat"

型のキャストは存在しない

```
# 10 * 3.14;
stdIn:2.1-2.10 Error: operator and operand don't agree [literal]
  operator domain: int * int
  operand: int * real
  in expression:
    10 * 3.14
```

必要なら型の変換を行う.

```
# real 10;
val it = 10.0 : real
# it * 3.14;
val it = 31.4 : real
```

練習問題（1）

1. SML♯をインストールし上記のような対話が可能であることを確認せよ.
2. 以下の結果を予想せよい.

```
1   44;
2   it mod 3;
3   44 - it;
4   (it mod 3) = 0;
5   if it then "Boring" else "Strange";
```

   予想結果を，SML♯で確認せよ.

3. 変数Xに英字の大文字が定義されているとする．ASCIIコードはAからZまでこの順に並んでいる．小文字に付いても同様であ

る．この事実を使い，xに束縛された英字の小文字が得られる式を書け.

4. ASCIIコードでは，英字大文字と英字小文字は連続していない．これら集合の間にある文字数を求める式を書け．その結果を使って，英字の大文字と小文字の間にある文字を連結して得られる文字列を求める式を書け.

2. 値の束縛と関数定義

値の束縛

プログラム開発の第一歩は式に名前をつけ，その後の式の中で使用すること．

`val` *name* `=` *exp* `;`

これによって*name*が*expr*に束縛される． environment.

```
# val OneMile = 1.6093;
val OneMile = 1.6093 : real
# OneMile;
val it = 1.6093 : real
# 100.0 / OneMile;
val it = 62.1388181197 : real
```

変数の型を宣言する必要はない.

```
# val OneMile = 1.609;
val OneMile = 1.609 : real
# val OneMile = 1609;
val OneMile = 1609 : int
# OneMile * 55;
val it = 88495
```

定義されていない変数の参照は型エラーとなる.

```
# onemile * 55;
stdIn:22.1-22.8 Error: unbound variable or constructor: onemile
```

Identifiers

変数として使用可能な名前は，以下2種類

1. アルファベット名
   大文字，小文字, または#"'" ではじまり，文字, 数字, #"'", #"_" だけからなる名前.
   ただし#"'"から始まる名前は型変数としてのみ使用可能.
2. 記号名
   以下の記号からなる名前.

```
!  %  &  $  #  +  -  /  :  <  =  >  ?  @
\  ~  `  ^  |  *
```

予約語

```
abstype and andalso as case datatype do else end
eqtype exception fn fun functor handle if in
include infix infixr let local nonfix of op open
orelse raise rec sharing sig signature struct
structure then type val where while with withtype
(  )  [  [  {  }  ,  :  :>  ;  ...  _  |  =  =>  ->
#
```

関数定義

関数は以下のfun構文で定義する:

```
fun f p = body ;
```

ここで

- fは関数の名前,
- pは仮引数,
- $body$は関数本体.

これによって，引数pを受け取り，$body$の値を計算する関数が定義される.

簡単な例：

```
# fun double x = x * 2;
val double = fn : int -> int
```

ここで

- `val double = fn`は変数`double`が関数の値に束縛されたことを示す.
- `int -> int`はこの関数が，整数を引数として取り整数を返す関数であることを表す型.

  関数は以下のように使用する.

  ```
  # double 1;
  val it = 2 : int
  ```

関数適用

型に関する規則:

> 型τ_1 -> τ_2を持つ関数fは，型τ_1をもつ式Eに適用することができ，結果は型τ_2である．

この条件を簡潔に以下のように書ける．

$$\frac{f : \tau_1 \text{ -> } \tau_2 \quad E : \tau_1}{f\ E : \tau_2}$$

型付き言語の基本原理を思いだそう．

> 式は，型が正しい限り自由に組み合わせることができる．

そこで，Eはどのような式でもよく，また，$f\ E$はτ_1型が許されるところにならどこにでも使用できる．

```
# double (if true then 2 else 3);
```
$val\ it = 4 : int$
```
# double 3 + 1;
```
$val\ it = 7 : int$
```
# double (double 2) + 3;
```
$val\ it = 11 : int$

重要な表記上の約束:

関数適用 $E_1\ E_2$ は左結合し，最も結合力が強い.

たとえば`double 3 + 1`は`(double 3) + 1`の意味である.

簡単なパターンを用いた関数定義

```
# fun f (x,y) = x * 2 + y;
val f = fn : int * int -> int
# f (2,3);
val it = 7 : int
```

ここで,

- `(`E_1`,...,`E_n`)` は組を表す式,
- τ_1 `*` $\cdots$ `*` τ_n は組型.
- 関数構成子 `->` は他の型構成子より結合力が弱い．従って，`int * int -> int` は `(int * int) -> int` と解釈される.

関数的用の型推論規則

$$\frac{E_1 : \tau_1 \quad \cdots \quad E_n : \tau_n}{(E_1, \ldots, E_n) \ : \ \tau_1 \ * \ \cdots \ * \ \tau_n}$$

そこで，以下のように任意の組を構成できる．

```
# ("Oleo",("Kenny","Drew"),1975);
val it = ("Oleo",("Kenny","Drew"),1975)
         : string * (string * string) * int
```

3. 再帰的関数と高階の関数

再帰的関数

関数定義構文

`fun` f p `=` $body$

の$body$ではこの構文で定義している関数f自身を使用することができる.

簡単な例: 階乗を計算する関数

$$0! = 1$$
$$n! = n \times (n-1)!$$

```
fun f n =
    if n = 0 then 1
    else n * f (n - 1)
```

多くの問題は再帰的に考えることによって解くことができる.

再帰的関数の設計:

$$0! = 1$$
$$n! = n \times (n-1)!$$

1. 自明な基底のケースを書く.
   ```
   if n = 0 then 1
   ```
2. 複雑なケースを部分問題に分解し，部分問題を今定義している関数自身を使って解く.
   ```
   f (n - 1)
   ```
3. 結果を合成し，最終結果を作成.
   ```
   n * f (n - 1)
   ```

再帰的関数の分析:

```
fun f n =
    if n = 0 then 1
    else n * f (n - 1)
```

1. fが目的とする結果を返すと仮定する.
   f n is $n!$
2. この仮定の下で$body$が正しいことを確認する.
   f 0 is 0!.
   仮定より，f (n - 1)は$(n-1)!$を計算する
   したがってf nは$n!$. そこで，この定義の本体は正しい.
3. 以上が成功したら，fは目的とする結果を返す関数であると証明される

例：Fibonacci数列:

$$
\begin{aligned}
F_0 &= 1 \\
F_1 &= 1 \\
F_n &= F_{n-2} + F_{n-1} \qquad (n \geq 2)
\end{aligned}
$$

Fibonacci数を計算する関数.

```
fun fib n = if n = 0 then 1
            else if n = 1 then 1
            else fib (n - 1) + fib (n - 2)
```

末尾再起関数

以下の関数定義を再考してみる.

```
fun f n =
    if n = 0 then 1
    else n * f (n - 1)
```

この関数は$O(n)$のスタックスペースを使用. しかし，Cで書けば2，3の変数を使うループですむ.

MLはやはり非効率的？

この答えは自明なもの.

効率的なMLプログラムは効率的で，非効率なMLプログラムは非効率.

再帰的関数定義`fun` f p `=` $body$に於て，fが$body$の中で末尾（tail calls positions）にしか現れないとき，この関数を末尾再起（tail recursive）関数と呼ぶ.

Tの中で[ ]は末尾：

$$T := [\,] \mid \texttt{if}\ E\ \texttt{then}\ T\ \texttt{else}\ T \mid (E;\ \cdots;\ T)$$
$$\mid \texttt{let}\ decs\ \texttt{in}\ T\ \texttt{end}$$
$$\mid \texttt{case}\ e\ \texttt{of}\ p_1\ \texttt{=>}\ T\ \texttt{|}\ \cdots\ \texttt{|}\ p_n\ \texttt{=>}\ T$$

末尾呼び出しの例:

- `f` e
- `if` e_1 `then f` e_2 `else f` e_3
- `let` $decls$ `in f` e `end`

末尾再起関数はスタックを使用せず，従ってより効率がよい.

末尾再起に変更したfact関数:

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

fun factorial n = fact (n,1);
val factorial = fn : int -> int
```

factはfactorialの補助関数.

Let式

letで局所的な定義を導入できる.

```
let
    sequence of val or fun definitions
in
    exp
end
```

簡単な例:

```
let
    val x = 1
in
    x + 3
end;
val it = 4 : int
```

MLでは関数は通常このようにlet式を使って定義される.

```
- fun factorial n =
      let
         fun fact n a = if n = 0 then a
                        else fact (n - 1) (n * a)
      in
         fact n 1
      end;
val factorial = fn : int -> int
```

`let` *decl* `in` *exp* `end`に関する注意:

- もれも式.
- この式の型は*exp*の型.

```
# let
      val pi = 3.141592
      fun f r = 2.0 * pi * r
  in
      f 10.0
  end * 10.0;
val it = 628.3184 : real
```

Local構文

以下のlocal構文も局所的な定義に使用できる.

```
local
    declList1
in
    declList2
end
```

- $declList1$は$declList2$の中でのみ有効
- $declList2$のみがこの構文の後有効.

```
# local
    fun fact n a = if n = 0 then a
                   else fact (n - 1) (n*a)
  in
    fun factorial n = fact n 1
  end;
```
val factorial = fn : int -> int
```
# fact;
```
stdIn:10.1-10.4 Error: unbound variable or constructor fact

相互再帰関数

互いに他の関数を利用し合う2つ以上の関数も定義できる．これらを相互再帰的関数と呼ぶ.

相互再帰的な関数の簡単な例として，ある年の預金の利率がその年の初めの預金残高によって決まる場合の，残高と利率を計算する問題を考えてみよう.

- A_x^n n年目の年末の預金残高
- I_x^n x万円を預けたときの，n年目の利率

とすると以下の関係がある.

$$I_x^n = F(A_x^{n-1}) \quad (n \geq 1)$$
$$A_x^n = \begin{cases} x & (n = 0) \\ A_x^{n-1} \times (1.0 + I_x^n) & (n \geq 1) \end{cases}$$

これらは，キーワード and を用いて以下のように相互再帰的関数として定義できる．

```
# fun I(x,n) = F(A(x,n - 1))
  and A(x,n) = if n = 0 then x
               else A(x,n-1)*(1.0+I(x,n));
val I = fn : real * int -> real
val A = fn : real * int -> real
```

高階の関数

高階言語の基本原理

関数は式で表現される値（第1級のデータ.）

(1) 関数を返す関数

2引数関数は以下のように書ける.

```
# fun Power(m,n) = if m = 0 then 1
                   else n * Power(m - 1,n);
val Power = fn : int * int -> int
# Power(3,2);
val it = 8 : int
```

しかしMLでは，以下のような別の書き方も可能.

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

powerは，mを受け取り「nを受け取りm^nを返す関数」を返す関数.

たとえば，以下のように使用できる.

```
# val cube = power 3;
val cube = fn : int -> int
# cube 2;
val it = 8 : int
```

注意：

- 関数適用は左結合する
  power (m - 1) nは(power (m - 1)) nの意味.
- ->は右結合する
  int -> int -> intはint -> (int -> int)の意味

(2) 関数を受け取る関数

この機能を理解するために，以下の関数を考えてみよう:

```
# fun sumOfCube n = if n = 1 then cube 1
                    else cube n + sumOfCube (n - 1);
val sumOfCube = fn : int -> int
# sumOfCube 3;
val it = 36 : int
```

sumOfCubeは以下のような計算パターンの特殊な場合である.

$$\sum_{k=1}^{n} f(k) = f(1) + f(2) + \cdots + f(n)$$

MLでは，このような計算のパターンを直接関数として定義できる.

```
# fun summation f n = if n = 1 then f 1
                      else f n + summation f (n - 1);
val summation = fn : (int -> int) -> int -> int
```

注:

- (int -> int) -> int -> int は(int -> int) -> (int -> int).
- この型は，summationが
  - 型int -> intの値を受け取り
  - int -> intの値を返す

  関数であることを意味している.

すまり，summationは関数を受け取り関数を返す関数.

「高階言語の基本原理」を「型付き言語の基本原理」と組み合わせれば，以下のように使用可能であることが分かる．

```
# val newSumOfCube = summation cube;
val newSumOfCube = fn : int -> int
# newSumOfCube 3;
val it = 36 : int
```

さらに，

```
# val sumOfSquare = summation (power 2);
val sumOfSquare = fn : int -> int
# sumOfSquare 3;
val it = 14
# summation (power 4) 3;
val it = 98 :  int
```

関数式

関数は値であるから，当然式で表現できる．

$$\texttt{fn}\ p\ \texttt{=>}\ body$$

これは，pを引数として受け取り$body$の値を計算する名前のない関数．

```
# fn x => x + 1;
val it = fn : int -> int
# (fn x => x + 1) 3 * 10;
val it = 40 : int
```

この式を使えば，実行時に関数を作り出すようなプログラムを定義できる．

```
fun f n m = (fib n) mod m = 0;
```

この関数は各mに対してfib mを計算するため，効率がわるい．

```
val g = f 35;
(g 1,g 2,g 3,g 4,g 5);
```

はfib 35を5回計算すうr.

関数式をつかいこのfib 35の計算をくくり出すことが可能.

```
fun f n = let val a = (fib n)
          in fn m => a mod m = 0
          end
```

この関数は，以下の計算を行う.

1. nを受け取る
2. fib nを計算し結果をmに束縛,
3. このmを使い，関数fn m => a mod m = 0を生成し返す.

この定義を使えば，例えば

```
val g = f 35;
(g 1,g 2,g 3,g 4,g 5);
```

などでもfib 35の計算は1回しか実行されない.

静的スコープ規則

名前の使い方の基本法則：

1. 名前の定義には有効範囲があり，
2. 新しい名前の定義は，その有効範囲で，以前の定義を隠す．

この法則は，人間が使用するおよそすべての言語にあてはまる．

簡単な例：

```
# let val x = 3 in (fn x => x + 1) x end;
```
val it = 4 : int

名前が定義される構文：

- `val` 定義
- `fun` 定義
- `fn` x `=>` e 式

val 定義

val x_1 = exp_1
and x_2 = exp_2
⋮
and x_n = exp_n ;

$x_1, \ldots, x_n$のスコープはこの定義に続く部分

```
val x = 1
val y = x + 1
val x = y + 1
and y = x + 1
```

この構文によりxは3にyは2に束縛される.

関数定義

$$\begin{array}{l}\texttt{fun}\ f_1\ p_1^1\ \cdots\ p_{k_1}^1\ \texttt{=}\ exp_1\\ \texttt{and}\ f_2\ p_1^2\ \cdots\ p_{k_2}^2\ \texttt{=}\ exp_2\\ \quad\vdots\\ \texttt{and}\ f_n\ p_1^n\ \cdots\ p_{k_n}^n\ \texttt{=}\ exp_n\ \texttt{;}\end{array}$$

$f_1, \ldots, f_n$のスコープ：

- $exp_1, \ldots, exp_n$
- この定義に続く部分

注:

新しい名前は古い名前を隠すのみ．古い定義を変更するわけではない（通常の言語でもあたりまえ.）.

```
# val x = 10;
```
$val\ x = 10 : int$
```
# val y = x * 2;
```
$val\ y = 20 : int$
```
# val x = 20;
```
$val\ x = 20 : int$
```
# y;
```
$val\ it = 20 : int$

関数定義でも同様.

```
# val x = 10;
val x = 10 : int
# val y = 20;
val y = 20 : int
# fun f x = x + y;
val f = fn : int -> int
# f 3;
val it = 23 : int
# val y = 99;
val y = 99 : int
# f 3;
val it = 23 : int
```

2項演算子

MLでは, 演算子は二引数関数にすぎない.

e_1 `op` e_2

は

$op(e_1, e_2)$

の**糖衣構文**.

この糖衣構文を使いたい場合，以下の宣言を行う.

- `infix` n $id_1 \cdots id_n$: 結合力nの左結合演算子の導入.
- `infixr` n $id_1 \cdots id_n$: 結合力nの右結合演算子の導入.

システムではあらかじめ以下の宣言をしている.

```
infix 7 * /
infix 6 + -
```

ユーザも自由に定義可能.

```
# infix 8 Power;
infix 8 Power
# 2 Power 3 + 10;
val it = 19 : int
```

この宣言を一時的に無効にするには`op` *id*構文を用いる.

```
# Power;
stdIn:4.1 Error: nonfix identifier required
# op Power;
val it = fn : int * int -> int
# op Power (2,3);
val it = 9 : int
```

練習問題 (2)

1. 以下のそれぞれに対して，再帰的な漸化式を書き，それをもとに，関数を定義せよ.

   (1) $S_n = 1 + 2 + \cdots + n$

   (2) $S_n = 1 + (1 + 2) + (1 + 2 + 3) + \cdots + (1 + 2 + \cdots + n)$

2. 上記の各関数の末尾再帰版を定義せよ.

3. 2×2の行列$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$を$(a, b, c, d)$と表現することにする．行列$A$と自然数$n$を受け取り$A^n$を計算する関数`matrixPower(n,A)`を定義せよ.

4. Fibonacci数の定義から以下の等式が得られる.

$$F_n = F_n$$
$$F_{n+1} = F_{n-1} + F_n$$

さらに,

$$G_n = \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}$$

と定義すると

$$G_n = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} G_{n-1}$$

したがって,

$$G_n = \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix} = \underbrace{\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}\cdots\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}}_{n \text{ times}}\begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

となる．そこで，もし $A = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ なら,

$$G_n = A^n \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

である．この性質を使って，F_nを計算する関数を`matrixPower`を使って定義せよ．

5. `summation`を末尾再帰関数として定義せよ．
6. 以下の各関数を定義せよ．using `summation`.

   (1) $f(x) = 1 + 2 + \cdots + n$
   (2) $f(n) = 1 + 2^2 + 3^2 + \cdots + n^2$
   (3) $f(n) = 1 + (1 + 2) + (1 + 2 + 3) + \cdots + (1 + 2 + \cdots + n)$

7. 型`int -> real`をもつ関数fに対して$\Sigma_{k=1}^{n} f(k)$を計算する後悔関数

   ```
   summation' : (int -> real) -> int -> real
   ```

   を定義せよ．
8. $f(x)$を実数上の関数とする．

$$\int_a^b f(x)dx$$

は，ある程度大きなnに対して，以下のように近似できる．

$$\sum_{k=1}^{n}\left(f\left(a+\frac{k(b-a)}{n}\right)\times\frac{b-a}{n}\right)$$

f，n，a，bを受け取り，上記の近似値を計算する関数`integral`を定義せよ．

必用なら，`int`型の値を`real`型に変換する関数`real : int -> real`を使用せよ．

9. `summation`は以下のより一般的な計算パターンの特殊な場合である．

$$\Lambda_{k=1}^{n}(h,f,z)=h(f(n),\cdots,h(f(1),z)\cdots)$$

例えば．$\Sigma_{k=1}^{n}f(k)$は$\Lambda_{k=1}^{n}(+,f,0)$と定義できる．

$\Lambda_{k=1}^{n}(h,z,f)$を計算する高階関数

```
accumulate h z f n
```

を定義せよ.

10. `accumulate`を使って`summation`を定義せよ.

11. `accumulate`を使って以下の各関数を定義せよ.

(1) $f_1(n) = 1 + 2 + \cdots + n$

(2) $f_2(n) = 1 \times 2 \times \cdots \times n$

(3) $f_3(n, x) = 1 \times x^1 + 2 \times x^2 + \cdots + n \times x^n$

12. `int -> bool`なる型の関数fは，fが`true`を返す要素の集合と見なせる．例えば，

```
fn x => x = 1 orelse x = 2 orelse x = 3
```

は$\{1, 2, 3\}$の表現である．この表現に対する以下の集合操作関数を定義せよ．ただし，SMLではexp_1とexp_2の論理和はexp_1 `orelse` exp_2と書木，論理積はexp_1 `andalso` exp_2と書く.

(1) 空集合`emptyset`.

(2) 与えられた要素nのみからなる集合$\{n\}$を返す関数 `singleton`.
(3) 要素を集合に加える関数`insert`.
(4) 要素が集合に含まれているか否かをテストする関数`member`.
(5) 以下の集合論演算. `union`, `intersection`, `difference`.

4. MLの型システム

型推論と型チェック

MLは

1. 任意の式の型を自動推論する.
2. さらに，推論される型は最も一般的な多相型.

自動型推論

MLプログラムは型宣言を必要としない.

```
fun products f n = if n = 0 then 1
                   else f n * products f (n - 1)
```

これは，例えば，Lispなどの型なし言語と同等：

```
(defun products (f n)
   (if (<= n 0) 1
       (* (funcall f n) (products f (- n 1)))))
```

しかし，MLはその型を自動推論する.

```
val products = fn : (int -> int) -> int -> int
```

これにより，完全な静的型チェックが可能.

```
fun factorial n = products n (fn x => x);
```
stdIn:1.19-1.40 Error: operator and operand don't agree
operator domain: int
operand: 'Z -> 'Z
in expression: products n ((fn x => x))

LISPなどと比べると，その利点は明らか.

```
# (defun factorial (n)
          (products n #'(lambda (x) x)))
```
factorial
⋮
```
(··· (factorial 4) ···)
```
Worng type argument: integer-or-marker-p (lambda (x) x)

多相性

一般にプログラムは多くの型を持つ.

```
fun id x = x
```

idは,

```
int -> int
```

や

```
string -> string.
```

その他τ-> τの形の無限に多くの型をもつ.

MLは，これらすべてを正確に表現する型を推論する.

```
val id = fn : 'a -> 'a
```

ここで，'aは任意の型を代表する**型変数**.

型変数は自動的に**具体化（*instantiate*）**される.

```
# id 21;
val it = 21 : int
# id "Standard ML";
val it = "Standard ML" : string
# id products;
val it = fn : (int -> int) -> int -> int
# fn x => id id x
val it = fn :  'a -> 'a
```

以下のようなものも可能

```
# fn x => id id x;
val it = fn : 'a -> 'a
# fun twice f x = f (f x);
val twice = fn : ('a -> 'a) -> 'a -> 'a
```

```
# twice cube 2;
val it = 512 : int
# twice (fn x => x ^ x) "ML";
val it = "MLMLMLML" : string
# fn x => twice twice x;
val it = fn : ('a -> 'a) -> 'a -> 'a
# it (fn x => x + 1) 1;
val it = 5 : int
```

レコード

レコード型とレコード式

レコード型の構文

$\{l_1 : \tau_1, \cdots, l_n : \tau_n\}$

- $l_1, \cdots, l_n$はレコードラベル
- l_iは名前か数字のどちらか.
- $\tau_1, \cdots, \tau_n$はどのような型でもよい.

例

```
type malt = {Brand:string, Distiller:string,
             Region:string, Age:int}
```

レコード式の構文

$\{l_1 = exp_1, \cdots, l_n = exp_n\}$

exp_iはどのような式でもよい.

型の規則

$$\frac{exp_1 : \tau_1 \quad \cdots \; exp_n : \tau_n}{\{l_1 = exp_1, \cdots, l_n = exp_n\} : \{l_1 : \tau_1, \cdots, l_n : \tau_n\}}$$

```
# val myMalt= {Brand = "Glen Moray",
               Distiller = "Glenlivet",
               Region = "the Highlands", Age = 28};
```

val myMalt = {Age = 28, Brand = "Glen Moray",
Distiller = "Glenlivet",
Region = "the Highlands" }
: {Age : int, Brand : string, Distiller : string, Region : string}

型付き言語の基本原理：

expressions can be freely combined as far as they are type correct.

により他の構文と自由に組み合わせることが可能.

```
# fun createGlen'sMalt (name,age) =
      {Brand = name, Distiller = "Glenlivet",
       Region = "the Highlands", Age = age};
val createGlen'sMalt =
    fn : 'a * 'b -> {Age:'b, Brand:'a, Distiller:string, Region:string}
```

レコードの操作

#*l* *e*により，レコード式*e*のラベル*l*の値を取り出す.

```
# #Distiller myMalt;
val it = "Glenlivet" : string
# fun oldMalt (x:{Brand:'a, Distiller:'b,
                  Region:'c, Age:int}) =
      #Age x > 18;
val oldMalt = fn : {Age:int, Brand:'a, Distiller:'b, Region:'c} -> bool
# oldMalt myMalt;
val it = true : bool
```

レコードパターン

$\{l_1 = pat_1, \cdots, l_n = pat_n\}$
$\{l_1 = pat_1, \cdots, l_n = pat_n, \ldots\}$

を関数とともに使用すれば，フィールドを取り出せる．

```
# fun oldMalt {Brand, Distiller, Region, Age} = Age > 18
val oldMalt = fn : {Age:int, Brand:'a, Distiller:'b, Region:'c} -> bool
```

```
# val {Brand = brand, Distiller = distiller,
       Age = age, Region = region} = myMalt;
```
val age = 28 : int
val brand = "Glen Moray" : string
val distiller = "Glenlivet" : string
val region = "the Highlands" : string
```
# val {Region = r, ...} = myMalt;
```
val r = "the Highlands" : string
```
# val {Region, ...} = myMalt;
```
val Region = "the Highlands" : string
```
# val distiller = (fn {Distiller,...} => Distiller) myMalt;
```
val distiller = "Glen Moray" : string
```
# fun getRegion ({Region, ...}:malt) = Region;
```
val getRegion = fn : {Age:'a, Brand:'b, Distiller:'c, Region:'d } -> 'd
```
# getRegion myMalt;
```
val it = "the Highlands" : string

Standard MLにおける型に関する制限

レコードを多相関数と一緒に使おうとすると旨くいなかい.

```
fun name {Name = x, Age =a} = x
```

は許されるが,

```
fun name x = #Name x
stdIn:17.1-17.21 Error: unresolved flex record
(can't tell what fields there are besides #name)
```

となる.

これは現在の理論と実装技術の欠陥.(すべての言語に共通)

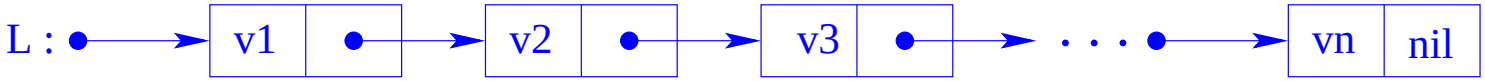
SML♯では,解決済:

```
# fun name x = #Name x;
val name : ['a,'b.'a#Name:'b -> 'b]
```

リストを使ったプログラミング

リストの構造

リストはポインターで実現された以下のような構造.



この構造から，リストの以下の性質が理解できる.

1. リストは，その大きさによらず，一つのポインタで表現.
2. 空のリストは`nil`と呼ばれる特殊なポインターで表現.
3. リストの要素は先頭からしかアクセスできない.
4. 先頭要素を取り除いたものもやはりリスト．先頭に要素を付け加えたものもやはりリスト.

リストの数学的理解

リストは組みが入れ子になった，以下のような構造と理解することができる．

$$(\mathrm{v1}, (\mathrm{v2}, \cdots (\mathrm{vn}, \mathrm{nil}) \cdots))$$

Aを要素の集合とし，Aの要素を$v_1, v_2, \ldots, v_n$の様に書く．Nilを集合$\{nil\}$とする．

２つの集合A, Bのデカルト積$A \times B$を以下の様に定義する．

$$A \times B = \{(a, b) | a \in A, b \in B\}$$

するとn個の要素のリストの集合は以下のように与えられる．

$$\underbrace{A \times (A \times (\cdots (A}_{n\text{個の}A} \times Nil) \cdots))$$

従って，リスト全体の集合は以下の等式で表される．

$$L = Nil \cup (A \times L)$$

この等式の解を以下の手順で求めることができる.

1. 集合列X_iを以下の様に定義する

$$X_0 = Nil$$
$$X_{i+1} = X_i \cup (A \times X_i)$$

もす上記の等式がLに関して解を持てば, 各iについて

$$X_i \subseteq L$$

が成り立つことを示せる.

2. この集合属を用いて,

$$X = \bigcup_{i \geq 0} X_i$$

と定義すると，Xはリストに関する等式を満たすこと，従って求める解であることが分かる.

リスト型：τ `list`

τ list は，要素の型をτとするリスト型．例:

- `int list` : integer lists.
- `int list list` : lists of integer lists.
- `(int -> int) list` : lists of integer functions.

リスト生成関数

リストを生成生成のための定数と関数

```
val nil : 'a list
infixr 5 ::
val op :: : 'a * 'a list -> 'a list
```

これらを用いて，$v_1, \cdots, v_n$を要素とするリストは

v_1 `::` v_2 `::` $\cdots$ `::` v_n `::` `nil`

と書ける．

以下の略記法も用意されている．

`[]` $\Longrightarrow$ `nil`

`[`exp_1`,`exp_2`,`$\cdots$`,`exp_n`]` $\Longrightarrow$ exp_1 `::` exp_2 `::` $\cdots$ `::` exp_n `::` `nil`

リストの例

```
# nil;
val it = [] : 'a list
# 1 :: 2 :: 3 :: nil;
val it = [1,2,3] : int list
# [[1],[1,2],[1,2,3]];
val it = [[1],[1,2],[1,2,3]] : int list list
# [fn x => x];
val it = [fn] : ('a -> 'a) list
```

リスト生成の例

[1,2,3,4,...,n] を生成する関数.

最初のトライ:

```
# fun mkList n = if n = 0 then nil
                 else n ::  f (n - 1);
```

val mkList = fn : int -> int list

```
# mkList 3;
```

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

再挑戦

```
# fun mkList n m = if n = 0 then nil
                   else (m - n) ::  f (n - 1) m
```

val mkList = fn : int -> int-> int liist

```
# mkList 3 4;
```

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

よりまともなコード

```
# fun mkList n =
      let
          fun f n L = if n = 0 then L
                      else f (n - 1) (n::L)
      in
          f n nil
      end;
val mkList = fn : int -> intlist
# mkList 3;
val it = [1,2,3] : int list
```

これのほうが簡潔でかつ効率的. 末尾再帰で考えよ.

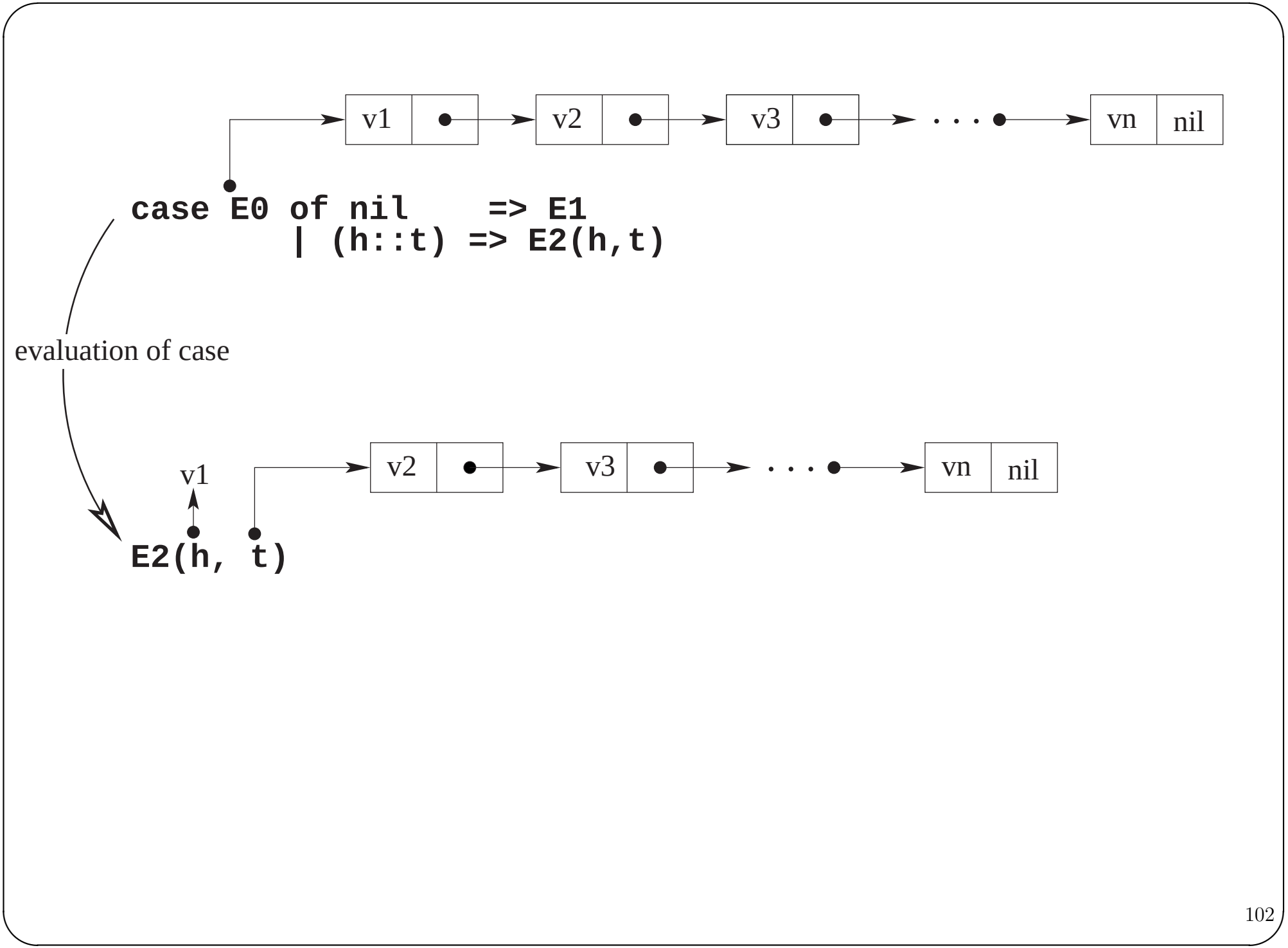
パターンを用いたリストの分解

リスト処理のための基本演算はパターンマッチング:

```
case E0 of nil => E1
         | (h::t) => E2(h,t)
```

これは以下の処理を行う.

1. E_0を評価しリストの値Lを得る.
2. もしLが`nil`なら$E1$を評価.
3. もしLが`h::t`の形なら, `h`をLの先頭要素に，tをLの残りの要素に束縛し，この下で$E2(\mathtt{h},\mathtt{t})$を評価する.


v1
v2
v3
. . .
vn
nil
case E0 of nil => E1
| (h::t) => E2(h,t)
evaluation of case
v1
v2
v3
. . .
vn
nil
E2(h, t)

```
# case nil of nil => 0 | (h::t) => h;
```
$val\ it = 0 : int$
```
# case [1,2] of nil => 0 | (h::t) => h;
```
$val\ it = 1 : int$

パターンマッチングを用いた簡単な例

```
fun length L = case L of nil => 0
                       | (h::t) => 1 + length t;
```

パターンマッチングの一般形

`case` exp `of` pat_1 `=>` exp_1 `|` pat_2 `=>` exp_2 `|` $\cdots$ `|` pat_n `=>` exp_n

ここでpat_iは以下の要素から構成されるパターン

- 定数,
- 変数,
- データ構成子. リストの場合以下の構成子パターンがある
  - `nil`,
  - pat_1::pat_2,
  - [pat_1,$\cdots$,pat_n]

```
fun zip x = case x of (h1::t1,h2::t2) =>
                          (h1,h2) :: zip (t1,t2)
                    | _ => nil
fun unzip x = case x of (h1,h2)::t =>
                            let val (L1,L2) = unzip t
                            in (h1::L1,h2::L2)
                            end
                      | _ => (nil,nil)
```

“_ ”は何にでもマッチするワイルドカードパターン.

パターンは共通部分があってもよく網羅的で無くてもよい.

```
fun last L = case L of [x] => x
                     | (h::t) => last t
```

有用な糖衣構文

```
fun f pat1 = exp1
  | f pat2 = exp2
  ⋮
  | f patn = expn
```

$\Longrightarrow$

```
fun f x = case x of
            pat1 => exp1
          | pat2 => exp2
          ⋮
          | patn => expn
```

```
fn pat1 => exp1
 | pat2 => exp2
 ⋮
 | patn => expn
```

$\Longrightarrow$

```
fn x => case x of
          pat1 => exp1
        | pat2 => exp2
        ⋮
        | patn => expn
```

パターンを用いた関数定義の例

```
fun length nil = 0
  | length (h::t) = 1 + length t

fun fib 0 = 1
  | fib 1 = 1
  | fib n = fib (n - 1) + fib (n - 1)
```

汎用のリスト処理関数

リスト処理関数の再考：

```
fun sumList nil = 0
  | sumList (h::t) = h + sumList t
```

これは，以下の処理パターンの特殊な場合と見なせる．

1. もしリストが`nil`ならある決められた値Zを返す．
2. もしリストがh::tの形なら，自分自身を用いいてtに対する値を再帰的に計算し結果Rを求め，
3. hとRから，ある関数fを用いて最終結果を計算する．

`sumList`のばあい，$Z = 0$および$f(h, R) = h + R$と取ればよい．

以下の高階関数は，上記のパターンを実際に実現する関数.

```
fun foldr f Z nil = Z
  | foldr f Z (h::t) = f(h,foldr f Z t)
```

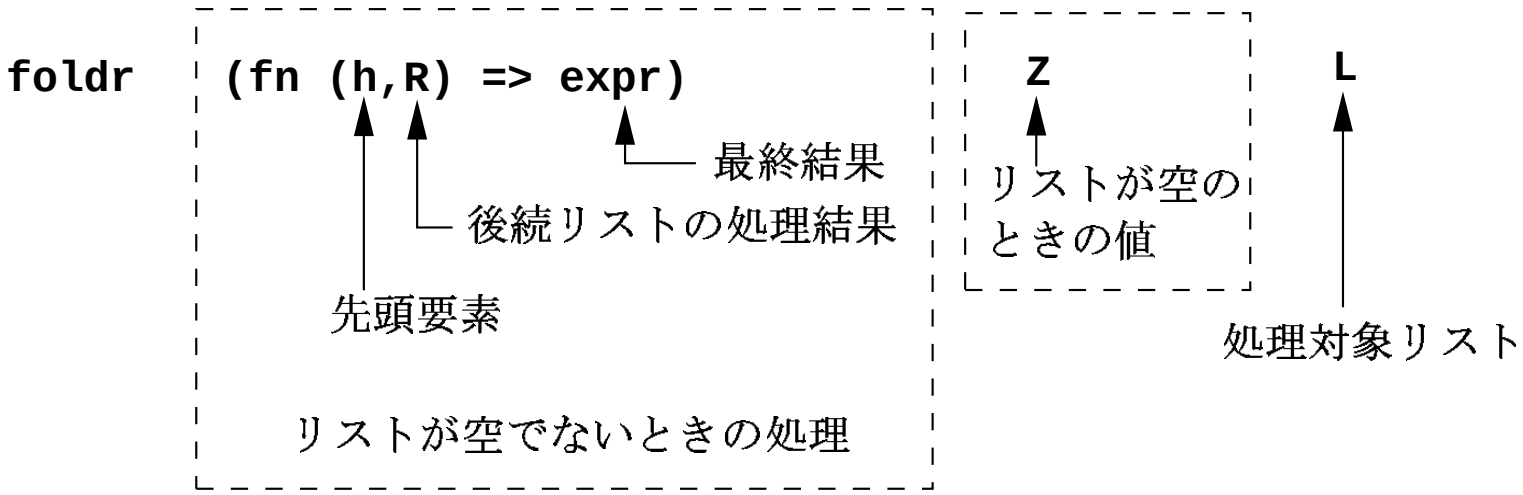
これを用いれば，個々のリスト処理関数を，以下のような適用によって簡単に実現できる.

`foldr (fn (`h`,`R`) => `exp`)` Z

ここで,

- Z は`nil`に対する値.
- `fn (`h`,`R`) => `expは，２番目以降のリストを処理した結果Rtoリストの先頭要素hから最終結果を計算する関数.

```
# val sumList = foldr (fn (h,R) => h + R) 0 ;
```
val sum : int list -> int



プログラミング例

関係Rの推移的閉包R^+は以下のように定義される関係である.

$$R^+ = \{(x,y) \mid \exists n \exists z_1 \cdots \exists z_n \; x = z_1, y = z_n, \\ (z_i, z_{i+i}) \in R (1 \leq i \leq n-1)\}$$

与えられたRからR^+を計算するプログラムを定義してみよう.

1. アルゴリズムを設計ために，まず，R^+の定義を実行可能な（構成的な）なもに書き換える.

   R^+の中に含まれるべき要素の性質を考えると，十分に大きなNを取ると，R^+は以下のように書き直せる.

$$R^+ = R^1 \cup R^2 \cup \cdots \cup R^N$$

2. 次に，R^k を以下のように再帰的に定義する.

$$\begin{aligned} R^1 &= R \\ R^k &= R \times R^{k-1} \qquad (k \geq 2) \end{aligned}$$

ここで，$\times$ は以下のような演算である.

$$R \times S = \{(x, y) | \exists a \ (x, a) \in R, (a, y) \in S\}$$

3. R^+を定義する.
   まず，$\mathcal{R}^N = R^1 \cup \cdots \cup R^N$を$N$に関して関数として定義する.
   以前でてきたaccumulate関数の考え方を使えば，hを和集合演算，$f(k) = R^k$，$z = \emptyset$として，

$$\mathcal{R}^N = h(f(N), \cdots, h(f(1), z) \cdots) = \Lambda_{k=1}^{N}(h, f, z)$$

   と書ける.
   Nは，Rに含まれる最大のチェーンの長さ以上であればよい.
   その一つの見積もりとしてRの要素の数ととることができる.
   以上から，以下のような定義が得られる.

$$R^+ = \Lambda_{k=1}^{|R|}(h, f, z)$$

4. $R \times S$, R^n, R^+をそれぞれ計算する関数timesRel, powerRel, tcのコードを上に与えた定義に従って書き下す.

```
fun timesRel (R,S) =
    foldr (fn ((x,a),r) =>
              foldr (fn ((b,y),rs) =>
                        if a=b then (x,y)::rs
                        else rs)
              r S)
    nil R;
fun powerRel r 1 = r
  | powerRel r n = timesRel (r,powerRel r (n - 1));
fun tc R =
    accumulate (op @) nil (powerRel R) (length R)
```

すると，以下のような動作をする関数が得られる．

```
# tc [(1,2),(2,3),(3,4)];
val it = [(1,4),(1,3),(2,4),(1,2),(2,3),(3,4)]: (int * int) list
```

練習問題 (4)

1. 以下の書く関数を再帰的関数として直接定義せよ.
   (1) 与えられた整数リストの和を求める関数sumList.
   (2) 与えられた値が与えられたリストに入っているか否かをテストする関数member
   (3) 与えられたリストから重複する要素を取り除いて得られるリストを返す関数unique
   (4) filter 型'a -> boolを持つ関数Pと型'a listのリストを受け取り，Pがtrueを返す値のみからなるリストを返す関数filter
   (5) 以下のようにリストのリストを一つのリストにする関数flatten：

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

(6) 文字列のリストLと文字列dを受け取り，Lの中の文字列をすべてdを区切り記号として繋げて得られる一つの文字列を返す関数splice．たとえば以下のような動作をする．

```
# splice (["","home","ohori","papers"],"/");
```
val it = ”/home/ohori/papers” : string

2. 以下の関数をfoldrを使って定義せよ．

(1) 以下の各関数．map，flatten，member，unique，prefixList，permutations

(2) リストと述語P（すなわちboolの型の値を返す関数）を受け取り，リストの総ての要素が述語Pを満たすか否かをテストする関数forall．同様に，リストの中に述語Pを満たす要素が存在するか否かをテストする関数exists．
上記の定義から，任意のPに対してexists P nilはfalseであり，forall P nilはtrueであることに注意．

2.1. 整数のリストの先頭からのそれぞれの要素までの和のリストを返す関数prefixSum．例えば，与えられたリストが$[a_1, a_2, \cdots, a_n]$であれば，
$[a_1, a_1 + a_2, \cdots, a_1 + \cdots a_{n-1}, a_1 + \cdots + a_n]$を返す．

3. foldr は，nil と :: で構成されているリストの構造に従って再帰的に処理する上で威力を発揮するが，再帰的な処理がうまく記述できない問題に対しては，うまく利用できない．たとえば，リストの最後の要素を求める関数や，リストを反転させる関数などはうまく記述できない．これらの処理は，リストの要素を逆順に処理する関数があれば，簡単に定義できる．SMLでは，以下のような動作をするfoldlも定義されている．

$\texttt{foldr}\ f\ Z\ [a_1,\cdots,a_{n-1},a_n] = f(a_n, f(a_{n-1}, f(\cdots, f(a_1, Z)\cdots)))$

(1) foldlを実現する関数定義を与えよ．

(2) foldlを用いてrevを定義せよ．

4. 関数Rの表現が重複する要素を含まない時，その豹変は正規形であるということにする．

   (1) Rが正規形であってもtc Rが正規形でないような例をあげよ．

   (2) 正規形のRに対しては，常に正規形の表現を返すようにtcの定義を改良せよ．

   関係に関する以下の関数を定義せよ．

   (1) 関係Rが与えられた時，(a, b)は関係しているかをテストする関数isRelated.

   (2) 与えられた関係Rと与えられた要素aに対して，集合を$\{x \mid (a, x) \in R\}$を計算する関数targetOf.

   (3) 関係Rと要素aに対して集合$\{x \mid (x, a) \in R\}$を計算する関数sourceOf.

   (4) 関係Rの逆関係R^{-1}を求める関数inverseRel

データ型の定義

datatype文は新しい型を定義する

例: 二分木

型τの値をノードに持つ二分木を型τの二分木と言うということにする．型τの二分木は，再帰的に以下のように定義される．

1. 空の木は型τの二分木である．
2. もしvがτの値，T_1とT_2が型τの二分木なら(v, T_1, T_2)も型τの二分木である．

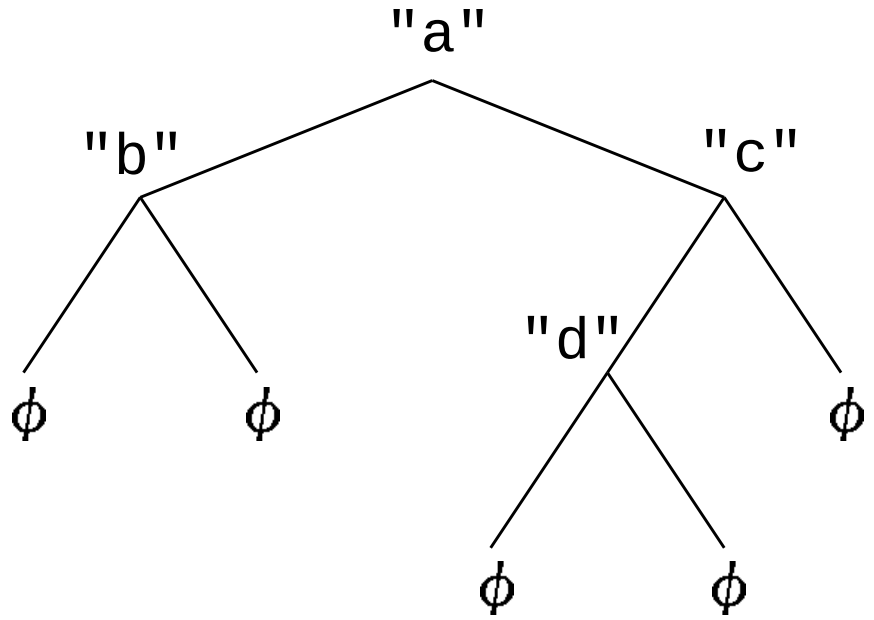
このような再帰的な定義によって定まる型は，直接以下の構文で導入できる．

```
# datatype 'a tree =
            Empty
          | Node of 'a * 'a tree * 'a tree;
datatype 'a tree = Empty | Node of 'a * 'a tree * 'a tree
```

この定義の後Emptyと Nodeは'a tree型データの値を構成する構成子（関数）として使用できる.

```
# Empty;
val it = Empty : 'a tree
# Node;
val it = fn : 'a * 'a tree * 'a tree -> 'a tree
# Node (1,Empty,Empty);
val it = Node (1,Empty,Empty) : int tree
# Node ((fn x => x),Empty,Empty);
val it = Node (fn,Empty,Empty) : ('a -> 'a) tree
```

```
Node("a",Node("b",Empty,Empty),
         Node("c",Node("d",Empty,Empty),Empty))
```



（φ は空の木を表す）

木の pre-order 表現 `a(b()())(c(d()())())`.

二分木は文字列で系統的に表現できる.
その一つは，以下のように定義される pre-order 表現である.

- `Empty` は空文字列で表現する.
- `Node`(a, L, R) は，$a(S_L)(S_R)$ と表現する．ここで，S_L および S_R は R と L の pre-order 表現である.

この表現は，二分木を

1. 木のルール,
2. 左部分木,
3. 右部分木

とたどって得られる文字列である.

pre-ordre表現から木再構成関数は，以下のように定義できる．

```
fun fromPreOrder s =
    let fun decompose s = ...
            (* 文字列を‘(’と‘)’を区切りとして分解する．*)
    in if s = "" then Empty
       else let val (root,left,right) = decompose s
            in Node(root,
                    fromPreOrder left,
                    fromPreOrder right)
    end
```

decomposeは以下のような分割を行う．

```
# decompose "a(b()())(c(d()())())";
val it = ("a","b()()","c(d()())()") : string * string * sting
```

```
fun decompose s =
    let fun searchLP s p = ...
            (* 位置pからsを走査し最初の左括弧の位置を返す. *)
        fun searchRP s p n = ...
            (* 位置pからsを走査しn個外側の右括弧の位置を返す.
        val lp1 = searchLP s 0
        val rp1 = searchRP s (lp1+1) 0
        val lp2 = searchLP s (rp1+1)
        val rp2 = searchRP s (lp2+1) 0
    in (substring (s,0,lp1),
       substring (s,lp1+1,rp1-lp1-1),
       substring (s,lp2+1,rp2-lp2-1))
    end
```

`datatype`構文の一般形

```
datatype typeSpec =
           Con1 ⟨of type1⟩
         | Con2 ⟨of type2⟩
           ⋮
         | Conn ⟨of typen⟩
```

- *typeSpec*は定義する型の名前
- =の右側が新しい型の構造の定義

パターンマッチングを用いたdatatypeの利用

以下の`case`構文において

`case` exp `of` pat_1 `=>` exp_1 `|` pat_2 `=>` exp_2 `|` $\cdots$ `|` pat_n `=>` exp_n

pat_iは以下の要素を含むパターン:

1. 変数,
2. 定数,
3. `datatype`文で定義済のデータ構成子,
4. ワイルドカードパターン：`_`

```
# case Node ("Joe",Empty,Empty) of Empty => "empty"
                                  | Node (x,_,_) => x;
```
val it = ”Joe” : string

木の高さの計算の例

1. 空の木の高さは0
2. Node(a,L,R) の形の木の高さは1 + max(Rの高さ, Lの高さ)

従って，以下の様にコードできる

```
# fun height t =
      case t of Empty => 0
              | Node (_,t1,t2) => 1 + max(height t1, height t
```
val height = fn : ’a tree -> int

パターンマッチングの例

```
fun height Empty = 0
  | height (Node(_,t1,t2)) = 1 + max(height t1, height t2)

fun toPreOrder Empty = ""
  | toPreOrder (Node(s,lt,rt)) =
      s ^ "(" ^ toPreOrder lt ^ ")"
      ^ "(" ^ toPreOrder rt ^ ")"
```

システム定義のデータ型

Lists

```
infix 5 ::
datatype 'a list = nil | :: of 'a * 'a list
```

真理値

```
datatype bool = true | false
```

bool型を操作する特殊構文:

exp_1 `andalso` exp_2 $\Longrightarrow$ `case` exp_1 `of true =>` exp_2
`| false => false`

exp_1 `orelse` exp_2 $\Longrightarrow$ `case` exp_1 `of false =>` exp_2
`| true => true`

`if` exp `then` exp_1 `else` exp_2 $\Longrightarrow$ `case` exp `of true =>` exp_1
`| false =>` exp_2

そのほかのシステム定義データ型:

```
datatype order = EQUAL | GREATER | LESS
datatype 'a option = NONE | SOME of 'a
exception Option
val valOf : 'a option -> 'a
val getOpt : 'a option * 'a -> 'a
val isSome : 'a option -> bool
```

プログラミング例：辞書

```
type 'a dict = (string * 'a) tree
val enter : string * 'a * 'a dict -> 'a dict
val lookUp : string * 'a dict -> 'a option
```

- enterは与えられた辞書に項目を追加して得られる新しい辞書を返す.
- lookUpは与えられたキーの値を辞書から探す.

効率よい辞書を実現するために，二分木を，以下の性質を満たす二分探索木として用いる．任意のNode(key, L, R)の形のノードに対して,

1. Lの中のキーはkeyよりも小さい,
2. Rの中のキーはkeyよりも大きい

```
fun enter (key,v,dict) =
    case dict of
         Empty => Node((key,v),Empty,Empty)
       | Node((key',v'),L,R) =>
       if key = key' then dict
       else if key > key' then
            Node((key',v'),L, enter (key,v,R))
       else Node((key',v'),enter (key,v,L),R)


fun lookUp (key,Empty) = NONE
  | lookUp (key,Node((key',v),L,R)) =
      if key = key' then SOME v
      else if key > key' then lookUp (key,R)
      else lookUp (key,L)
```

無限のデータ構造

もちろん，我々は有限のデータしか扱えない．したがって，

```
fun fromN n = n :: (fromN (n+1));
```

などは役に立たない．

無限のデータ表現の鍵: 必用になるまで計算を遅延する．

計算遅延機構: `fn () =>` *exp*

遅延評価の幾つかの例

```
fun cond c a b = if c then a () else b();
val cond = fn : bool -> (unit -> unit)-> (unit -> unit)-> unit
cond true (fn () => print "true") (fn () => print "false");
```

無限リストを表すデータ型定義

```
datatype 'a inflist =
         NIL | CONS of 'a * (unit -> 'a inflist)
```

例:

```
fun FROMN n = CONS(n,fn () => FROMN (n+1));
# FROMN 1;
val it = CONS (1,fn) : int inflist
# val CONS(x,y) = it;
val x = 1 : int
val y = fn : unit -> int inflist
# y ();
val it = CONS (2,fn) : int inflist
```

inflist操作関数.

```
fun HD (CONS(a,b)) = a
fun TL (CONS(a,b)) = b()
fun NULL NIL = true | NULL _ = false
```

例:

```
# val naturalNumbers = FROMN 0;
val naturalNumbers = CONS (0,fn) : int inflist
# HD naturalNumbers;
val it = 0 : int
# TL naturalNumbers;
val it = (1,fn) : int inflist
# HD (TL(TL(TL it)));
val it = 4 : int
```

inflistを操作するための戦略:

1. hd, tl, nullを使ってリスト処理を設計する.
2. 以下の置き換えを行う.

$$\begin{aligned} \texttt{hd} &\Longrightarrow \texttt{HD} \\ \texttt{tl} &\Longrightarrow \texttt{TL} \\ \texttt{null} &\Longrightarrow \texttt{NULL} \\ \texttt{h::t} &\Longrightarrow \texttt{CONS(h,fn () => t)} \end{aligned}$$

例:

```
fun NTH 0 L = HD L
  | NTH n L = NTH (n - 1) (TL L)
int -> 'a inflist -> -> 'a
# NTH 100000000 naturalNumbers;
val it = 100000000 : int
```

少々複雑な例：
通常のリストの場合:

```
fun filter f l = if null l then nil
                 else if f (hd l) then
                   hd l ::  (filter f (tl l))
                 else filter f (tl l);
```

無限リストの場合:

```
fun FILTER f l = if NULL l then NIL
                 else if f (HD l) then
                   CONS(HD l,fn () => (FILTER f (TL l)))
                 else FILTER f (TL l);
```

エラトステネスの篩:
２から始まる整数の無限列に対して以下の処理を繰り返す.

1. 先頭要素を取りだして出力する.
2. 残りのリストから，先頭要素で割りきれるすべての要素を取り除く

コードの例

```
fun SIFT NIL = NIL
  | SIFT L =
    let val a = HD L
    in CONS(a, fn () =>
               SIFT (FILTER (fn x => x mod a <> 0)
                            (TL L)))
    end
```

```
# val PRIMES = SIFT (FROMN 2);
val PRIMES = CONS(2,fn) : int inflist
# TAKE 20 PRIMES;
val it = [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47,53,59,61,67,71]
       : int list;
# VIEW (10000,10) PRIMES;
val it = [104743,104759,104761,104773,104779,104789,
          104801,104803,104827,104831] : int list
```

MLの手続き的機能

参照型

'a ref型とrefデータ構成子

```
infix 3 :=
val ref :  'a -> 'a ref
val !   :  'a ref -> 'a
val :=  :  'a ref * 'a -> unit
```

簡単な例:

```
- val x = ref 1;
val x = ref 1 : int ref
- !x;
val it = 1 : int
- x:=2;
val it = () : unit
- !x;
```

$val\ it = 2 : int$

参照型を操作するプログラムの結果は，式の評価順序に依存する．

MLに置ける評価順序：

- `let val` x_1 `=` exp_1 $\cdots$ `val` x_n `=` exp_n `in` exp `end`構文では，$exp_1, \ldots, exp_n$をこの順に評価した後，expを評価する．
- 組`(`exp_1`,`$\cdots$`,`exp_n`)`やレコード`{`l_1 `=` exp_1`,`$\cdots$`,` l_n `=` exp_n`}`は左から右に評価する．
- 関数適用式epx_1 exp_2では，まずexp_1を評価し，関数`fn` x `=>` exp_0を取りだし，次にexp_2を評価し結果vを得，xをvに束縛し，exp_0を評価する．

評価順序を制御する構文

- (*exp*$_1$; $\cdots$; *exp*$_n$)
- *exp*$_1$ `before` *exp*$_2$
- `while` *exp*$_1$ `do` *exp*$_2$

プログラミング例

新しい名前を生成する関数：gensym

- `gensym : unit -> string`.
- 呼ばれる毎に，`"a"`，`"b"`，$\cdots$，`"z"`，`"aa"`，`"ab"`，$\cdots$，`"az"`，`"ba"`，$\cdots$ の順で英文字からなる新しい名前を返す.

以下のデータ構造を使用:

- 最後に生成した名前の内部表現を記憶する参照型データ`state`.
- `state`から次の名前を生成する関数`next`
- 内部表現`state`を文字列に変換する関数`toString`.

これらを使えば，gensymは以下のようにコードで実現できる．

```
local
  val state = ···
  fun toString = ···
  fun next s = ···
in
   fun gensym() = (state:=next (!state);
                   toString (!state))
end
```

文字列の内部表現

長さkの文字列$s_n s_{n-1} \cdots s_1$を$[ord(s_1), ord(s_2), \cdots, ord(s_n)]$と表現.

`next`は，以下のようにこのリストに１を足す.

```
val state = ref nil :  int list ref
fun next nil = [ord #"a"]
  | next (h::t) = if h = ord #"z" then
                      ord #"a" ::  (next t)
                  else (h+1::t)
```

参照型と参照透明性

プログラムに関する数学的分析の基本:

プログラムの意味はその構成要素の意味によって決定される.

言い換えれば,

プログラムの意味は，そのプログラムの一部を等価なもので置き換えても変化しない.

これは，参照透明性と呼ばれる，数学的推論の基本原則の例：

２つの等価な式は，文の意味を変えること無く互いに置き換えることができる.

例:

`(fn x => x = x)` exp

と

exp `=` exp

は同一の意味をもつ.

しかし, 参照型があるMLではこの性質は成り立たない,

```
# (fn x => x = x) (ref 1);
val it = true : bool
# ref 1 = ref 1;
val it = false : bool
```

参照型と値多相性

値多相性：多相型を持ちうる式は値式に限る.

```
# ref (fn x => x);
stdIn:19.1-19.16 Warning: type vars not generalized because of
   value restriction are instantiated to dummy types (X1,X2,...)
val it = ref fn : (?.X1 -> ?.X1) ref
```

もし以下のような式に多相型を許せば

```
ref (fn x => x) :  ('a -> 'a) ref
```

以下のような問題が起る.

```
val polyIdRef = ref (fn x => x);
polyIdRef := (fn x => x + 1);
(!polyIdRef) "You can't add one to me!";
```

値多相性は，この問題を回避するために導入された制限.

例外処理

例外：制御された “goto”.

`exception`文は新しい例外を定義する.

```
exception exnId
exception exnId of τ
```

`raise`は例外を生成する.

```
raise exnId
raise exnId exp
```

例:

```
# exception A;
exception A
# fun f x = raise A;
val f = fn : 'a -> 'b
# fn x => (f 1 + 1, f "a" andalso true);
val it = fn : 'a -> int * bool
```

システム定義の例外

exception name	meaning
`Bind`	bind failure
`Chr`	illegal character code
`Div`	divide by 0
`Empty`	illegal usage of `hd` and `th`
`Match`	pattern matching failure
`Option`	empty option data
`Overflow`	
`Size`	array etc too big
`Subscript`	index out of range

exp `handle` $exnPat_1$ `=>` $handler_1$
`|` $exnPat_2$ `=>` $handler_2$
$\vdots$
`|` $exnPat_n$ `=>` $handler_n$

例外処理:

1. 例外状態検出
2. システムは呼び出し系列を逆にたどり例外ハンドラを探す
3. もしハンドラ—が見つかったら，例外とハンドラ—のパターンマッチングを試みる．
4. もしマッチするパターンを持つハンドラ—が見つかれば，そのパターンを例外に束縛し対応するハンドラ—を評価する．
5. もしマッチするハンドラ—が見つからなければ式の評価を中断しトップレベルに戻る．

```
# exception Undefined;
```
exception Undefined
```
# fun strictPower n m = if n = 0 andalso m = 0 then
                           raise Undefined
                        else power n m;
```
val strictPower = fn : int -> int -> int
```
# 3 + strict_power 0 0;
```
uncaught exception Undefined
```
# 3 + (strictPower 0 0 handle Undefined => 1);
```
val it = 4 : int

プログラミング例 (1)

```
type 'a dict = (string * 'a ) tree
val enter : (string * 'a) * 'a dict -> 'a dict
val lookUp : string * 'a dict -> 'a option
exception NotFound
fun lookUp (key,Empty) = raise NotFound
  | lookUp (key,Node((key',v),L,R)) =
      if key = key' then v
      else if key > key' then lookUp (key,R)
      else lookUp (key,L)
val lookUp : string * 'a dict -> 'a
fun assoc (nil,dict) = nil
  | assoc ((h::t),dict) =
      (h, lookUp (h,dict))::  assoc (t,ditc)
      handle NotFound => (print "Undefined key."; nil)
```

プログラミング例 (2)

```
fun lookAll key dictList =
  let exception Found of 'a
      fun lookUp key Empty = ()
        | lookUp key (Node((key',v),L,R)) =
            if key = key' then raise Found v
            else if key > key' then lookUp key R
                 else lookUp key L
      in (map (lookUp key) dictList; raise NotFound)
         handle Found v => v
      end
```

練習問題 (5)

1. decompose関数を完成せよ.

```
# decompose "a(b()())(c(d()())())";
```
$val\ it = (”a”,”b()()”,”c(d()())()”) : string * string * sting$

2. pre-order表現に加え，以下の表現がある.
   - post-order表現
     木を以下の順でたどって得られる表現
     (1) 左部分木
     (2) 右部分木
     (3) ルートノード.
   - in-order表現
     木を以下の順でたどって得られる表現
     (1) 左部分木

(2) ルートノード,
(3) 右部分木.

以下の関数を定義せよ.

- post-order表現から木を再構築する関数`fromPostOrder`.
- in-order表現から木を再構築する関数`fromInOrder`.
- 木のpost-order表現を返す関数`toPostOrder`.
- 木のin-order表現を返す関数`toInOrder`.

3. 二分木上の以下の各関数を定義せよ,
   - 木の中のノードの総数を求める関数`nodes`.
   - 木の中の総ての値の和を求める関数`sumTree`.
   - 木の中のdデータに与えられた関数を適用して得られる値で置き換えて得られる木を返す関数
     `mapTree : ('a -> 'b) -> 'a tree -> 'b tree`
4. 木に対して，リストの`foldr`と類似の動作する関数`treeFold`. この関数は以下の引数を取る.
   (1) 処理対象の木t.
   (2) 木が空の時返す値z
   (3) 部分木の処理結果とノードの値から最終的な値を計算する関数f.

   この関数は以下のような動作をする.

```
# fun treeFold f z Empty = z
    | treeFold f z (Node (x,L,R)) = ···
      ...
```

val treeFold = fn : ('a * 'b * 'b -> 'b) -> 'b -> 'a tree -> 'b

(1) treeFoldの定義を完成させよ.

(2) 以下をtreeFoldを使って定義し直せ. nodes, sumTree, and mapTree using treeFold.

5. 型'a list -> 'a option を持つ以下の関数を定義せよ.

(1) リストの先頭要素を取り出す関数：car.

(2) 先頭要素を取り除いたリストを返す関数：cdr.

(3) リストの先頭要素を取り出す関数：last.

6. 与えられたリストのデータからなる辞書を生成する関数 makeDict : (string * 'a) list -> 'a dict. makeDictを

用いて`lookUp`のテストを行え.

7. 以下の関数を定義せよ.

```
type ('a,'b) dict = ('a * 'b) tree
val makeEnter : ('a * 'a -> order)
                -> 'a * 'b * ('a,'b) dict
                -> ('a,'b) dict
val makeLookUp : ('a * 'a -> order)
                 -> 'a * ('a,'b) dict -> 'b
```

ただし`makeEnter`は比較関数を受け取り辞書への登録関数を返す関数. `makeLookup`は比較関数を受け取り辞書の探索関数を返す関数.

8. `enter`と`lookUp`関数を`(int,string) dict`の型の辞書に対して定義せよ.

9. 偶数の無限リスト evenNumbers を naturalNumbers から FILTER を使って定義せよ．例えば，以下のような動作をする．

```
# NTH 10000000 evenNumbers;
```
$val\ it = 20000000 : int$

10. 無限リスト上の以下の関数を定義せよ．

- 最初のn個の要素を取り除いて得られるリストを返す関数：`DROP : int -> 'a inflist -> 'a inflist`.
- 最初のn個の要素からなるリストを返す関数：`TAKE : int -> 'a inflist -> 'a list` that returns the list of the first n elements in a given infinite list.
- n番目からm個の要素をリストとして返す関数：`VIEW : int * int -> 'a inflist -> 'a list`.

11. genSym の定義を完成せよ．

12. 文字のリストを受け取り，その文字を使って以前のgensym同様に文字列を作り出す関数

```
makeGensym  : char list -> unit -> string
```

を定義せよ．

例えば，makeGensym [#"S",#"M",#"L"]は"S"，"M"，"L"，"SS"，"SM"，"SL"，"MS"，"MM"，"ML"，"LS"，...の順で文字列を作り出す関数を返す．

13. enterの定義を変更し，同一のキーに追加しようとするとDuplicateEntry例外を発生させるようにせよ．

14. 整数のリストの積を計算する関数を定義せよ．ただし，途中で0を検出した場合，以降の処理を中断し直ちに0を返すような動作をするものとする．

SYNTAX OF THE STANDARD ML CORE LANGUAGE

表記の約束

定義は,

$$\begin{array}{rcl} \mathsf{definition} & := & \mathsf{structure1}\ \ \mathsf{explanation1} \\ & | & \mathsf{structure2}\ \ \mathsf{explanation2} \\ & & \vdots \\ & | & \mathsf{stricture}\quad\ \ \mathsf{explanation} \end{array}$$

のように書く.
省略可能な文法要素は⟨optional⟩と書く.

$[E_1,\cdots,E_n]$	はE_1からE_nまでのどれかを表す.
E*	Eの0回以上の繰り返し
$E+$	Eの1回以上の繰り返し

定数と識別子

scon	:=	*int* | *word* | *real* | *string* | *char*	(constant)
int	:=	⟨~⟩[0-9]+	(decimals)
	|	⟨~⟩0x[0-9,a-f,A-F]+	hexa decimal notation
word	:=	0w[0-9]+	(unsigned decimal)
	|	0wx[0-9,a-f,A-F]+	(unsigned hexadecimal)
real	:=	*integers* .[0-9]+ [E,e]⟨~⟩[0-9]+	(reals)
	|	*integers* .[0-9]+	
	|	*integers* [E,e]⟨~⟩[0-9]+	
char	:=	#"[*printable*,*escape*]"	(characters)
string	:=	" [*printable*,*escape*]* "	(strings)

printable　　\と"を除く印字可能な文字

$escape$	:=	\a 警告文字 (ASCII 7)
	|	\b バックスペース(ASCII 8)
	|	\t タブ(ASCII 9)
	|	\n 改行文字(ASCII 10)
	|	\v 垂直タブ(ASCII 11)
	|	\f フォームフィード(ASCII 12)
	|	\r リターン (ASCII 13)
	|	\^C 制御文字C
	|	\\ \自身
	|	\" "自身
	|	\ddd dddをコードとする文字
	|	\f $\cdots$ f\ fがフォーマット文字の時f$\cdots$fを無視
	|	\u$xxx$ UNICODE(?)

識別子のクラス

class	contents	note
vid	variables	long
$tyvar$	type variables startintg with ’	
$tycon$	type constructors	long
lab	record labels	
$strid$	structure names	long
$sigid$	signature names	alphanumeric
$funid$	functor names	alphanumeric

Long identifiers

$$longX ::= X \mid strid_1 . \cdots . strid_n . X$$

Core MLの文法規則

補助規則:

$$
\begin{array}{lcll}
xSeq & ::= & x & \text{one lement} \\
 & | & & \text{empty sequence} \\
 & | & (x_1, \cdots, x_n) & \text{finite sequence}\ (n \geq 2)
\end{array}
$$

式の文法

$$
\begin{array}{lcl}
exp & ::= & infix \\
 & | & exp\ \texttt{:}\ ty \\
 & | & exp\ \texttt{andalso}\ exp \\
 & | & exp\ \texttt{orelse}\ exp \\
 & | & exp\ \texttt{handle}\ match \\
 & | & \texttt{raise}\ exp \\
 & | & \texttt{if}\ exp\ \texttt{then}\ exp\ \texttt{else}\ exp \\
 & | & \texttt{while}\ exp\ \texttt{do}\ exp \\
 & | & \texttt{case}\ exp\ \texttt{of}\ match \\
 & | & \texttt{fn}\ match \\
match & ::= & mrule\ \langle \texttt{|}\ match \rangle \\
mrule & ::= & pat\ \texttt{=>}\ exp
\end{array}
$$

関数適用と中置演算子式

$$
\begin{array}{lcll}
appexp & ::= & atexp & \\
 & | & appexp\ atexp & \text{(left associative)} \\
infix & ::= & appexp & \\
 & | & infix\ vid\ infix &
\end{array}
$$

原子式

$$
\begin{array}{rcl}
atexp & ::= & scon \\
& | & \langle \texttt{op} \rangle\ longvid \\
& | & \texttt{\{} \langle exprow \rangle\ \texttt{\}} \\
& | & \texttt{()} \\
& | & \texttt{(}exp_1\texttt{,}\cdots\texttt{,}exp_n\texttt{)} \\
& | & \texttt{[}exp_1\texttt{,}\cdots\texttt{,}exp_n\texttt{]} \\
& | & \texttt{(}exp_1\texttt{;}\cdots\texttt{;}exp_n\texttt{)} \\
& | & \texttt{let}\ dec\ \texttt{in}\ exp_1\texttt{;}\cdots\texttt{;}exp_n\ \texttt{end} \\
& | & \texttt{(}exp\texttt{)} \\
exprow & ::= & lab\ \texttt{=}\ \ exp\ \langle\texttt{,}\ exprow\rangle
\end{array}
$$

パターン

$$
\begin{array}{lcl}
\mathit{atpat} & ::= & \mathit{scon} \\
& | & \langle \texttt{op} \rangle\ \mathit{longvid} \\
& | & \{ \langle \mathit{patrow} \rangle\ \} \\
& | & \texttt{()} \\
& | & (\mathit{pat}_1 \texttt{,} \cdots \texttt{,} \mathit{pat}_n) \\
& | & \texttt{[}\mathit{pat}_1 \texttt{,} \cdots \texttt{,} \mathit{pat}_n \texttt{]} \\
& | & (\mathit{pat}) \\
\mathit{patrow} & ::= & \ldots \\
& | & \mathit{lab}\ \texttt{=}\ \ \mathit{pat} \langle \texttt{,}\ \mathit{patrow} \rangle \\
& | & \texttt{vid} \langle \texttt{:} \mathit{ty} \rangle\ \langle \texttt{as}\ \mathit{pat} \rangle\ \langle \texttt{,} \mathit{patrow} \rangle \\
\mathit{pat} & ::= & \mathit{atpat} \\
& | & \langle \texttt{op} \rangle\ \mathit{longvid}\ \mathit{atpat} \\
& | & \mathit{pat}\ \mathit{vid}\ \mathit{pat} \\
& | & \mathit{pat}\ \texttt{:}\ \mathit{ty} \\
& | & \langle \texttt{op} \rangle\ \mathit{pat} \langle \texttt{:}\ \mathit{ty} \rangle\ \texttt{as pat}
\end{array}
$$

型

$$
\begin{array}{lcl}
ty & ::= & tyvar \\
 & | & \{\langle tyrow \rangle\} \\
 & | & tySeq\ longtycon \\
 & | & ty_1\ \texttt{*}\ \cdots\ \texttt{*}\ ty_n \\
 & | & ty\ \texttt{->}\ ty \\
 & | & \texttt{(}\ ty\ \texttt{)} \\
tyraw & ::= & lab\ \texttt{:}\ \langle\ \texttt{,}\ tyraw \rangle
\end{array}
$$

宣言 (1)

$$
\begin{array}{rcl}
dec & ::= & \texttt{val}\ tyvarSeq\ valbind \\
& | & \texttt{fun}\ tyvarSeq\ funbind \\
& | & \texttt{type}\ tybind \\
& | & \texttt{datatype}\ datbind\ \langle\texttt{withtyp}\ tybind\rangle \\
& | & \texttt{datatype}\ tycon\ \texttt{=}\ longtycon \\
& | & \texttt{exception}\ exbind \\
& | & \texttt{local}\ dec\ \texttt{in}\ dec\ \texttt{end} \\
& | & \texttt{opne}\ longstrid_1 \cdots longstrid_n \\
& | & dec\ \texttt{;}\ dec \\
& | & \texttt{infix}\ \langle d\rangle\ vid_1 \cdots vid_n \\
& | & \texttt{infixr}\ \langle d\rangle\ vid_1 \cdots vid_n \\
& | & \texttt{nonfix}\ vid_1 \cdots vid_n \\
valbind & ::= & pat\ \texttt{=}\ exp\ \langle\texttt{and}\ valbind\rangle \\
& | & \texttt{rec}\ valbind
\end{array}
$$

宣言 (2)

$$
\begin{array}{lcll}
\mathit{funbind} & ::= & \langle \texttt{op} \rangle\ \mathit{vid}\ \mathit{atpat}_{11} \cdots \mathit{atpat}_{1n}\ \langle \texttt{:}\ \mathit{ty} \rangle\ \texttt{=}\ \mathit{exp}_1 & (m, n \geq 1) \\
 & & \texttt{|}\ \langle \texttt{op} \rangle\ \mathit{vid}\ \mathit{atpat}_{21} \cdots \mathit{atpat}_{2n}\ \langle \texttt{:}\ \mathit{ty} \rangle\ \texttt{=}\ \mathit{exp}_2 & \\
 & & \texttt{|}\ \cdots & \\
 & & \texttt{|}\ \langle \texttt{op} \rangle\ \mathit{vid}\ \mathit{atpat}_{m1} \cdots \mathit{atpat}_{mn}\ \langle \texttt{:}\ \mathit{ty} \rangle\ \texttt{=}\ \mathit{exp}_m & \\
\mathit{tybind} & ::= & \mathit{tyvarSeq}\ \mathit{tycon}\ \texttt{=}\ \mathit{ty}\ \langle \texttt{and}\ \mathit{tybind} \rangle & \\
\mathit{datbind} & ::= & \mathit{tyvarSeq}\ \mathit{tycon}\ \texttt{=}\ \mathit{conbind}\ \langle \texttt{and}\ \mathit{datbind} \rangle & \\
\mathit{conbind} & ::= & \langle \texttt{op} \rangle\ \mathit{vid}\ \langle \texttt{of}\ \mathit{ty} \rangle\ \langle \texttt{|}\ \mathit{conbind} \rangle & \\
\mathit{exbind} & ::= & \langle \texttt{op} \rangle\ \mathit{vid}\ \langle \texttt{of}\ \mathit{ty} \rangle\ \langle \texttt{and}\ \mathit{exbind} \rangle & \\
 & | & \langle \texttt{op} \rangle\ \mathit{vid}\ \texttt{=}\ \langle \texttt{op} \rangle\ \mathit{lognvid}\ \langle \texttt{and}\ \mathit{exbind} \rangle &
\end{array}
$$