＊ † ‡ § ¶

ML プログラミング入門 (I)

大堀　淳

ML は，多相型システム，パターンマッチング，例外処理，参照型，モジュールシステムなどの高度な諸機能を，関数型言語の枠組の中にうまく統合した汎用プログラミング言語である．本講座では，ML でプログラムを書くために必要な基礎知識および基本的なプログラミング技術を 4 回に分けて解説する．今回は，ML の概要を説明した後，ML でのプログラミングの基本的な諸概念を説明し，ML プログラミングの基礎となる高階の関数を用いたプログラミング技術を解説する．最後に，ML の処理系の入手方法及び ML プログラミングに関する文献を紹介する．

1 はじめに

ML は，定理証明システム Edinburgh LCF [7]のメタ言語 (Meta Language) として開発され，定理証明戦略 (proof tactics) などを記述するために使用された．その最大の特徴は，Robin Milner によって提案された `let` 式を含む多相型型推論機構[18]を装備していたことである．この型推論機構は，関数適用に関する型の不整合を自動的に検出することができ，高階の関数を多用する定理証明システムのプログラムのエラーを検出する上で極めて有効であった．この性質は，定理証明システムばかりでなく一般のアプリケーションの構築にも有用であることが認識され，その後 Milner をはじめとする多くの研究者によって，LCF ML を独立の汎用プログラム言語へと拡張する努力がなされた．現在では，Standard ML[20]としてその仕様が確立している．Standard ML の主な特徴は以下の通りである．

- 関数型言語
  ML は，式を評価しその値を計算する機構を基本的計算戦略とする関数型言語である．関数は値として自由に使うことのできる第一級のデータであり，関数を引数として受け取り関数を値として返す高階の関数を定義することができる．
- 多相型型推論システム
  ML の静的な型システムは，型宣言を含まないプログラムから，そのプログラムの持つ汎用性を正確に表現する「最も一般的な多相型」を自動的に推論することができる．この機構により ML は，Lisp などの型なし言語（動的な型システムを持つ言語）の利点であるプログラミングの簡便性と柔軟性，および静的型システムの利点である高信頼性と高い実行効率の双方を実現している．
- パターンマッチング
  ML の型システムは，ユーザによるデータ型の定義とパターンマッチングによる構造データの利用機構を提供している．この機構によりプログラマは，個々のアプリケーションに適したデータ構造を簡単に定義でき，それらデータ構造の処理を，パターン

本稿は，1992 年度ソフトウェア科学会大会併設チュートリアル「本当のプログラマは ML を使うようになるか？」で使用した資料を改訂したものである．本稿の基礎となったチュートリアルの資料の作成は，筆者が沖電気工業株式会社に勤務していた時になされたものである．

Introduction to ML Programming (I)

Atsushi Ohori, 京都大学 数理解析研究所, Kyoto University, Research Institute For Mathematical Sciences.

コンピュータソフトウェア，Vol.x,No.x(199x)，pp.xx–xx.

[講座]199x 年 x 月 x 日受付.

を用いて簡潔に書くことができる.

- モジュールシステム

  ML はその言語の一部として，柔軟なモジュールシステムをサポートしている．モジュールシステムは ML の多相型システムと統合されており，モジュールとのインターフェイスも静的に型チェックされる．モジュールを引数として取り，それを使って別のモジュールを作り出すモジュールも定義することができる．この高機能なモジュールシステムを用いてプログラムを構造化することによって，大規模なシステムをより安全にかつ効率良く開発することができる.

- 参照型と例外

  純粋な関数型言語では，変数の破壊的代入や広域的なジャンプは表現できない．しかしプログラムによっては，これら機能があるとその効率や読みやすさが大幅に向上する場合がある．ML は，その静的型システムを拡張し参照型及び例外を加えることによって，関数型言語の利点を損なうことなくこれら二つの機能を導入することに成功している.

  これら機能を導入するために ML は，Miranda [32]や Haskel [11]などの遅延評価 (lazyevaluation) に基づく，いわゆる純粋な関数型言語と違い，関数適用の際に引数を必ず先に評価する適用順評価 (applicative order evaluation, call-by-value evaluation) を採用している.

さらに以上のすべての特徴が，整合性ある一つのシステムに統合されている．ML は，その意味論が厳密に定義されており，多くの望ましい性質が証明されている．特にその型システムは安全であることが証明されている．ML で書かれたプログラムは，実行時に未定義の操作を実行してシステムをクラッシュさせる様なことはないのである．この型システムの安全性は ML で書かれたプログラムに高い信頼性を与えている．ML の意味論の形式的な定義はまた，プログラム言語の新しい機能を考える上での理論的基礎ともなっている．現在 ML を基礎にして，並行計算，オブジェクト指向計算，データベース操作等の機能をプログラミング言語に統合する研究が行なわれている.

以上のような優れた特徴を持つ ML は，単なる研究や実験の為の言語ではなく，複雑なソフトウェアシステム開発する上で大きな可能性を持った実用言語でもある．ML のコンパイラは世界各地ですでに幾つか開発されている．その中でも，最近 AT&T の David MacQueen グループとプリンストン大学の Andrew Appel グループが共同で開発した ML コンパイラ Standard ML of New Jersey [1] [2]は，高速なコンパイルと高いオブジェクト効率を実現しており，十分実用コンパイラとして商業システムの開発にも使用し得るものである.

ML は種々の豊富な機能を持った高級言語である．しかしだからといって，ML でのプログラミングを修得するのは決して難しいものではない．ML は使い始める時の障壁が比較的小さい言語である．システムのインストールが完了すればすぐシステムと対話しながらプログラムを入力し，結果を見ることができる．ML は厳密な理論に基づいた洗練された言語であるが，その中核となる部分は比較的小さく，その基になる原理も少数である．これら少数の原理を身につければ，ML での対話型プログラミングを通じて，ML の諸機能や ML でのプログラミング技術を自然に身につけていくことができる．本講座の目的は，単なる ML 言語の紹介ではなく，ML で実用プログラムを書く技術の修得のきっかけを与えることである．ML にはじめて触れる人でも，ML で実用プログラムを書き始めるための十分な知識と技術を得ることができる様な資料となることを願い本稿を作成した．関数や再帰的プログラムといった計算機言語の初歩的な概念以上の知識は特に仮定していない．Lisp 等の関数型言語の経験は有用でるが，本稿を理解する上で必須ではない.

ML の仕様は Standard ML として定義されているが，システムとのインターフェースなど，処理系ごとに異なっている部分がある．本稿では，現在のところ最も完成度が高い処理系の一つである Standard ML of New Jersey システムを用いて説明する．本稿の大部分の内容はすべての Standard ML の処理系に当てはまるものである．特に Standard ML of New Jersey に依存する場合はその旨明示することにする.

本稿の説明を参考に実際にプログラムを書いてみるためのきっかけとして，いくつかのプログラミング演習を

用意した．大規模なプログラミングの例としては，簡単な関数型言語のインタープリタをとりあげた．関数型言語のインタープリタの作成は，MLでのプログラミングの演習として手頃であるばかりでなく，MLなどの関数型言語の原理を理解する上でも有用であると思われる．Standard ML of New Jerseyは誰でも無料で入手可能であり，自由に使うことができる．本稿末尾の入手方法を参考にこのコンパイラを入手し，本資料を参考にしてMLでのプログラミングを実際にお試しになることをお勧めする．

2 MLでのプログラミングの概要

Standard MLは本格的な大規模プログラムの開発に耐え得る言語であるが，同時に小規模のプログラムを手軽にテストするのにも適したシステムである．簡単なインストール処理が終われば，すぐにプログラムを入力し，結果を得ることができる．本節では，MLシステムの起動と終了，及び簡単なプログラムの入力と実行について説明する．

2.1 システムの起動と終了

Standard MLは通常対話モードで使用する．Standard MLの対話型システム(通常smlという名のコマンド)を起動すると，MLシステムは，標準出力に起動メッセージを印字した後プロンプト“-”を表示し，ユーザからの入力待ちの状態となる．

```
% sml                ( ML対話型コンパイラの起動 )
Standard ML .....   ( 起動メッセージ )
-                   ( プロンプト )
```

ユーザはこのプロンプトに続き，MLプログラムを入力する．入力されたプログラムはコンパイルされ，直ちに実行され，結果が表示される．その後，以前同様プロンプトを表示し，ユーザからの入力待ちの状態に戻る．対話型モードによるMLのプログラミングは，

1. 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. 結果の表示

というステップを繰り返すことによって行なわれる．大きなプログラムも，このステップを繰り返すことによって逐次的に作成することができる．

Standard ML of New Jersey対話型システムはインタープリタではなくコンパイラである．入力されたプログラムは機械語に翻訳され，対話型システムに含まれる実行時カーネルの下で直接実行される．このため，対話型システムでありながら，高い実行効率が得られる．翻訳されたプログラムは，対話型システムと独立に実行可能なコマンドにすることもできる．このための機能は第11節で説明する．また本稿では触れないが，Standard ML of New Jerseyはモジュール単位のコンパイルを行なうバッチモードのコンパイラも提供している．

2.2 式の入力と評価

MLのプログラムは一つの式である．複雑で大きなプログラムは多数の式の組合せによって構成され，それ自身やはり一つの式である．プログラムの実行は，式が示す値を計算することによって行なわれる．この過程を式の評価と呼ぶ．式の評価に必要なメモリー領域の割当や解放はMLの実行時カーネルによって自動的に行われ，プログラマが意識する必要はない．このメモリー領域の自動的な割当と解放は，C言語などのようにプログラムが明示的に必要領域を確保，解放する方式に比べるとオーバーヘッドが伴い，プログラムの効率を多少低下させることは事実である．しかし一方，領域の確保と解放をプログラムが行わなければならない言語では，領域の解放に漏れがあるため長時間動き続けるべきプログラムが，予期せずメモリー領域を使い尽くしシステムを停止させるという危険性を持っている．MLは，メモリーの解放が常に完全に行われるより安全なシステムでもあるとも言える．

最も簡単な式は，定数などの原子式(atomic expression)であり，それ自身が評価の結果となる．例えば以下のように自然数を入力すると，その数自身を評価の結果としてと印字し，入力待ちの状態にもどる．

```
- 21;
val it = 21 : int*
```

入力の最後の“;”はコンパイル単位の区切りを表す．システムによる出力の中の“`val it =`”は，ユーザの入力したプログラムの結果（ここでは21）が`it`という変

＊ ユーザの入力とシステムの出力を区別するために，後者には*slanted font*を使用する．

数に一時的に保持されていることを示している．これは第2.3節で説明する名前の束縛の特殊な場合である．“: int” はMLシステムが推論した型情報である．

原子式に限らず，任意に入れ子になった式を書くことができる．以下に式の入力とその評価結果の表示例を示す．

```
- 990 < 1000;
val it = false : bool
- 110 * 9;
val it = 990 : int
- (if it < 1000 then 1000 else it) * 130;
val it = 130000 : int
- (2 * 4) * (3 + 5) * 3 +
  (10 - 7) + 6;
val it = 201 : int
```

if e_1 then e_2 else e_3 は条件式である．e_1 は論理型(bool)を持つ式，e_2, e_3 はどちらも同じ型の式でなければならず，その型がこの条件式全体の型なる．この式の評価は以下の様に行われる．まず e_1 を評価し，もしその結果がtrue (真) なら式 e_2 を，false(偽) なら e_3 を評価し，その評価の結果をこの式自体の評価の結果とする．この条件式全体も式であることに注意．例えば上の例で，(if it < 1000 then 1000 else it) は整数型(int)を持つ式であり，従って整数が書ける所ならどこにでも書くことができる．最後の例のように，式の入力は2行以上にまたがってもよい．

入力したプログラムの構文に誤りがあれば，MLシステムは以下のような構文エラを報告する．

```
- 2);
std_in:1.2 Error: syntax error found at RPAREN
```

Standard ML of New Jerseyシステムでは，エラーを，エラーが検出されたファイル名，行番号とカラム位置に関する情報とともに報告する．上の例の場合は，標準入力の1行目，2カラム目でエラーが検出されたことを示している．

MLシステムは，プログラムの構文チェックのほかに，入力されたプログラムの型をチェックし，もし型エラーを含めば以下のようにエラーを報告する．

```
- 33 + "cat";
std_in:1.1-1.10 Error: operator and operand
don't agree (tycon mismatch)
  operator domain: int * int
  operand: int * string
  in expression:
    + : (33,"cat")
```

この場合は，式33 + "cat"において，整数型の組(int * int 型のデータ)に対して定義された関数+が整数型と文字列の組(int * string型のデータ)に適用されていることを報告している．この型チェックは実行時ではなく，コンパイルの段階で行なわれる．型チェックについては第5節で詳しく説明する．

2.3 名前の束縛

対話型モードでの逐次的なプログラム開発の基本は，入力した式の値に名前をつけ，それ以降の式で使用することである．このためにval宣言が用意されている．

val *name* = *expression* ;

と入力すると，システムは式 *expression* を評価しその値を計算しそれを印字するとともに，変数 *name* とその値の対応を，対話型システムのトップレベルの環境の中に保持する．この後，変数 *name* は，環境の中に保持されている値を持つ式として自由に使うことができる．この機構を名前の束縛(name binding)と呼ぶ．以下に簡単な例を示す．

```
- val mile = 1.60;
val mile = 1.6 : real
- mile;
val it = 1.6 : real
- 100.0 / mile;
val it = 62.5 : real
```

プロンプトに続き，式のみを入力した場合は，“val it =” が省略されたものとみなされ，変数 it の束縛として扱われている．

手続き型言語に慣れている読者は，val宣言によって導入された変数と手続き型プログラム言語での変数の違いに注意されたい．手続き型言語では，変数はコマンドによる操作の単位としてメモリー領域に付けられた名前であるが，MLの変数は値に付けられた名前であり，その値を返す式である．上の例では，変数mileは1.6を値として持つ式であり，1.6と同様に使用される．MLでは，第7.1節で説明するref型を使用した場合を除き，変数の値の更新といった概念は存在しない．

以下の例のように，同一の変数定義が複数回現れた場合は最後に定義された値がその変数の値となる．また宣

言されていない変数の使用は，コンパイル時にエラーとして検出される．

```
- val mile = 1600;
val mile = 1600 : int
- mile * 55;
val it = 80000
- 1 + Bogus;
std_in:1.5-1.9 Error: unbound variable
or constructor Bogus
```

2.4 ファイルからのプログラムの入力

プログラムをファイルから読み込むためには，

use "*file*" ;

と入力する．ここで *file* はプログラムテキストが入っているファイル名（パス名）である．ML システムは，ファイルの中のテキストを読み込み，";" で区切られたコンパイル単位ごとにプログラムを評価し，対話モードの場合同様に結果を表示する．ファイルのすべてのテキストの処理が終了するとファイルを閉じ，use 文を評価した時点での入力ストリームからの入力を継続する．use 文は任意に入れ子にして使用することができる．

2.5 プログラムの終了と中断

対話型コンパイラを終了するには，トップレベルでシステムで定められたファイル終了文字(通常 ^D) を入力する．

プログラムやデータに誤りがあり無限ループに陥っているような場合等，プログラムの実行を中断したい時は，システムで定義された割り込み文字（通常 ^C）を入力する．割り込み文字が入力されると，式の評価を中断しトップレベルに戻る．割り込み文字が入力されたときファイルを読み込み中であれば，すべてのファイルを閉じトップレベルに戻る．

```
- use "bad-file";
[opening bad-file]
^C
[closing bad-file]
Interrupt
-
```

3 関数を用いたプログラミング

プログラミング言語は単に計算の過程を記述するに留まらず，複雑なプログラムを構築するために必要な抽象化の概念と構造化の機構を提供する．関数型言語である ML では，関数の定義と利用に関する柔軟で強力な機構を提供することによってこの目的を達成している．本節では，ML でのプログラミングの基礎である関数の取り扱いを説明する．

3.1 関数の定義

関数の定義は fun 構文を用いて以下のように行なう．

fun *f p* = *body* ;

ここで *f* は定義される関数の名前，*p* は関数の仮引数，*body* は関数の本体である．例えば整数を 2 倍する関数 double は以下のように定義できる．

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

結果の表示の中の val double = fn は，変数 double が関数に束縛されたことを示している．式の値が関数である場合は，その内部表現はユーザにとって興味がないので，単に fn と表示される．int -> int は int 型の引数を受けとり int 型の値を計算する関数型である．

仮引数としては，上記のような変数名以外に，変数を含んだパターンが書ける．パターンは，関数が処理する引数がどのような形をしているかを示す変数を含んだ式である．例えば整数の組を受けとる関数は以下の様に定義できる．

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

int * int は int の組を表す型である．関数型構成子 (->) は組型構成子 (*) を含む他の型構成子より結合力が弱い．したがって int * int -> int は (int * int) -> int，つまり整数の組を受け取り整数を返す関数の型を表す．組以外にも種々のパターンを使うことができる．詳しくは第 6.1節，第 6.2節，第 8.2節で説明する．

定義した関数は関数適用 (function application) を通じて使用する. 型が τ_1 -> τ_2 の関数 f は型 τ_1 を持つ式 e に適用することができる. ML はラムダ計算の伝統にしたがって, 関数適用を関数と実引数を並置して $f\ e$ と書く. この関数適用自体は型 τ_2 を持つ式である. 関数適用は式の中で最も結合力が強く, 左結合する. 以下に簡単な関数適用の例を示す.

```
- double 2;
val it = 4 : int
- double 3 + 1;
val it = 7 : int
- double (double 2) + 3;
val it = 11 : int
- f (3,double 4);
val it = 14 : int
```

double 3 + 1 は double (3 + 1) の意味ではなく, (double 3) + 1 の意味である. また double (double 2) の括弧は必要である. 関数適用は左結合なので, 括弧がなければ ((double double) 2) と解釈され型エラーとなる.

3.2 関数適用の評価

ML では, 複雑なプログラムは関数の組合せで実現される. プログラムの実行は関数適用を繰り返し評価することによって行なわれる. この過程を理解することは, ML の複雑なプログラムを理解し設計する鍵である.

関数適用の評価は, まず引数を評価し値を計算し, 次に関数の仮引数名を引数を評価して得られた値に束縛し, この束縛を関数が定義された時の環境に追加して得られる環境の下で関数本体を評価することによって行なわれる. 例えば,

```
double (3 + 1)
```

の評価は, まず 3 + 1 を評価し結果の値 4 を得, double の仮引数である変数 x を値 4 に束縛し, この束縛を double が定義された時の環境に追加し double の本体 x * 2 を評価することによって行なわれる. この評価方法は作用順の評価 (applicative order evaluation) と呼ばれている.

この機構を理解するために, 以下の式の評価の過程をトレースしてみよう.

```
double (double 2) + 3
```

各変数 x_i が v_i に束縛されている環境を $\{\cdots x_i \mapsto v_i \cdots\}$ と書き, 環境 E の下での式 e の評価の結果が値 v となることを $Eval(E, e) = v$ と書くことにする. 今簡単のために, double が定義された時の環境と現在のトップレベルの環境とはともに空である仮定する. すると, 上式の評価はおよそ図 1 のように行なわれる. 与えられた式は, 原子関数 + を二つの引数 double (double 2) と 3 とへ適用したものである. この式を評価するためにまずその 2 つの引数が評価される. 最初の引数 double (double 2) を評価するため, 2 行めで $Eval$ が再帰的に用いられ, その結果が 16 行目に示されている. 4, 7, 13, 17 行目は原子定数の評価であり, 定数そのものが直ちに $Eval$ の結果となる. 6 行目と 12 行目は変数の評価の場合である. $Eval$ の第一引数で与えられた環境の中で定義されたその変数の値が, 変数を評価した結果の値となる. 8 行目, 14 行目及び 18 行目は定数関数の評価であり, 計算機の機械命令で直接実行される.

```
1  Eval({}, double (double 2) + 3)
2      ⟩ Eval({}, double (double 2))
3          ⟩     ⟩ Eval({}, (double 2))
4          ⟩     ⟩     ⟩ Eval({}, 2) = 2
5          ⟩     ⟩     ⟩ Eval({x ↦ 2}, (x * 2))
6          ⟩     ⟩     ⟩     ⟩ Eval({x ↦ 2}, x) = 2
7          ⟩     ⟩     ⟩     ⟩ Eval({x ↦ 2}, 2) = 2
8          ⟩     ⟩     ⟩ = Eval({x ↦ 2}, (2 * 2)) = 4
9          ⟩     ⟩ = 4
10         ⟩     ⟩ Eval({}, (double 4))
11         ⟩     ⟩     ⟩ Eval({x ↦ 4}, (x * 2))
12         ⟩     ⟩     ⟩     ⟩ Eval({x ↦ 4}, x) = 4
13         ⟩     ⟩     ⟩     ⟩ Eval({x ↦ 4}, 2) = 2
14         ⟩     ⟩     ⟩ = Eval({x ↦ 4}, (4 * 2)) = 8
15         ⟩     ⟩ = 8
16         ⟩ = 8
17     ⟩ Eval({}, 3) = 3
18= Eval({}, (8 + 3)) = 11
19= 11
```

図 1 関数を含む式の評価の過程

このように式の評価を再帰的に行なうことによって, 任意に複雑な式の系統的な評価が行われる. 関数型言語の評価機構の詳細は第 12.1 節で説明する.

3.3 再帰的関数

MLでは，種々の複雑な制御構造やアルゴリズムは再帰的関数で表現される．関数定義を行なうfun宣言で再帰的関数も定義できる．関数定義式fun f p = $body$ において，関数本体 $body$ の中でも，この宣言で定義している関数そのものを関数名 f を通じて使うことができる．

再帰的関数の簡単な例として，以下の定義で与えられるフィボナッチ数列 F_n を考えてみよう．

$$F_0 = 0 \quad F_1 = 1 \quad F_{n+2} = F_{n+1} + F_n$$

フィボナッチ数を計算する関数fibを，上記の定義に対応して，以下のように再帰的に書くことができる．

```
- fun fib n = if n = 0 then 0
              else if n = 1 then 1
              else fib(n - 1) + fib(n - 2);
val fib = fn : int -> int
- fib 10;
val it = 55 : int
```

この例で，関数の定義の中に現れる変数fibは，この文によって定義される関数そのものを意味している．

MLでは，二つ以上の関数が互いに他の関数を利用する相互再帰的関数も定義できる．相互再帰的な関数の例として，預金の利子が現在の預金残高によって決まる場合の残高と利子を計算する問題を考えてみよう．残高が x の時の利子を $F(x)$ とする．このとき x 万円を n 年預けたときの，n 年目の利子 I_x^n と n 年後の預金残高 A_x^n の間には以下の関係が成り立つ．

$$I_x^n = F(A_x^{n-1}) \quad (1 \leq n)$$

$$A_x^0 = x, \qquad A_x^n = A_x^{n-1} \times (1.0 + I_x^n) \quad (1 \leq n)$$

これら相互に依存した二つ関数は，キーワードandを用いて以下のように相互再帰的に定義できる．

```
- 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
```

ここでFは既に定義済みであるものとする．

二つに限らず，任意個の相互に依存する関数定義を，二つ目以降の定義でfunの代わりにandを用いて並べることにより，相互再帰的関数として定義できる．

3.4 第一級のデータとしての関数

MLでは，関数は整数などと同様データとして取り扱うことができる．この意味で関数は，MLの第一級の値(first-class values)であると言われる．例えば関数を引数として受けとり関数を結果として返すような関数を自由に定義できる．そのような関数を高階の関数と呼ぶ．MLにおいて，高度で簡潔なプログラムを書くための一つの鍵は，関数を値として扱う技法に習熟することである．

3.4.1 高階の関数

関数を値として返す関数の使い方を理解するために，自然数のべき乗 n^m の計算を考えてみよう．n^m を計算するための一つの方法は，m, n の組 (m,n) を引数として受け取り n^m を返す関数Powerを以下の様に定義することである．

```
- 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
```

この定義では，Powerを利用するために二つの整数 m, n を常に同時に与えなければない．これに対してMLでは，「m を受けとり，任意の n に対して n^m を計算する関数を返す関数」つまりint -> (int -> int)型を持つ関数powerを以下のように定義することができる．

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

この定義によって，powerが，power m nとして使う高階の関数として定義される．関数適用は左結合であり，power m nは((power m) n)の括弧が省略されたものである．関数型構成子は右結合であり，powerの型int -> int -> intは，int -> (int -> int)の括弧が省略されたものである．

このように定義されたpowerは以下のように使うことができる．

```
- power 3 2;
val it = 8 : int
```

これは「まずpower 3によって自然数を3乗する関数を作り，その関数を更に2に適用する」2重の関数適用である．Powerの場合と違い，以下の例のように

power 3 によって得られる関数を独立に値として使うことも可能である.

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

val cube = power 3 では, power 3 を評価し整数を 3 乗する関数が計算され, 変数 cube がその関数に束縛される. これ以降変数 cube は *int* -> *int* 型の関数として使用できる.

演習 1 τ_1 * τ_2 -> τ_3 の型の関数を τ_1 -> τ_2 -> τ_3 の型の関数に変換する高階の関数curryおよびその逆変換関数 uncurry を定義せよ. ただし, 任意の f, x, y について (もし型が正しければ) 常に

curry f x y = f(x,y)
uncurry f (x,y) = f x y

が成り立つものとする. ▌

次に関数を引数として取る関数の使い方を理解するため, $\Sigma_{k=1}^{n} k^3 = 1^3 + 2^3 + \cdots + n^3$ の計算を考えてみよう. $\Sigma_{k=1}^{n} k^3$ を計算する関数は, 上で定義した cube を使って以下のように定義できる.

```
- fun sum_of_cube n =
      if n <= 0 then 0
      else cube n + sum_of_cube (n - 1);
val sum_of_cube = fn : int -> int
- sum_of_cube 3;
val it = 36 : int
```

しかし, $\Sigma_{k=1}^{n} k^3 = 1^3 + 2^3 + \cdots + n^3$ はより一般的な計算スキーマ

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

において, $f(k) = k^3$ とした特殊な場合であることに着目すると, $\Sigma_{k=1}^{n} k^3$ を計算する関数は, 与えられた f から $\Sigma_{k=1}^{n} f(k)$ を計算する高階の関数を定義し, その関数を整数の三乗を計算する関数に適用することによっても得ることができる. $\Sigma_{k=1}^{n} f(k)$ を計算する高階の関数 summation は以下のように定義できる.

```
- fun summation f n =
      if n <= 0 then 0
      else f n + summation f (n - 1);
val summation = fn : (int -> int) -> int -> int
```

三乗の和を計算する関数は, この高階関数を cube に適用することによって作ることができる.

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

三乗の和を求める関数のみならず, 種々の関数の総和を求める計算を summation の簡単な適用を通じて作ることができる.

```
- val sum_of_square = summation (power 2);
val sum_of_square = fn : int -> int
- sum_of_square 3;
val it = 14
```

この summation を使ったプログラムの方が sum_of_cube などの個々の関数を直接定義する方法に比べて, プログラム全体のモジュール性, コードの簡潔性等の面から優れていることは明かであろう. このように高階の関数は, 一般性のあるプログラム構造を表現する能力が高く, 類似した構造を持つ多くの手続きを含んだプログラムの構造を抽象し, プログラム全体を簡潔にしかも分かりやすくする上で有用なことが多い.

演習 2 以下の関数を summation を使って定義せよ.
$f(n) = 1+(1+2)+(1+2+3)+\cdots+(1+2+\cdots+n)$ ▌

演習 3 今定義した summation は, 以下のより一般的な計算スキーマの特殊な場合と考えることができる.

$$\Lambda_{k=1}^{n}(h, f) = h(f(n), \cdots, h(f(2), f(1))\cdots)$$

ただし $\Lambda_{k=1}^{1}(h, f) = f(1)$ とする. 例えば $\Sigma_{k=1}^{n} f(k) = \Lambda_{k=1}^{n}(+, f)$ と考えられる.

1. $\Lambda_{k=1}^{n}(h, f(k))$ を計算する高階関数

   accumulate h f n

   を定義せよ.
2. summation を accumulate を使って定義せよ.
3. accumulate を使って以下の各計算をする関数を定義せよ.
   (a) $f_1(n) = 1 + 2 + \cdots + n$
   (b) $f_2(n) = 1 * 2 * \cdots * n$
   (c) $f_3(n, x) = 1 * x^1 + 2 * x^2 + \cdots + n * x^n$ ▌

3.4.2 式としての関数定義

fun 構文は, 関数を定義し変数をその関数値に束縛する宣言文であり, 宣言文の書けるところにしか書くことができない. これに対して, 以下の構文を持つfn式は, (名前のない) 関数の値そのものを表す式である.

fn *param* => *expression*

ここで *param* は関数の仮引数, *expression* は関数の本体である. fun 宣言と違い, この構文自体が式であ

るため，式が書けるところならどこにでも書くことができる．以下に例を示す．

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

ただしfn式では再帰的関数を直接定義することはできない．

3.5 局所データの宣言

val宣言及びfun宣言をMLシステムのトップレベルの文として実行すると，現在のトップレベルの環境に変数の束縛が追加され，それらの宣言が以下のすべてのプログラムで有効となる．このような環境の広域的な変更に対して，環境を一時的に変更する局所データの宣言ができると便利である．このためにlet式が用意されている．

```
let
   valまたはfun宣言リスト
in
   expression
end
```

このlet式の評価は，まずvalまたはfun宣言リストを順次評価し，その結果得られる名前の束縛を現在の環境に追加した環境のもとで式 *expression* を評価することによって行われる． *expression* の評価の結果がlet式の値となる． letとinの中で作られた名前の束縛は， let式の外側の環境には追加されない．

以下に関数定義の中で使用した例を示す．

```
- 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
- fact;
std_in:10.1-10.4 Error: unbound variable
or constructor fact
```

関数factはfacorialを効率よく計算するためにfactorialの定義のローカル変数として定義した補助関数であり，この定義の外側では使用できない．

演習 4 以前定義したフィボナッチ数を計算する関数fibでは， fib nを計算するためにfibが呼び出される回数は F_n 以上であることを確かめよ．上記の結果からわかるように， fibの定義は F_n を計算する関数としてあまり効率的ではない．上のfactorialの定義を参考にして， F_n を高々 n 回程度の関数呼びだしで計算する関数を定義せよ． ▌

let式自体も式であるため，式が書けるところならどこにでも書くことができる．

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

変数の宣言を局所化する機構にはlet式以外に以下のlocal特殊構文がある．

```
local
   valまたはfun宣言リスト
in
   valまたはfun宣言リスト
end
```

この構文の効果は， localとinの間の宣言をinとendの中でのみ有効にすることである．例えば以下のように使用できる．

```
- 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;
std_in:10.1-10.4 Error: unbound variable or constructor fact
```

let式との違いは，この構文自体は式ではなく値を返さないことである．第9節で説明するモジュール内部など，宣言しか書くことのできないところで用いられる．

3.6 関数内の変数のスコープ

関数定義の本体が仮引数以外の変数を含む場合，それらの変数の意味は，関数が定義された時点でのその変数に束縛されている値である．

```
- val x = 10;
val x = 10 : int
```

```
- fun f n = n + x;
val f = fn : int -> int
- val x = 99;
val x = 99 : int
- x;
val it = 99 : int
- f 3;
val it = 13 : int
```

ここで x は，この関数が使われた時点でその変数に束縛されている値を持つ式である．したがって x を再定義した後も f の意味は変化しない．変数の束縛に関する以上の規則は，静的な可視性規則と呼ばれている．

以前説明した通り，関数適用の評価は，関数が定義された時点での環境に引数の束縛を追加した環境の下で，関数本体を評価することによって行なわれる．この評価方法は，静的な可視性規則に対応している．

演習 5 静的可視性規則に対して，動的な可視性規則に基づいた変数の評価も可能である．動的な可視性規則では，変数の値は，それが実際に評価された時点での環境でその変数に束縛された値である．以下の式を ML で評価した時と動的可視性規則で評価した時の結果はそれぞれ何か？

```
let
   fun f x =
       let
          fun g y = x + y
          fun h x = g (x * 3)
       in
          h (x + 3)
       end
in
   f 10
end
```

■

3.7 2 項演算子

ML では種々の演算はすべて関数を定義することによって実現される．2 項間の演算は組型を引数として取る関数として実現される．しかし場合によっては，通常の代数における式のように，2 項演算子表現ができると便利である．この目的のために，infix 宣言が用意されている．infix n id は，識別子 id を結合度 n （大きいほど強い）の演算子として使用することを宣言する．例えば，以前 Power を int * int-> int-> int 型の関数と定義した．この Power に対して，以下のように infix 宣言を行なうことによって，Power を 2 項演算子として使用できる．

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

後に説明するように，+ などの整数演算子の結合力は 8 よりも弱い．したがって 2 Power 3 + 10 は (2 Power 3) + 10 の意味である．infix 宣言は識別子の定義の前でも後でも良い．以下のように infix 宣言を先に行った場合，関数の定義は演算子表現を用いて行われる．

```
- infix 9 hyperexp;
infix 9 hyperexp
- fun x hyperexp y =
      if x = 0 then y
      else 2 Power (x - 1) hyperexp y;
val hyperexp = fn : int * int -> int
- 2 hyperexp 2;
val it = 16 : int
- 2 Power 2 hyperexp 2;
val it = 256 : int
- (2 Power 2) hyperexp 2;
val it = 65536 : int
```

同一の結合力を持つ演算子は，ML では通常左結合する．つまり，op を演算子とすると，$e_1\ op\ e_2\ op\ \cdots\ op\ e_n$ の形をした式は $(\cdots((e_1\ op\ e_2)\ op\ e_3)\ \cdots\ op\ e_n)$ と解釈される．この規則は infix を使って定義した演算子の場合も成り立つ．例えば，

```
- 3 Power 2 Power 2;
val it = 256 : int
- 3 Power (2 Power 2);
val it = 64 : int
```

infix の代わりに infixr 宣言を用いると，右結合の演算子が定義できる．

```
- infixr 8 Power;
infixr 8 Power
- 3 Power 2 Power 2;
val it = 64 : int
- (3 Power 2) Power 2;
val it = 256 : int
```

特殊構文 op id は，識別子 id に対してなされた infix 宣言の効果を一時的に無効にする．

```
- Power;
std_in:4.1 Error: nonfix identifier required
- op Power;
```

```
val it = fn : int * int -> int
- op Power (2,3);
val it = 9 : int
```

infix 宣言の取り消しは，nonfix 宣言で行なう．

```
- nonfix Power;
nonfix Power
- Power (2,3);
val it = 9 : int
```

今後の予定

次回は，ML に組み込まれている基本データ型の操作方法，ML の多相型システム，及び ML のプログラミングで重要な役割を果たすレコード型とリスト型について解説する．第3回目では，ML に含まれる例外処理などの非関数的機能，ML におけるデータ構造の定義とその利用方法，ML のモジュールシステムの利用方法について解説する．最終の第4回目では，ML の多相型システムの特例およびオペレーティングシステムとのインターフェイス機能を解説した後，プログラミングの例として，簡単な関数型言語のインタープリタを作成する．

付録 A ML システムに関する情報

ML 処理系および ML でプログラムを書く上で参考になる資料を紹介する．

A.1 ML の処理系

ML システムはいくつか実装されているが，実用システムとしての信頼性が十分にありかつサポート体制が充実しているのは，私見では Standard ML of New Jersey システムである．このシステムは AT&T が著作権を保有するが，図 2に示すこのシステムの copyrighty notice に明示されているように，誰でも自由に入手して使うことができる．このシステムは anonymous ftp にて以下のホストから入手できる†．

ホスト名	IP アドレス	ディレクトリ
princeton.edu	128.112.128.1	/pub/ml
research.att.com	192.20.225.2	/dist/ml

システムを入手するためには，まず，上記のいずれかのホストに接続し，以下のように login し README ファイルを転送する．

```
% ftp princeton.edu
Name: anonymous
Password: 自己の電子メールアドレス
ftp> binary
ftp> get README ftp> close ftp> quit
```

この README ファイルに記載されているディレクトリの構成や必要なファイルの情報等に従い再度 login し必要なファイル転送を行なえば良い．

Standard ML of New Jersey システム以外にも種々の処理系が作られている．ML 系言語の研究のための種々の実験等の目的のためには，フランスの INRIA 研究所で開発された Caml-light[14]も便利である．このシステムは Standard ML の仕様を満たすモジュールをサポートしておらず，文法も ML の定義に準拠していない．またインタープリタであるため，作成したプログラムの効率は New Jersey システムに比べて高くはない．しかし，システムが小さくより分かりやすいため，

STANDARD ML OF NEW JERSEY COPYRIGHT NOTICE, LICENSE AND DISCLAIMER.

Copyright 1989, 1990, 1991, 1992, 1993 by AT&T Bell Laboratories

Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both the copyright notice and this permission notice and warranty disclaimer appear in supporting documentation, and that the name of AT&T Bell Laboratories or any AT&T entity not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission.

AT&T disclaims all warranties with regard to this software, including all implied warranties of merchantability and fitness. In no event shall AT&T be liable for any special, indirect or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this software.

図 2 Standrd ML of New Jersey の copyrighty notice (Standard ML of New Jersey システムの ANNOUNCE ファイルから転載)

† 1994 年 8 月現在．

ML系言語や他の関数型言語のコンパイラーに関する研究等に便利である. Caml-lightも以下のホストからanonymous ftpにて入手できる‡.

ホスト名	IPアドレス	ディレクトリ
ftp.inria.fr	192.93.2.54	lang/caml-light

上記のNew Jerseyシステムの場合と同様にREADMEファイルを転送し，その中の情報に従い，さらに必要なファイルを転送すれば良い.

A.2ML プログラミングに関する資料

本講座で得た知識をもとにMLで本格的なプログラムを書くための参考資料としては，Standard ML of New Jerseyシステムに含まれている以下のものが役に立つと思われる.

1. `./doc/manual/`
   言語の参照マニュアル.
2. `./doc/examples/`
   SMLプログラムの種々の例.
3. `./smlnj-lib/lib-manual.ps`
   SMLで書かれた種々のライブラリー使用マニュアル.
4. `src/boot/perv.sig`
   Standard ML of New Jerseyが，Standard MLの基本データ構造を実現するために定義した各種モジュールのsignatureファイル. 第4節で説明する各種の関数の型宣言はこのファイルを基にしている.

MLプログラミングに関する日本語のテキストはまだ出版されていないようである. 筆者が機会があって手にすることができた英文の解説には，[8]と[27]がある. 前者はMLの教科書が出版される以前に，MLの入門書として，欧米の大学で広く使われていたものである. [27]は最近出版されたMLの教科書である. 書店で容易に入手できるであろう. どちらも読み安い良書である. その他にも，近年になって，MLプログラミングの教科書が相次いで出版されている[28][29][33][23].

‡ 1994年8月現在.

謝　辞

本稿の草稿を丁寧に読んで下さり種々の誤りを御指摘下さった沖電気工業（株）の川北泰広氏，および査読者の方に深謝いたします.

参考文献

[1] Appel, A. W. and MacQueen, D. B.: A Standard ML Compiler, Functional Programming Languages and Computer Architecture (LNCS 274) (ed. Kahn, G.), New York (1987), Springer-Verlag.
[2] Appel, A. W. and MacQueen, D. B.: Standard ML of New Jersey, Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming (ed. Wirsing, M.), New York (August 1991), Springer-Verlag, (in press).
[3] Augustsson, L.: A compiler for Lazy ML, Symposium on LISP and Functional Programming, ACM (1984).
[4] Barendregt, H.: Lambda calculus with types, Handbook of Logic in Computer Science vol. 2, Oxford University Press (1992).
[5] Damas, L. and Milner, R.: Principal type-schemes for functional programs, Proceedings of ACM Symposium on Principles of Programming Languages (1982).
[6] Duba, B., Harper, R. and MacQueen, D.: Typing First-Class Continuations in ML, Eighteenth Annual ACM Symp. on Principles of Prog. Languages, New York (Jan 1991), ACM Press.
[7] Gordon, M., Milner, A. and Wadsworth, C.: *Edinburgh LCF: A Mechanized Logic of Computation*, Lecture Note in Computer Science, Springer-Verlag (1979).
[8] Harper, R.: Introduction to Standard ML. Technical Report ECS-LFCS-86-14, Department of Computer Science, University of Edinburgh, 1986.
[9] Harper, R., Mitchell, J. C. and E., M.: Higher-order modules and the phase distinction, Proceedings of ACM Symposium on Principles of Programming Languages (1990).
[10] Hindley, R.: The Principal Type-Scheme of an Object in Combinatory Logic, *Trans. American Mathematical Society*, **146** (Dec. 1969), 29–60.
[11] Hudak, P., Peyton Jones, S., Wadler, P., Boutel, B., Fairbairn, J., Fasel, J., Guzman, M., Hammond, K., Hughes, J., Johnsson, T., Kieburtz, D., Nikhil, R., Partain, W. and Perterson, J.: Report on Programming Language Haskel a non-strict, purely functional language version 1.2, *SIGPLAN Notices, Haskel special issue*, **27**, 5 (1992).
[12] Hoang, M., Mitchell, J. and Viswanathan, R.: Standard ML weak polymorphism and imperative constructs, Proc. Logic in Computer Science (to appear) (1993).

[13] Leroy, X. and Weise, P.: Polymorphic type inference and assignment, Proceedings of ACM Symposium on Principles of Programming Languages (1991).
[14] Leroy, X.: *The Caml Light system, release 0.6.* INRIA Rocquencourt, B.P. 105 78153 Le Chesnay France.
[15] MacQueen, D. B.: Structures and parametarization in a typed functional language, Proc. Symposium on Functional Programming Languages and Computer Architechture, Aspinas, Sweden (1981).
[16] MacQueen, D.: Modules for Standard ML, Proc. 1984 ACM Conf. on LISP and Functional Programming, New York (1984), ACM Press.
[17] MacQueen, D.: Using dependent types to express modular structure, Proceedings of Principles of Programming Languages (Jan. 1986).
[18] Milner, R.: A Theory of Type Polymorphism in Programming, *J. Comput. Syst. Sci.*, **17** (1978), 348–375.
[19] Milner, .R and Tofte, M.: *Commentary on Standard ML.* MIT Press, 1991.
[20] Milner, R., Tofte, M. and Harper, R.: *The Definition of Standard ML*, The MIT Press (1990).
[21] Mitchell, J. C. and Harper, R.: The Essence of ML, Proceedings of ACM Symposium on Principles of Programming Languages, San Diego, California (Jan. 1988).
[22] Mitchell, J.: Type systems for programming languages, Handbook of Theoretical Computer Science (ed. van Leeuwen, J.), MIT Press/Elsevier (1990), chapter 8, 365–458.
[23] Myers, C., Clack, C. and Poon, E.: *Programming with Standard ML*, Prentice Hall (1993).
[24] Ohori, A.: A compilation method for ML-style polymorphic record calculi, Proceedings of ACM Symposium on Principles of Programming Languages (1992).
[25] Paulson, L. C.: *Logic and Computation: Interactive proof with Cambridge LCF*, Cambridge University Press (1987).
[26] Paulson, L. C.: Isabelle: The next 700 theorem prover, Logic and Computer Science (ed. Odifreddi, P.), Academic Press (1990), 361–386.
[27] Paulson, L. C.: *ML for the Working Programmer*, Cambridge University Press (1991).
[28] Reade, C.: *Elements of Functional Programming*, Addison-Wesley (1989).
[29] Stansifer, R.: *ML Primer*, Prentice Hall (1992).
[30] Tofte, M.: *Operational Semantics and Polymorphic Type Inference*, PhD thesis, Department of Computer Science, University of Edinburgh (1988).
[31] Tofte, M.: Principal Signatures for Higher-order Program Modules, Proceedings of ACM Symposium on Principles of Programming Languages (1992).
[32] Turner, D.: Miranda: A non-strict functional language with polymorphic types, Functional Programming Languages and Computer Architecture, Lecture Notes in Computer Science 201, Springer-Verlag (1985).
[33] Wikstrom, A.: *Functional Programming Using Standard ML*, Prentice Hall (1987).
[34] 大堀淳: ML – 多相型システムを持つ関数型言語 –, 情報処理, **35**, 3, (1994), 215–226.
[35] 龍田真: 型理論 I ～ IV, コンピュータソフトウェア, **8**, 1,2,3,4 (1991), 25–33,40–46,3–8,56–68.
[36] 米澤明憲, 柴山悦哉: モデルと表現., 岩波口座ソフトエェア科学, 第 17 巻, 岩波書店 (1992).