Objective Caml 入門

五十嵐 淳

京都大学 工学部情報学科計算機科学コース

大学院情報学研究科知能情報学専攻

e-mail: igarashi@kuis.kyoto-u.ac.jp

平成 16 年 10 月 21 日

目 次

第1章　はじめに

1.1　関数型言語 ML と Objective Caml について

プログラミング言語 ML は元々，Edinburgh LCF という，計算機による証明記述システムのために開発されてきた．証明記述システムでは，証明そのものを記述する言語 (これを) と，証明を操作する (証明をどのような手順で行なうかなどを制御する) ための言語が使われている．前者を対象言語—object language—と呼び，後者をメタ言語—meta language—と呼ぶ．ML はこの証明操作用のメタ言語 (頭文字をとって ML) から発展してきた言語で，関数型プログラミングと呼ばれるプログラミングスタイルをサポートしている．ML は核となる部分が小さくシンプルであるため，プログラミング初心者向けの教育用に適した言語であると同時に，大規模なアプリケーション開発のためのサポート (モジュールシステム・ライブラリ) が充実している．ML の核言語は型付き λ 計算と呼ばれる，形式的な計算モデルに基づいている．このことは，言語仕様を形式的に (数学的な厳密な概念を用いて) 定義し，その性質を厳密に「証明」することを可能にしている．実際，Standard ML という標準化された言語仕様 [4] においては，(コンパイラの受理する) 正しいプログラムは決して未定義の動作をおこさない，といった性質が示されている．

この演習で学ぶのは ML の方言である Objective Caml という言語である．Objective Caml は INRIA というフランスの国立の計算機科学の研究所でデザイン開発された言語で，Standard ML とは文法的に違った言語であるが，ほとんどの概念・機能は共有している．また，Objective Caml では Standard ML には見られない，独自の拡張が多く施されており，関数型プログラミングだけでなく，オブジェクト指向プログラミングもサポートしている．またコンパイラも効率のよいものが開発されている．

1.1.1　ML・Objective Caml の特徴

ML の特徴としては，以下のようなものが挙げられる．

- 関数型プログラミングのサポート，特に，関数を引数として受けとる関数や，関数を返す関数という**高階関数**(*higher-order function*) の導入による抽象度の高いプログラミング.
- インタラクティブ・コンパイラによる対話的な開発.
- **パターンマッチング**(*pattern matching*) による，より宣言的なプログラミングのサポート.
- **静的型システム**(*static type system*) により，プログラム実行時に (ある種の) 安全性が保証される.
- **多相型システム**(*polymorphic type system*) により，プログラム中の一つの式に，複数の型を割り当てることが可能になり，コード再利用性が高まる.

- **型推論**(*type inference*) により，煩雑な型宣言を省略できる．
- 強力な**モジュール・システム**(*module system*) により，大規模プログラミングに必要な**分割コンパイル**(*separate compilation*) や，**抽象データ型**(*abstract data type*) による，プログラム部品間の情報隠蔽・言語に新たな基本データ型を加えるようなプログラミングが可能になる．また，**ファンクタ**(*functor*) と呼ばれる，**パラメータを持つモジュール**(*parameterized module*) により，再利用性の高い，大規模プログラミングがサポートされる．
- **ごみ集め**(*garbage collection*) による，自動メモリ管理．プログラマは C 言語の `malloc`/`free` などを使ったメモリ管理のように頭を悩ませる必要がない．

また，Objective Caml 独自の特徴としては，

- オブジェクト指向プログラミングのサポート．
- Tk, GTK, OpenGL などが呼びだせる GUI ライブラリ．
- バッチ・コンパイラが利用できる．
- バイトコード解釈器によるポータビリティの確保．

などが挙げられる．

1.2　参考書，資料，マニュアル

Objective Caml のマニュアル [3] (英語) は `http://www.sato.kuis.kyoto-u.ac.jp/~igarashi/class/isle4/ocamlman/` よりオンライン利用が可能なようにしてある．また `http://caml.inria.fr/` から，FAQ などの文書，処理系のソース・コンパイル済バイナリ，Objective Caml を使ったソフトウェアなどが利用できる．Objective Caml は，Caml という言語を拡張して，オブジェクト指向プログラミングの機能などを加えたものであるが，本来の Caml の教科書として [1] が出版されている．また，フランス語の Objective Caml の本が O'Reilly から出版されているが，現在，英訳プロジェクトが進行中で，オンラインで `http://caml.inria.fr/oreilly-book/` より利用可能になっている．一方，Standard ML の教科書はそれに比べれば多数出版されている [5, 6, 7] が，Objective Caml とは文法を含め微妙に異なるので ML 入門者はかえって混乱するかもしれない．

[2] は，再帰/型の概念を対話形式で平易に解説している一風変った本である．読みやすいので，この実験で興味を持ったら読んでみると面白いだろう．

1.3　環境設定

環境設定は，Emacs エディタでのプログラム編集/実行のための設定を行う．

Emacs の設定

Emacs (Mule) 上で Objective Caml プログラムの編集を助けるプログラム `tuareg-mode` (`http://www-rocq.inria.fr/~acohen/tuareg/mode/`) が `~igarashi/lib/elisp` にインストールされている．以下は `~/.emacs` に加える設定である．

```
;; append-tuareg.el - Tuareg quick installation: Append this file to .emacs.

(setq load-path (cons "~igarashi/lib/elisp/tuareg-mode" load-path))

(setq auto-mode-alist (cons '("\.ml\w?" . tuareg-mode) auto-mode-alist))
(autoload 'tuareg-mode "tuareg" "Major mode for editing Caml code" t)
(autoload 'camldebug "camldebug" "Run the Caml debugger" t)

(if (and (boundp 'window-system) window-system)
    (when (string-match "XEmacs" emacs-version)
        (if (not (and (boundp 'mule-x-win-initted) mule-x-win-initted))
            (require 'sym-lock))
        (require 'font-lock)))
```

Emacs を起動し直して，.ml という拡張子を持つファイルを開いたときにモードラインに (Tuareg) と表示されることを確認すること．

なお，以上の設定は，授業 WWW ページの授業スケジュール欄の環境設定 (http://www.sato.kuis.kyoto-u.ac.jp/~igarashi/class/isle4/configure.txt) よりオンラインで利用できるので，カット&ペーストするとよいだろう．

第2章　基本データ型，変数の宣言，簡単な関数

この章のキーワード: インタラクティブコンパイラ，型，型システム，型安全性，有効範囲，環境，関数

2.1　インタラクティブコンパイラを使う

Objective Caml 処理系には 2 種類のコンパイラが用意されている．ひとつは gcc や javac などのように，ソースファイルから実行のためのファイルを生成するバッチコンパイラ ocamlc，もうひとつは，ユーザからの入力をインタラクティブに処理する ocaml である．このインタラクティブな処理系は，(ユーザからの) プログラムの入力 → コンパイル → 実行・結果の表示，を繰り返すもので[1]，直前で実行されたプログラムの結果が次の入力時に反映されるため，開発中のテストなど，試行錯誤を伴う過程で特に便利なものである．また，後述するように，入力はキーボードからだけでなく，ファイルからの読み込みもできるので，毎回プログラムを最初から打たなければいけないなどの不便もない．

余談であるが，関数型言語処理系にはインタラクティブな処理系が用意されているものが多いようだ．

この演習では，まず ocaml の方を用いて進めていく．

2.1.1　簡単な使い方

起動方法は Emacs で M-x tuareg-run-caml (M-x はエスケープキーに続いて x をタイプする) とする (後述する Tuareg mode のバッファからは C-c C-s で起動できる)．ミニバッファ(画面の最下段)に Caml toplevel to run: というプロンプトとともに，起動するコマンドを聞かれるが (既に ocaml になっていることを確認し)，そのまま Enter キーをタイプする．すると，以下のような内容の新しいバッファが現われる．

```
            Objective Caml version 3.08.1

    #
```

はインタラクティブコンパイラの入力プロンプトである．

さて，プロンプトに続いて，簡単な式を入力してみよう．

```
# 1 + 1;;
- : int = 2
```

このテキストでは，ユーザの入力を行頭に # をつけ，タイプライター体 (abc) で，コンパイラからの出力をタイプライター斜体 (*abc*) で示す．最後の ;; は入力終了のしるしで，プロンプトからここま

[1] この手順を read-eval-print ループと呼ばれることもある．

表 2.1: コンパイラバッファ(Tuareg Interactive mode) 内キーバインディング

C-c C-c	入力途中で中断しプロンプトに戻る
C-c TAB	Caml に割り込みを入れる
C-c C-k	Caml を終了
M-p	過去に入力した式の履歴を遡る
M-n	過去に入力した式の履歴を新しい方へ辿る

での部分がコンパイル・実行される．(途中に改行があってもよい．) コンパイラの出力は，評価結果につけられた名前 (ここでは式だけを入力したので，名前をつけていないという意味である -)，式および評価結果の型 (*int*)，評価結果 (*2*) からなっている．

複雑な式は，() で囲むことで，部分式の構造を示すことができる．また，多くの演算には常識的な結合の強さが定義されていて，() を省略できる．

```
# 1 + 2 * 3;;
- : int = 7
# (1 + 2) * 3;;
- : int = 9
```

また，入力中 ;; を入力する前に Control-C を 2 回続けて入力することでコンパイルせず，プロンプトに戻ることができる．

このバッファでは式の入力を助けるコマンドがいくつか用意されており，例えば M-p, M-n で以前に入力した式を呼び出したりすることができる．(表 2.1 にキーバインディングをまとめてある．)

さて，いくつか誤った入力例についてもみていこう．

```
# 2 + 3 - ;;
  2 + 3 - ;;
          ^^
Syntax error
# 5 + "abc";;
  5 + "abc";;
      ^^^^^
This expression has type string but is here used with type int
# 4 / 0;;
Exception: Division_by_zero.
```

(端末上では下線で示されているかもしれない．) 1 番目の入力は，いわゆる文法エラーである．エラーメッセージはかなりあっさりしていて，C や Java コンパイラに比べてやや (かなり?) 不親切である．2 番目は，入力された式の構成自体は文法に沿っているものの，**型チェック**(*typechecking*) を通らなかったことを示す．Objective Caml では，+ の両辺は，整数に評価される式でなくてはならない．しかし， ここでは "abc" という文字列を加えようとしているためエラーとなっている．エラーメッセージは,「下線部 (エラーの発生した個所) は文字列型 (string) の式であるのに，整数型 (int) が必要な個所 (つまり + の右側) で使われている」ことを示している．**型**(*type*) や型チェックは Objective Caml では非常に重要な概念で，演習を通して詳しく学んでいくことになる．最後の例では，式は型チェックも通っているが，コンパイル後の実行中に**例外**(*exception*)—ここでは 0 での除算—が発生し

たことを示している．例外についても詳しく学ぶが，ここではとりあえず実行時のエラーの発生だと思っていればよい．

終了はプロンプトの出ている状態で C-c C-d を，もしくは #quit;; と入力することで行う．

```
        Objective Caml version 3.08.1

# #quit;;
Process inferior-caml finished
```

2.1.2 その他: ファイルからのプログラムの読み込み・コメント

ocaml 内では，コンパイラの動作を制御するためのディレクティブと呼ばれるいくつかの命令が利用できる．たとえば，上でででてきた #quit もディレクティブの一種である．ディレクティブは多数あるがここではファイルからのプログラム読み込みに関するふたつ #use, #cd を紹介する．詳しくはマニュアル [3] を参照のこと．

#use はファイル名を引数にとって，ファイルの内容を入力としてコンパイルを行う．

two.ml の内容

```
1 + 1;;
```

#use を使う

```
        Objective Caml version 3.08.1

# #use "two.ml";;
- : int = 2
```

ちなみに，ディレクティブは言語の一部ではなく，通常の式と組み合わせて使うことはできないことに注意．

```
# 1 + #use "two.ml";;
  1 + #use "two.ml";;
      ^
Syntax error
```

#cd は，#use と同様に文字列を引数にとって，シェルの cd コマンドと同様にカレントディレクトリを引数のものへ変更するものである．

演習のレポートは，プログラムファイルを提出することになるので，主に別ファイルにプログラムを書いて，#use でコンパイラに読み込んでテストをすることになる．この時，ファイル名の拡張子として .ml を持つファイルを読み込むと，Tuareg mode という Objective Caml プログラムの入力を支援するモードになり，ソースのインデントなどができる．コマンドは表 2.2 にまとめてある．

コメント，日本語の扱い ファイルにプログラムを書くときは，コメントを書くようにしたい．プログラム中のコメントは (* と *) で囲まれた部分である．また，コメントは入れ子になってもよいし，途中に改行をはさんでもよい．

EUC でエンコードされている限り，コメント，文字列定数に日本語を用いることができる．しかし，文字列に関しては，文字数などが正しく認識されないのでできれば使わない方が無難である．

表 2.2: Tuareg mode キーバインディング

TAB	現在の行のインデント
C-c C-p	前のフレーズ (意味のあるまとまり) へ移動
C-c C-n	次のフレーズへ移動
ESC C-h	フレーズにマーク
ESC q	フレーズをインデント
C-c . t	try 式の挿入
C-c . m	match 式の挿入
C-c . l	let 式の挿入
C-c . i	if 式の挿入
C-c . w	while 式の挿入
C-c . f	for 式の挿入
C-c . b	begin 式の挿入
C-c C-s	ocaml を起動．起動中には以下のコマンドが使用可能
C-c C-k	ocaml を終了
C-c C-b	バッファを評価 (ocaml プロセスへ送信)
C-c C-r	選択部分の評価
C-c C-e	フレーズの評価

2.2　基本データ型と演算

Objective Caml プログラムは**式**(*expression*) から，それが示す**値**(*value*)(例えば，式 `1 + 2` の値は `3` である) を計算することでプログラムの実行が進んで行く．式から値を計算する過程を**評価**(*evaluation*) という．最も簡単な式は，整数や文字列などの，基本的なデータ定数である．これらはそれ自身が値である．複雑な式は簡単な式を組み合わせることで構成する．例えば，`1 + 2` という式は二つの**部分式**(*subexpression*) `1` と `2` と `+` という二項演算子から構成されている．

本格的なプログラミングに入る前に，Objective Caml で使用される基本的なデータ (整数，実数，文字列など) とそれに対する演算 (加減乗除，文字列の結合など) を，データの型ごとに説明し，複雑な式を構成する方法をみていく．

> **メタ変数について** テキスト 中，Objective Caml 式を表記する際に，i + j のように，斜体英小文字とタイプライタ体を混ぜて表記することがある．`+` 記号が Objective Caml の式の一部の文字であることに対して，i, j は (このテキスト 中では) 任意の整数式を表すためのテキスト上での表記である．すると，例えば Objective Caml 式 `1 + 2` は i を `1`, j を `2` と考えた場合の例と考えることになる．このようなプログラムの世界の外での表記のための変数を**メタ変数**(*metavariable*) と呼ぶ．プログラムに用いられる変数 (プログラム変数) と混同しないように気をつけたい．特に，プログラム変数のためのメタ変数を用いる場合には注意が必要である．例えば，テキスト中で x, y などをプログラム変数のためのメタ変数として使用するが，x `+ 1` と書いたときには，`a + 1`, `pi + 1`, `hoge + 1`, `x + 1` など，`a`, `pi`, `hoge`, `x` という具体的な変数を使った任意の式を表わしている．

2.2.1 unit 型

unit は，() (unit value と呼ぶ) という値をただひとつの要素として含むような型である．

```
# ();;
- : unit = ()
```

この値に対して行える演算はなく，役に立たないものに思えるかもしれない．典型的な使用法にはふたつある．ひとつは，返り値に意味がないような，例えばファイルに書き込みを行なうだけの式は，**unit** 型を持つ．その意味で，C などにおける void 型と似ている[2]．また，もうひとつは，(意味のある) 引数の要らない手続きは **unit** 型の引数を取る関数として表される．

2.2.2 int 型

いわゆる整数，$\ldots, -2, -1, 0, 1, 2, \ldots$ の型である．算術演算として四則演算 `+`, `-`, `*`, `/`, 剰余を求める `mod` などが，また，ビット演算として次のようなものが用意されている．

- i `land` j, i `lor` j, i `lxor` j, `lnot` i: ビット毎の論理積/論理和/排他的論理和/論理的否定をとる．
- i `lsl` j: i の左方向への j ビットシフト $(= i * 2^{(j \bmod 32)})$.
- i `lsr` j: i の右方向への j ビット (論理) シフト．(最上位ビットには常に 0 がはいる．)
- i `asr` j: i の右方向への j ビット (算術) シフト．(最上位ビットには i の正負を保存するものがはいる．)

2.2.3 float 型

(浮動小数点表現の) 実数の型である．`3.1415` などの小数点表現と `31.415e-1` などの 10 を基底とする指数表現 $(= 31.415 \times 10^{-1})$ が使用できる．また，小数点の前の 0 は省略できない．

先述の四則演算記号は浮動小数点に対して用いることはできない．その代りに小数点をつけた `+.`, `-.`, `*.`, `/.` を使う．また，逆に整数を「そのまま」実数とみなし，`+.` などを使うこともできない．整数/実数間の変換には `int_of_float`, `float_of_int` という関数が用意されている．(つまり，C 言語などのように暗黙の型変換は存在しない．)

```
# 2.1 +. 5.9;;
- : float = 8.
# 1 +. 3.4;;
  1 +. 3.4;;
  ^
This expression has type int but is here used with type float
# float_of_int(1) +. 3.4;;
- : float = 4.4
# float_of_int 1 +. 3.4;;
- : float = 4.4
# 1 + (int_of_float 3.4);;
- : int = 4
```

[2]ただし C の void 型はそれに属する値をもたない．

表 2.3: エスケープシーケンス

\\	バックスラッシュ(\)
\'	引用符 ('), ' 内でのみ有効
\"	二重引用符 ("), " 内でのみ有効
\n	改行
\r	(行頭への) 復帰
\t	水平タブ
\b	バックスペース
ddd	*ddd* を 10 進の ASCII コードとする文字

関数の引数のまわりの () は省略可能である．省略可能，というより，そもそも関数適用の文法と括弧は関係がなく，括弧はあくまで (1+2)*3 で使うように演算の結合の強さに逆らって部分式をまとめるためのものと考えた方がよい．上の 4 番目の例でわかるように，関数適用 (float_of_int 1) は +. などの二項演算子よりも結合が強い．そのため，引数が 1+2 のような複雑な式である場合には，f 1 + 2 は (f 1)+2 の事なので，引数に括弧をつけ f(1+2) と書く必要がある．

実数演算に関しては，三角関数 sin, cos, tan, 平方根 sqrt なども予め用意されている．詳しくはマニュアルを参照のこと．

2.2.4 char 型

ASCII 文字の型で，定数として引用符 ' で囲まれた文字，もしくは表 2.3 のエスケープシーケンス (\" を除く)，また，**int** 型との変換関数 char_of_int, int_of_char が用意されている．

```
# '\120';;
- : char = 'x'
# int_of_char 'Z';;
- : int = 90
```

2.2.5 string 型

文字列の型で，定数として二重引用符 " で囲まれた文字列が使われる．文字列中の文字には，\' を除く，表 2.3 のエスケープシーケンスが使用できる．また，C とは異なり，\000 は文字列の終端を表さない．

s_1 ^ s_2 で二つの文字列 s_1, s_2 を結合した文字列に評価される．また，s.[i] で s から i 番目の文字を取り出すことができる．

```
# "Hello," ^ " World!";;
- : string = "Hello, World!"
# ("Hello," ^ " World!").[10];;
- : char = 'l'
```

2.2.6 bool 型

真偽値を示す型で，値は `true` (真), `false` (偽) の二つである．演算として，

- `not` b: b の否定を返す．
- b_1 `&&` b_2 または b_1 `&` b_2: b_1, b_2 の論理積を返す．b_1 の評価結果が `false` の場合は b_2 の評価は行わない．
- b_1 `||` b_2 または b_1 `or` b_2: b_1, b_2 の論理和を返す．b_1 の評価結果が `true` の場合は b_2 の評価は行わない．

また以下の比較演算子が用意されている．どの演算子も両辺の型が同じでなければならない．

- e_1 `=` e_2: 式 e_1, e_2 の値が等しいか判定する．
- e_1 `<>` e_2: 式 e_1, e_2 の値が等しくない場合に真を返す．
- e_1 `<` e_2, e_1 `>` e_2, e_1 `<=` e_2, e_1 `>=` e_2: e_1, e_2 の値の大小比較を行う．

```
# (not (1 < 2)) || (() = ());;
- : bool = true
# 3.2 > 5.1;;
- : bool = false
# 'a' >= 'Z';;
- : bool = true
# 2 < 4.1;;
  2 < 4.1;;
      ^^^
This expression has type float but is here used with type int
```

また，`if`-式: `if` b `then` e_1 `else` e_2 で条件分岐を行うことができる．b が `true` に評価されたときは e_1 の値，`false` であれば e_2 の値が式全体の値になる．

```
# (if 3 + 4 > 6 then "foo" else "bar") ^ "baz";;
- : string = "foobaz"
```

分岐後に評価される式 e_1, e_2 の型は一致している必要がある．また，`if`-式の `else`-節は省略可能であるが，その場合は `else ()` が隠れていると見なされる．(すなわち，その場合 `then`-節には **unit** 型の式が来なければならない．)

2.2.7 型システムと安全性

Objective Caml には**型**(*type*) の概念があり，これから学んでいくように言語の大きな特徴のひとつをなしている．型は，最も単純には，上でみた `1` は **int** 型に属するといった，プログラム中で使われるデータの分類である． この分類は，`true` に加算を行うなどの，**型エラー**(*type error*) と呼ばれる，ある種の「意味のない」操作が行われるのを防ぐのに用いられる．**型システム**(*type system*) という用語は，プログラムから**型チェック**(*typecheck*) により，型エラーの発生を防ぎ，安全にプログラムを実行するための仕組みで，言語ごとに大きく異なっている．そもそも型エラーがなんであるか，

ということも言語によって違ってくるものであることに注意．例えば，多くの言語では 0 での除算は型エラーとは見なされないことが多い．

Lisp, Perl, Postscript などの言語では，文法に即したプログラムはそのまま実行を始めることができる．そのかわり実行時に，何かの操作が行われる度に，それが型エラーを起すかどうかをチェックする．このような言語を，しばしば**動的に型づけされる言語**(*dynamically typed language*) と呼ぶ．

これに対して，C, C++, Java などの言語は，コンパイラがプログラム実行前に型チェックを行い，それを通ったもののみがコンパイルされる．このようなプログラム実行前に型チェックを行うものを，**静的に型づけされる言語**(*statically typed language*) と呼ぶ[3]．Objective Caml も静的に型づけされる言語である．

静的に型づけされる言語でも，C や C++ などは型システムが弱く，型エラーを完全に防ぐことはできない．一般に静的に型づけされる言語において，型エラーを起す操作の結果は (言語レベルで) 未定義[4] なので，C などのプログラムはクラッシュしてしまう．これに対して Objective Caml は一度型チェックを通ったプログラムは型エラーを起さない性質 (安全性) が保証されている．静的に型づけされ安全性が保証できる言語を**強く型づけされた**(*strongly typed*) 言語ということがある．

以上を，まとめると以下のようになる．動的に型づけされる言語は必然的に全ての安全性のチェックを行えるので unsafe–dynamically typed の欄が空いている．

	statically typed	dynamically typed
unsafe	C, C++, etc.	—
safe	Java, ML (Objective Caml, Standard ML), Haskell, etc.	Lisp, Scheme, Perl, PostScript, etc.

静的型システムは，プログラムの文面だけから，つまり計算前の複雑な式に対して型の整合性を判定しなければならず，見積もりがどうしても保守的にならざるを得ない．例えば

```
if 〈複雑な式〉 then 1 else "foo"
```

は，例え 〈複雑な式〉 が常に `true` を返すような式であったとしたら，`else`-節が実行されることがないので，整数が必要な文脈で使用しても実行時には何の問題もない．しかし，型システムは条件式の値に関係なく，分岐先の式の型が一致することを要求する．このため，型エラーを起さずに実行できるはずのプログラムが型チェックを通らない可能性がある．言語設計者にとっては安全なプログラムだけを受理しつつ，できる限り多くの安全なプログラムを受理できるような型システムを設計するのが，頭の悩ませどころである．

一方，動的に型づけされる言語は操作が行われる度に，実行が安全に行えるかどうかチェックをするので，チェックをまじめにやる限り安全に実行できるものの，チェックのコストを余計に払うことになる．

2.2.8　練習問題

Exercise 2.1 次の式の型と評価結果は?

[3] コンパイルと静的な型づけの間に直接の関係はない．Lisp は動的にチェックが行われるがコンパイラが存在するし，インタプリタ実行する言語に静的型システムを導入することができる．

[4] 先に見た 0 での除算は，型エラーではないとしたが，(Objective Caml では) その結果が言語内の概念である例外の発生として定義されており，しかも実行中にその発生を検知することができる．(C では OS の助けを借りないと，0 での除算の発生を検知することはできない．)

1. `float_of_int 3 +. 2.5`
2. `int_of_float 0.7`
3. `if "11" > "100" then "foo" else "bar"`
4. `char_of_int ((int_of_char 'A') + 20)`
5. `int_of_string "0xff"`
6. `5.0 ** 2.0`

Exercise 2.2 次の式は誤った式 (文法エラー，型エラー，例外を発生する) である．まず，どこが誤りかを試さずに予想せよ．次に，コンパイラでエラーメッセージを確認し，当初予想した理由とエラーメッセージと違う場合，コンパイラの解釈した誤りの理由を説明せよ．

1. `if true&&false then 2`
2. `8*-2`
3. `int_of_string "0xfg"`
4. `int_of_float -0.7`

Exercise 2.3 次の式は，括弧の付き方がおかしい，もしくは型変換関数を入れ忘れたため，型エラーが発生する，もしくは期待した結果に評価されない．各式をどう直せば，⇒ の後に示す期待した結果に評価されるか．

1. `not true && false` ⇒ `true`
2. `float_of_int int_of_float 5.0` ⇒ `5.0`
3. `sin 3.14 /. 2.0 ** 2.0 +. cos 3.14 /. 2.0 ** 2.0` ⇒ `1.0`
4. `sqrt 3 * 3 + 4 * 4` ⇒ `5` (整数)

Exercise 2.4 式 b_1 `&&` b_2 を，`if`-式と `true`, `false`, b_1, b_2 のみを用いて，同じ意味になるように書き直せ．式 b_1 `||` b_2 も同様に書き直せ．

2.3 変数の束縛

前節で学んだのは簡単な操作を組み合わせて，複雑な計算を行う式を組み立てる方法である．計算した結果の値には名前をつけておいてあとで参照することができる．これを行うのが `let` 宣言である．

2.3.1 let 宣言

まずは，let 宣言の例をみてみる．

```
# let pi = 3.1415926535;;
val pi : float = 3.1415926535
```

これは pi という名前の変数を宣言し，その変数を 3.1415926535 という実数に束縛している．コンパイラの出力として，値の名前が宣言されたことを示す val，変数名 pi，その変数が束縛された値 (もちろん 3.1415926535)，とその型が得られる．この値は，以降で pi という名前で参照することができ，pi と書くことと，3.1415926535 と書くことは同じことを意味する．

```
# pi;;
- : float = 3.1415926535
# let area_circle2 = 2.0 *. 2.0 *. pi;;
val area_circle2 : float = 12.566370614
```

一般的には

```
let x = e;;
```

という形で，宣言された変数 x を「式 e を評価した値」に束縛する．変数を宣言することには，

- 名前をつけることにより，計算結果を**抽象化**(*abstraction*) することができる．また名前によりプログラムの意味を明らかにし，間違いを減らす．
- ある計算結果を何度も使う際に，結果に名前をつけることで，計算をやり直すことなく再利用することができる．

といった意義がある．ひとつめの観点から言えば，分かりにくい変数名をつけることは避けるべきであり，たとえ一時的にしか使わない変数でも意味を反映した名前をつけるべきである．ふたつめに関して補足しておくと，変数が束縛される対象は計算結果の値であって，式自体ではないことに注意．上で,「pi と書くことと，3.1415926535 と書くことは同じことを意味する」といったのは式自体が値になっているからである．ただ，再度計算することの無駄を除けば，(ディスプレイ出力などの副作用がない限り) 式とその値は計算結果に影響をおよぼさない．

Objective Caml における変数宣言は値に名前をつけるもので，C, C++ のように，メモリ領域に名前をつけるものではなく，代入文のようなもので「中身を更新する」ことはできない．ただし同じ名前の変数を再宣言することはできる．

```
# let one = 1;;
val one : int = 1
# let two = one + one;;
val two : int = 2
# let one = "One";;
val one : string = "One"
# let three = one ^ one ^ one;;
val three : string = "OneOneOne"
```

この場合，one の値は 1 から "One" に更新されたわけではなく，同じ名前の変数宣言により前の宣言が隠されて見えなくなっただけなのである．そもそも前の宣言と型が一致していないことに注意．

and	as	assert	asr	begin	class
closed	constraint	do	done	downto	else
end	exception	external	false	for	fun
function	functor	if	in	include	inherit
land	lazy	let	lor	lsl	lsr
lxor	match	method	mod	module	mutable
new	of	open	or	parser	private
rec	sig	struct	then	to	true
try	type	val	virtual	when	while
with					

表 2.4: Objective Caml キーワード

変数のひとつひとつの使用に対して，その定義は，(以前に宣言されている物で) 最も近いものが参照される．別の言い方をすると,「let 宣言の**有効範囲**(*scope*) は (再宣言で隠されない限り) 宣言以降，ファイル (ocaml セッション) 終了まで」といわれる．このような定義の参照の仕方を**静的有効範囲**(*lexical scope, static scope*) と呼ぶ.

ところで，Objective Caml では変数の型はコンパイラが自動的に推論してくれるため，宣言する必要がない．ただ，プログラムの意味をわかりやすくするため，デバッグのため，変数の型を明示的に示しておきたいときは，変数名の後に ": ⟨型⟩" として宣言することもできる．また，複数の let-宣言はその境目がはっきりしている (次の let が来る直前で切れる) ので間に ;; をつけずに並べることができる.

```
# let pi : float = 3.1415926535
# let e = 2.718281828;;
val pi : float = 3.1415926535
val e : float = 2.718281828
```

二つの宣言がまとめてコンパイルされて結果がまとまって出力されていることに注目.

変数の名前　変数の名前として用いることができるのは,

1. 一文字目が英小文字またはアンダースコア (_) で,
2. 二文字目以降は英数字 (A...Z, a...z, 0...9)，アンダースコアまたはプライム (')

であるような任意の長さの文字列で，表 2.4 の Objective Caml の文法キーワードと _ 一文字のみからなるものを除くものである.

2.3.2 環境と lexical scoping

ここで高レベルな式の意味を離れて，名前参照がどのように実現されているかをみてみる．プログラムの実行中には，実行している時点で定義されている (有効範囲にある) 変数名とその値の組のリストがメモリ上に保存されている．このデータのことを**環境**(*environment*) という．特に，プロ

変数名	値
⋮	⋮
⋮	⋮
sin	正弦関数
max_int	1073741823
⋮	⋮

図 2.1: ocaml 起動時の大域環境

変数名	値
⋮	⋮
⋮	⋮
one	1
two	2
one	"One"
three	"OneOneOne"

図 2.2: let 宣言実行後の大域環境

グラムの一番外側における環境を**トップレベル環境**(*top-level environment*)，または**大域環境**(*global environment*) という．

例えば，ocaml を起動したときには，sin, max_int などの名前が大域環境にある状態でセッションが始まる (図 2.1).

let 宣言を実行する際には，このトップレベル環境の最後に，新しい変数とその値の組が追加される (図 2.2) ．変数の参照は，この環境を下から順番に変数を探して行く操作に対応する．そのため，同じ名前の変数が再定義された場合，上のエントリに探索が到達しないために参照することができなくなる．大域環境からエントリが削除されることはない．そのため let 宣言の有効範囲は宣言直後からプログラム終了までなのである．

2.3.3　練習問題

Exercise 2.5 次のうち変数名として有効なものはどれか．実際に let 宣言に用いて確かめよ．

```
a_2'    ____    Cat    _'_'_    7eleven    'ab2_    _
```

2.4　関数宣言

多くのプログラミング言語では，計算手順に名前をつけて抽象化することができる．Objective Caml ではこれを**関数**(*function*) と呼ぶ．

上で定義した変数 pi を使って，円の面積を求めることを考える．円の面積自体はもちろん，

〈半径〉*.〈半径〉*. pi

という式で求まるわけだが，違った半径に対して「同じような式」を何度も入力するのは，間違いのもとであり，また，その式が何を意味するのかがわかりにくくなる．そこで,「似たような計算の手順」に名前をつけ，出現個所によって違う部分 (実際の半径) は，**パラメータ化**(*parameterization*) することを考える．この「パラメータ化された計算手順」が関数である．

Objective Caml では円の面積を求める関数は次のように定義することができる．

```
# let circle_area r = (* area of circle with radius r *)
#   r *. r *. pi;;
val circle_area : float -> float = <fun>
```

関数宣言にも `let` 宣言を使用する．入力の `circle_area` が宣言された関数の名前である．関数名の後の `r` がパラメータであり，定義内で通常の変数と同じように使用することができる．`=` よりあとの式 `r *. r *. pi` が関数の**本体**(*body*) と呼ばれる部分で計算手順であるところの式を書くところである．(`;;` はいつものようにコンパイラに入力終了を知らせるものである．) コンパイラからの応答は，宣言された名前 `circle_area`，その型 `float->float`，その値 `<fun>` と並んでいる．型の中の `->` は，⟨パラメータの型⟩`->`⟨結果の型⟩ という形で，その `circle_area` が関数であることを意味しており，ここでは実数をとって実数を返すことを表している．`->` のようにより単純な型から型を構成する記号を**型構築子**(*type constructor*) と呼ぶ．基本型も 0 個の型から型を作る型構築子と考えられる．`<fun>` は，なんらかの関数であることを示している．今まで見てきた整数などとは異なり，ディスプレイに表示できる表現 (“3” など) を持たない．

宣言された関数は，組み込みの `int_of_float` などと同じように呼び出すことができる．

```
# circle_area 2.0;;
- : float = 12.566370614
```

関数呼び出し (**関数適用**(*function application*) ともいう) は，最も素朴な見方では，関数本体中のパラメータ `r` を引数 `2.0` に置き換えた式，`2.0 *. 2.0 *. pi` を評価し，その値が，呼び出し式全体の値となる．

Objective Caml では，値の束縛と同様，宣言される関数のパラメータおよび結果の型を明示的に宣言する必要がない．これは，コンパイラが**型推論**(*type inference*) を行って，上の例のように型情報を補ってくれるためである．簡単な型推論の仕組みについては，後程見ていくことにする．それでも明示的に型を宣言したい場合には，値の束縛と同様，型情報を補うことができる．

```
# let circle_area(r : float) : float = (* area of circle with radius r *)
#   r *. r *. pi;;
val circle_area : float -> float = <fun>
```

結果の型は `=` の前に記述する．また，パラメータを囲む `()` が必要になる．(型宣言をしない場合でも `()` をつけることができるが，呼び出しの時の省略を行うのと同様な理由で `()` は省略することが多い．)

ここでの，関数宣言の文法をまとめると，

`let` f ⟨*parameter*⟩ [`:` t] `=` e
　ただし ⟨*parameter*⟩ ::= x | (`(`x`:` t`)`)

となる[5]．[ ] 部分はオプションである．f は関数名を表すメタ変数，t は型を表すメタ変数である．関数名・パラメータ名として許される名前は変数の場合と同じである．(実は，変数名，関数名，パラ

[5]テキストを通じて関数宣言の文法は徐々に拡大されていく．

メータ名を区別する必要はない．) 値の名前と同じように，関数名・パラメータ名もわかりやすいものをつけ，関数が何を計算するのか，コメントを書く癖をつけたい．

> **lexical scoping について補足** 関数本体中の `pi` は，lexical scoping によって，関数宣言の時点で宣言されているものが参照される．そのため，`circle_area` のあとで `pi` を再宣言しても，`circle_area` の定義には影響がない．
>
> ```
> # let pi = 1.0;;
> val pi : float = 1.
> # circle_area 2.0;;
> - : float = 12.566370614
> ```
>
> これに対して，関数を呼び出した時点の `pi` の値を見る dynamic scoping という方式を採用している言語 (例えば Emacs Lisp) もある．dynamic scoping の下では，上の結果は，`4.0` (つまり `2.0 *. 2.0 *. 1.0`) になる．

2.4.1　練習問題

Exercise 2.6 次の関数を定義せよ．実数の切り捨てを行う関数 `floor` を用いてよい．

1. US ドル (実数) を受け取って円 (整数) に換算する関数 (ただし 1 円以下四捨五入)．(入力は小数点以下 2 桁で終わるときに働けばよい．) レートは 1$ = 111.12 円とする．
2. 円 (整数) を受け取って，US ドル (セント以下を小数にした実数) に換算する関数 (ただし 1 セント以下四捨五入)．レートは 1$ = 111.12 円とする．
3. US ドル (実数) を受け取って，文字列 "〈ドル〉 `dollars are` 〈円〉 `yen.`" を返す関数．
4. 文字を受け取って，アルファベットの小文字なら大文字に，その他の文字はそのまま返す関数 `capitalize`．(例: `capitalize 'h'` ⇒ `'H'`, `capitalize '1'` ⇒ `'1'`)

第3章　再帰による繰り返し

この章のキーワード: 局所変数，組，パターンマッチ，再帰関数

前回，非常に簡単な関数の宣言方法を学んだが，本当に簡単なことしか実現できないことに気づくだろう．例えば，複数のパラメータを持つような関数はどのようにすればよいのだろうか．また，関数本体の式が複雑になるにつれ，その意味を追うのが大変になってくることに気づくかもしれない．

今回の主な内容は，再帰的な関数定義による繰り返しの実現であるが，その前に少し寄り道をして，一時的に使用する局所変数の宣言と，複数の値をまとめて扱うためのデータ構造である**組**(*tuple*) をみていく．

3.1　局所変数と `let` 式

関数の本体内で，計算が数ステップに及び式が複雑になってくると部分式の意味を捕らえることが徐々に困難になってくる．

Objective Caml では `let` 式 (宣言ではない) によって，局所変数を宣言し，値に一時的な名前をつけることができる．

まずは，簡単な例から見ていこう．

```
# let vol_cone = (* 半径 2 高さ 5 の円錐の体積 *)
#   let base = pi *. 2.0 *. 2.0 in
#   base *. 5.0 /. 3.0;;
val vol_cone : float = 20.9439510233333337
```

“`let base =` ” 以下が `let` 式である．`base` という局所変数を宣言，底面の面積に束縛したあとで，体積を計算している．(その結果は `vol_cone` になる．)

一般的な `let` 式の形は

`let` x `=` e_1 `in` e_2

で，e_1, e_2 が `let` 式であってももちろんよい．この式は，

1. e_1 を評価し，
2. x をその値に束縛して，
3. e_2 の評価結果の値を求める，

という手順で値が求まる．変数 x の有効範囲は e_2 である．よって `vol_cone` の宣言以降では `base` は参照できない．

```
# base;;
  base;;
  ^^^^
Unbound value base
```

また，e_1 は x の有効範囲に含まれない．

もちろん，let 式は関数の本体に用いることもできる．また，let 式で局所的に使う補助的な関数を宣言することもできる．3 番目の例は，その (やや人工的な) 例である．

```
# let cone_of_heightTwo r =
#   let base = r *. r *. pi in
#   base *. 2.0 /. 3.0;;
val cone_of_heightTwo : float -> float = <fun>
# let f x =                    (* f(x) = x^3 + (x^3 + 1) *)
#   let x3 = x * x * x in
#   let x3_1 = x3 + 1 in
#   x3 + x3_1;;
val f : int -> int = <fun>
# let g x =                    (* g(x) = x^3 + (x+1)^3 *)
#   let power3 x = x * x * x in
#   (power3 x) * (power3 (x + 1));;
val g : int -> int = <fun>
```

let 式のもっとも素朴な意義は，部分式に名前をつけることによる抽象化の手段を提供することである．また二次的ではあるが，同じ部分式が複数回出現する場合にその評価を 1 度ですませられる，といった効果が得られる．また，部分式の計算方法が似ている場合には，パラメータ抽象を使って，局所関数を定義することで，プログラムの見通しがよくなる．

複数の変数宣言　let 宣言/式ともに，and キーワードを使って，複数の変数を**同時**に宣言することができる．

```
# let x = 2 and y = 1;;
val x : int = 2
val y : int = 1
# (* swap x and y;
#    the use of x is bound to the previous declaration! *)
# let x = y and y = x;;
val x : int = 1
val y : int = 2
# let z =
#   let x = "foo"
#   and y = 3.1 in
#   x ^ (string_of_float y);;
val z : string = "foo3.1"
```

各変数の使用がどこの宣言を参照しているかに注目．

let 式，関数呼出しと環境　大域変数を宣言する let 宣言は，大域環境の末尾に変数の束縛を表すペアを追加していくものであった．これに対し，let x = e_1 in e_2 の場合，x のエントリが追加されるのは e_2 の評価をする一時的な間だけである．この追加される期間が，有効範囲に対応している．

変数名	値
⋮	⋮
sin	正弦関数
max_int	1073741823
⋮	⋮

3+2 の評価中の環境

変数名	値
⋮	⋮
sin	正弦関数
max_int	1073741823
⋮	⋮
x	5

x+7 の評価中の環境

変数名	値
⋮	⋮
sin	正弦関数
max_int	1073741823
⋮	⋮

加算 (5+7) 実行後の環境

図 3.1: 式 `let x = 3 + 2 in x + 7` の実行．二重線以下が一時的に発生した束縛を表す．

変数名	値
⋮	⋮
pi	3.1415926535
c_area	<fun>
pi	1

c_area 2.0 呼出し直前の環境

変数名	値
⋮	⋮
pi	3.1415926535
r	2.0

関数呼出し直後の環境

変数名	値
⋮	⋮
pi	3.1415926535
c_area	<fun>
pi	1
area	12.566370614

実行終了後の環境

図 3.2: 式 `let area = c_area 2.0` の実行．二重線以下が一時的に発生した束縛を表す．

また関数呼出しの実行もこれと似ていて，パラメータを実引数に束縛して本体の実行を行う．ただし，有効範囲は静的に決まるため，関数が**定義された時点**での環境を用いて本体を評価する．

```
let pi = 3.1415926535;;
let c_area(r) = r *. r *. pi;;
let pi = 1;;
let area = c_area 2.0;;
```

の `let area = c_area 2.0` の実行中の環境は，図 3.2 のように表される．関数本体評価中の環境に注意すること．

3.1.1 練習問題

Exercise 3.1 次の各式においてそれぞれの変数の参照がどの定義を指すかを示せ．また評価結果を，まずコンパイラを使わずに予想せよ．その後で実際に確かめよ．

1. `let x = 1 in let x = 3 in let x = x + 2 in x * x`
2. `let x = 2 and y = 3 in (let y = x and x = y + 2 in x * y) + y`
3. `let x = 2 in let y = 3 in let y = x in let z = y + 2 in x * y * z`

Exercise 3.2 トップレベルでの以下の 2 種類の宣言の違いは何か？(ヒント: e_2 が x を含む場合を考えよ．)

- `let` x `=` e_1 `and` y `=` e_2`;;`
- `let` x `=` e_1 `let` y `=` e_2`;;`

3.2　構造のためのデータ型: 組

次に，複数の値をまとめて扱う方法を見ることにする．これによって複数のパラメータを取る関数，複数の結果を返す関数を定義することができる．

3.2.1　組を表す式

数学では，ベクトルのように，複数の「ものの集まり」から，それぞれの要素を並べたものを要素とするような，新たな集まり (集合の言葉でいえば (デカルト) 積) を定義することがある．それと同じように Objective Caml でも複数の値を並べて，新しいひとつの値を作ることができる．このような値を**組**(*tuple*) と呼ぶ．tuple は，() 内に，式を , で区切って並べて表記する．

```
# (1.0, 2.0);;
- : float * float = (1., 2.)
```

結果の型 `float * float` はこの値が,「第 1 要素 (`1.0`) が `float` で，第 2 要素 (`2.0`) も `float` であるような組」であることを示す．`*` は組の型構築子である．

組として並べられる要素は二つ以上 (メモリの許す限り) いくつでもよく，また同じ型の式でなくてもよい．また，もちろん他の値と同様に `let` で名前 をつけることができる．

```
# let bigtuple = (1, true, "Objective Caml", 4.0);;
val bigtuple : int * bool * string * float = (1, true, "Objective Caml", 4.)
# let igarashi = ("Atsushi", "Igarashi", 1, 16)
#         (* Igarashi was born on January 16 :-) *);;
val igarashi : string * string * int * int = ("Atsushi", "Igarashi", 1, 16)
```

3.2.2　パターンマッチと要素の抽出

組の中の値にアクセスするには**パターンマッチ**(*pattern matching*) の機能を使う．パターンマッチの概念は UNIX の grep などのコマンドでもみられるが，おおざっぱには

1. データの部分的な構造を記述した式 (パターン) と，
2. 与えられたデータの構造と比べることで，
3. パターン記述時には不明だった部分を簡単に知る・使う

ための機能であるとしてよいだろう[1]．例えば，`(x, y, z, w)` というパターンは，4 要素からなる組にマッチし，`x` を第 1 要素に, `y` を第 2 要素に, `z` を第 3 要素に，`w` を第 4 要素にそれぞれ束縛するパ

[1]最後の「不明だった部分を retrieve する」というのは耳慣れないかもしれないが，UNIX の egrep コマンドでは，() でマッチした文字列に番号をつけ同じパターン式の中で \1 などとして参照することができる．

ターンである．パターンは変数の束縛 (let, 関数のパラメータ) をするところに使用できる．先ほどの，bigtuple の要素は，

```
# let (i, b, s, f) = bigtuple;;
val i : int = 1
val b : bool = true
val s : string = "Objective Caml"
val f : float = 4.
```

のようにして，取り出すことができる．このパターンは値の骨組みだけで各要素の型には言及していないので，同じパターンで要素型の異なる組にマッチさせることができる．

```
# let (i, b, s, f) = igarashi;;
val i : string = "Atsushi"
val b : string = "Igarashi"
val s : int = 1
val f : int = 16
```

厳密には上のパターンは 4 つの**変数パターン**と**組パターン**を使って構成される複合的なパターンである．変数パターンは,「i」のように変数名のみから構成され,「何にでもマッチし，i をマッチした値に束縛する」ものである．[2]組パターンは，($\langle$ パターン$_1\rangle$, ... ,$\langle$ パターン$_n\rangle$) という形でより小さい部分パターンから構成される．パターンとしての解釈は,「n 個の組で，それぞれの要素が $\langle$ パターン$_i\rangle$ にマッチするとき全体がマッチし，部分パターンが作る束縛の全体をパターン全体の束縛とする」となる．また，ひとつのパターン中に変数はただ 1 度しか現れることができない．例えば，組中のふたつの要素が等しいことをパターンで表すことはできない．

```
# (* matching against a person whose first and family names are the same *)
# let (s, s, m, d) = igarashi;;
  let (s, s, m, d) = igarashi;;
          ^
This variable is bound several times in this matching
```

もうひとつ，よく使うパターンを紹介しよう．上の例では，全部の要素に名前をつけているが，プログラムの部分によっては一部の要素だけ取り出せばよい場合もある．このような場合には，変数の代りに，_ (アンダースコア) という「何にでもマッチするがマッチした内容は捨てる」**ワイルドカードパターン**(*wildcard pattern*) と呼ぶパターンを使用することができる．

```
# let (i, _, s, _) = bigtuple;;
val i : int = 1
val s : string = "Objective Caml"
```

3.2.3 組を用いた関数

次に，float のペア (2 要素の組) から各要素の平均をとる関数を定義してみよう．パラメータを今までのように変数とする代りにパターンを用いて,

```
# let average (x, y) = (x +. y) /. 2.0;;
val average : float * float -> float = <fun>
```

[2]実は，今まで使ってきた let x = ... の x も変数パターンである．つまり let の等号の左辺は常にパターンを記述する．

と宣言することができる．average の型 float * float -> float は「実数のペア float * float を受け取り，実数を返す」ことを示している．(型構築子 * の方が -> より強く結合するので，この型は (float * float) -> float と同じ意味である．) これを使って，ふたつの実数の平均は

```
# average (5.7, -2.1);;
- : float = 1.8
```

として求められる． このように，組は，引数が複数あるような関数を模倣するためによく用いられる．ここで，わざわざ「模倣」と書いたのは，実際には average は組を引数としてとる 1 引数関数であるからである．また，実は Objective Caml の関数はすべて 1 引数関数である．つまり，average は

```
# let pair = (0.34, 1.2);;
val pair : float * float = (0.34, 1.2)
# average pair;;
- : float = 0.77
```

として呼び出すこともできるのである．逆に，

```
# let average pair =
#   let (x, y) = pair in (x +. y) /. 2.0;;
val average : float * float -> float = <fun>
```

と定義することもできる[3].

組の要素として組を使うこともできる．次の定義は，(2 次元) ベクトルの加算をするものである．

```
# let add_vec ((x1, y1), (x2, y2)) = (x1 +. x2, y1 +. y2);;
val add_vec : (float * float) * (float * float) -> float * float = <fun>
```

この関数は例えば次のように呼び出される．

```
# add_vec ((1.0, 2.0), (3.0, 4.0));;
- : float * float = (4., 6.)
# let (x, y) = add_vec (pair, (-2.0, 1.0));;
val x : float = -1.66
val y : float = 2.2
```

この関数は見方によっては，複数の計算結果 (引数として与えられるふたつの実数のペアの，第 1 要素の和，と第 2 要素の和) を同時に返している関数と思うこともできる．このように，組は，複数の引数を伴う関数だけでなく，複数の結果を返す関数を模倣するのにも使用される．

3.2.4　練習問題

Exercise 3.3 2 実数の相乗平均をとる関数 geo_mean を定義せよ．

Exercise 3.4 2 行 2 列の実数行列と 2 要素の実数ベクトルの積をとる関数 prodMatVec を定義せよ．行列・ベクトルを表現する型は任意でよい．

Exercise 3.5 次のふたつの型

[3]が，この定義の場合 pair が他の場所で使われていないので最初の定義の簡潔さに勝るメリットはないだろう

- `float * float * float * float`
- `(float * float) * (float * float)`

の違いを，その型に属する値の構成法と，要素の取出し方からみて比較せよ．

Exercise 3.6 `let (x : int) = ...` などの `(x : int)` もパターンの一種である．このパターンの意味をテキストに倣って (何にマッチし，どんな束縛を発生させるか) 説明せよ．

3.3 再帰関数

関数定義は再帰的に，つまり定義のなかで自分自身を参照するように，行うことも可能である．このような**再帰関数**(*recursive function*) は，繰り返しを伴うような計算を表現するために用いることができる．

3.3.1 簡単な再帰関数

まずは簡単な例から見ていこう．自然数 n の階乗 $n! = 1 \times 2 \times \cdots \times n$ を計算する関数を考える．この式を別の見方をすると，

- 階乗が定義される数のうち最も小さい数 (つまり 1) の階乗は 1 であり[4]，
- $n! = (n-1)! \cdot n$，つまり n の階乗は $n-1$ の階乗から計算できる

ことがわかる．これは，自分自身を使って定義している再帰的な定義である．ただし，大きな数の階乗はより小さな数の階乗から定義されており，1 に関しては，自分自身に言及することなく定義されている．これは再帰定義が意味をなすための，非常に重要なポイントである．この規則を Objective Caml で定義すると，

```
# let rec fact n = (* factorial of positive n *)
#   if n = 1 then 1 else fact (n-1) * n;;
val fact : int -> int = <fun>
```

となる．(`n-1` に括弧が必要なことに注意．) 関数本体中に `fact` が出現していることがわかるだろう．また，上で述べた規則が素直にプログラムされていることがわかる．この `fact` は正の整数に対しては，正しい答えを返す．

```
# fact 4;;
- : int = 24
```

一般には，再帰関数を定義する際にはキーワード `rec` を `let` の後につけなければならないこと以外，文法は普通の関数定義と同じである．また，`rec` が有効なのは関数宣言のみである[5]．

```
# let rec x = x * x + 1;;
  let rec x = x * x + 1;;
              ^^^^^^^^^
This kind of expression is not allowed as right-hand side of 'let rec'
```

[4] 0 の階乗を 1 と定義するほうが数学では標準的である．

[5] 現在の Objective Caml の実装では，関数以外のもので再帰的に定義できる式があるがここでは扱わない．

はエラーである (そもそも「定義」といえない．また，x に関する二次方程式 $x = x^2 - 1$ を解いてくれるわけでもない．)

再帰関数を定義する際には，この階乗の例のように，何らかの意味で引数が減少していくことが重要であり，実際の関数定義は

- 最小の引数の場合の定義
- “より小さい” 引数における値からの計算式

とを場合わけを使って組み合わせることからなる．一般的なアドバイスとして，再帰関数を定義するときには「どうやって計算するか」よりも「この関数は何を計算するのか」ということを，まずはっきりさせることが重要である．

3.3.2　関数適用と評価戦略

さて，これまでに，式は値に評価されること，関数適用式は，パラメータを実引数で置き換えたような式を評価する[6]，ということは学んだが，`square(square(2))` のような式の，二つある関数適用のうち，どちらを先に評価するか，といった「どのような順番で」値に評価されるかについては説明してこなかった．そのひとつの理由は，再帰関数を導入するまでにふれた式については，評価方法に関わらず値が変らなかったからである．このような部分式の評価順序を**評価戦略**(*evaluation strategy*) という．ここですこし寄り道をして，いろいろな評価戦略をみていこう．

最も単純かつ人間が紙の上で計算する場合と近いのが,「関数を適用するときにはまず引数を値に評価する」という**値呼出し**(*call-by-value*) の戦略である．例えば，上の式は `square` の定義を

```
# let square x = x * x;;
val square : int -> int = <fun>
```

とすると，

```
square(square(2)) → square(2 * 2)
                  → square(4)
                  → 4 * 4
                  → 16
```

というように，まず，外側の `square` の引数である `square(2)` の評価を行っている．Objective Caml を含む多くのプログラミング言語では，値呼出しが使われている．

次に再帰を伴う評価について見てみよう．`fact 4` は，以下のような手順で評価される．

```
fact 4    →    if 4 = 1 then 1 else fact(4-1) * 4
          →    fact (4 - 1) * 4
          →    fact 3 * 4
          →···→ (fact (3-1) * 3) * 4
```

[6]すでに見たようにパラメータの置換は環境の仕組みで実現されているが，この節ではより単純かつ直観的な置き換えモデルを使って説明を行う．

```
→ ··· →  (fact 2 * 3)* 4
→ ··· →  ((fact (2-1) * 2) * 3) * 4
→ ··· →  ((fact 1 * 2) * 3) * 4
→ ··· →  ((1 * 2) * 3) * 4
→ ··· →  (2 * 3) * 4
→ ··· →  6
→ ··· →  24
```

値呼出しは直観的で多くのプログラミング言語で使われているものの，余計な計算を行ってしまうことがあるという欠点がある．例えば，(やや人工的な例であるが)

```
# let zero (x : int) = 0;;
val zero : int -> int = <fun>
```

のような関数は，引数がどんな整数であろうとも，0 を返すにも関わらず，`zero(square(square(2)))` のような式の評価の際，引数を計算してしまう．また，値呼び出しの言語では，条件分岐を関数で表現することはできない (練習問題参照)．

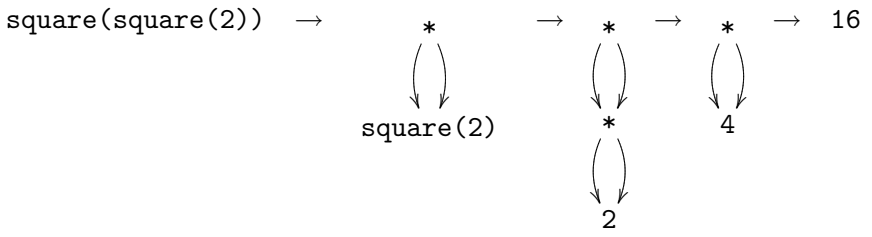
この欠点は，とにかく引数を先に評価していくこと (この性質を eagerness, strictness と呼ぶことがある) に起因する．これに対して，いまから述べるふたつの戦略は，lazy な評価と呼ばれ,「引数は使うまで評価しない」戦略である．

まず，lazy な戦略のひとつめが,「外側の関数適用から，引数を式のままパラメータに置換する」**名前呼出し**(*call-by-name*) である．この戦略の下では，先ほどの `square(square(2))` および，`zero(square(square(2)))` は，それぞれ，

```
                square(square(2))  →  square(2) * square(2)
                                   →  (2 * 2) * square(2)
                                   →  4 * square(2)
                                   →  4 * (2 * 2)
                                   →  4 * 4
                                   →  16
zero(square(square(2)))            →  0
```

のように評価される．たしかに引数を使わない関数の評価においては無駄がなくなっていることがわかる．その代わりに，計算式をそのままコピーしてしまうために，部分式 `square(2)` の計算が二度発生している．

この欠点をなくしたものが，**必要呼出し**(*call-by-need*) の戦略である．これは,「外側の関数適用から，引数を式のままパラメータに置換するが，一度評価した式は，結果を覚えておいて二度評価しない」もので，パラメータを引数で置換する代りに，引数式の共有関係を示したようなグラフで考えるとわかりやすい．



call-by-need で評価が行われる言語には Haskell, Miranda などがあり，いずれも関数型言語である．lazy な言語には無限の大きさを持つ構造などをきれいに表現できるなどの利点があるが，部分式がいつ評価されるかわかりにくいため，入出力などとの相性が悪い．また実装も call-by-value 言語に比べ複雑である．(上に示したようなグラフの書き換えに相当する graph reduction という技術がよく用いられている．)

`#trace` ディレクティブ　ここでプログラムのデバグに便利なディレクティブを紹介しておこう．`#trace` ⟨ 関数名 ⟩`;;` とすると，その関数に与えられた引数と結果を呼出された順に表示することができる．

```
# #trace fact;;
fact is now traced.
# fact 4;;
fact <-- 4
fact <-- 3
fact <-- 2
fact <-- 1
fact --> 1
fact --> 2
fact --> 6
fact --> 24
- : int = 24
```

また，`#untrace` ディレクティブで以降の表示をやめることができる．

```
# #untrace fact;;
fact is no longer traced.
# fact 5;;
- : int = 120
```

3.3.3　末尾再帰と繰り返し

上で定義した `fact` 関数の評価の様子をみるとわかるように，計算途中で「関数呼び出し後に，あとで計算される部分」`((...) * 3) * 4` といったものを何らかの形で記憶しておかなければならない．n が大きくなるとこの式の大きさも大きくなり，評価に必要な空間使用量が大きくなってしまう．ところが乗算に関しては結合則から，$((n-2)! \cdot (n-1)) \cdot n) = (n-2)! \cdot ((n-1) \cdot n)$ が成立するため $(n-2)!$ の計算にとりかかる前に，$n \cdot (n-1)$ を先に計算してしまうことで,「あとで計算する部分」の大きさを小さく保つことが可能である．このような工夫をプログラムすることを考えると，引数 n の情報以外に,「本来なら残りの計算である式の結果」の情報が必要であり，

```
# let rec facti (n, res) = (* iterative version of fact *)
#   if n = 1 then res (* equal to res * 1 *)
#   else facti (n - 1, n * res);;
val facti : int * int -> int = <fun>
```

のような定義になる．引数 res が，fact の実行過程における再帰呼出しの外側の乗算式の値に対応する．この関数は，正確には，facti (n,m) で，$n! \cdot m$ を計算する．

```
# facti (4, 1);;
- : int = 24
```

以下に，facti (4, 1) の評価の様子を示す．

```
facti (4, 1)  →  if 4 = 1 then 1 else facti(4 - 1, 4 * 1)
              →  facti (3, 4)
              →  if 3 = 1 then 4 else facti(3 - 1, 4 * 3)
              →  facti (2, 12)
              →  if 2 = 1 then 12 else facti(2 - 1, 12 * 2)
              →  facti (1, 24)
              →  if 1 = 1 then 24 else facti(1 - 1, 24 * 1)
              →  24
```

fact 4 と違い，計算の途中経過の式が小さい (引数の大きさに依存しない) ことがわかるだろう．また，どの段階においても facti の引数 n, m に関して，

$$n! \cdot m = 120 = 4!$$

が成立している．このような定義を，再帰呼出しが本体中の計算の一番最後にあることから，**末尾再帰的**(*tail-recursive*) である，という．一般には再帰関数は再帰が深くなるにつれ，メモリの使用量が増大するが，賢いコンパイラは末尾再帰関数を (自動的に) 特別扱いして，メモリの使用量が再帰の深さに関わらず固定量であるようなコードを生成することができる．また，この関数定義は C 言語などで行なう for 文などの繰り返し構文を使ったプログラムに似ているため，**反復的**(*iterative*) な定義ということもある．どんな再帰関数も反復的に定義すればよいというわけでもない．実際，素朴な再帰的定義を反復的にすると引数の数がひとつ増え，それに伴って定義のわかりやすさがかなり減少する．また，(慣れれば) fact から facti の定義を導くのはほぼ機械的なのだが，より複雑な (具体的には再帰呼び出しが複数回発生するような) 再帰関数では，単純に反復的な定義に変換することはできない．

ところで，ここでは facti をトップレベルの関数として宣言したが，これはいわば補助的な関数である．第 1 引数を 1 以外でよぶ必要がない場合は，facti を誤用されないように，

```
# let fact n = (* facti is localized *)
#   let rec facti (n, res) =
#     if n = 1 then res else facti (n - 1, res * n)
#   in facti (n, 1);;
val fact : int -> int = <fun>
```

のように，局所的に宣言するか，

```
# let rec fact (n, res) = if n = 1 then res else fact (n - 1, res * n);;
val fact : int * int -> int = <fun>
# let fact n = fact (n, 1);;
val fact : int -> int = <fun>
```

同じ名前の関数を宣言することで隠すのが，Objective Caml プログラミングの常套テクニックとして使われる．

3.3.4　より複雑な再帰

これまでに登場した再帰関数は再帰呼出しを行う場所がせいぜい 1 個所しかなかった．このような再帰の仕方を線形再帰と呼ぶことがある．ここでは再帰呼出しが 2 個所以上で行われるような再帰関数をいくつかみていく．

フィボナッチ数　フィボナッチ数列 F_i は以下の漸化式を満たすような数列である．

$$\begin{aligned} F_1 &= 1 \\ F_2 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \end{aligned}$$

n 番目のフィボナッチ数を求める関数は，

```
# let rec fib n = (* nth Fibonacci number *)
#   if n = 1 || n = 2 then 1 else fib(n - 1) + fib(n - 2);;
val fib : int -> int = <fun>
```

として宣言できる．`else` 節に再帰呼出しが 2 個所現れている．

しかし，この定義は，F_n の計算に F_{n-2} の計算が二度発生するなど，非常に多くの再帰呼出しを伴うために効率的ではない．(`fib 30` の評価を試してみよ．) これを改善したのが，次の定義である．

```
# let rec fib_pair n =
#   if n = 1 then (0, 1)
#   else
#     let (prev, curr) = fib_pair (n - 1) in (curr, curr + prev);;
val fib_pair : int -> int * int = <fun>
```

この定義では，n から F_n とともに F_{n-1} も計算する．また，線形再帰的定義になっている．

Euclid の互除法　Euclid の互除法は，自然数 m と n (ただし $m < n$) の最大公約数は，$n \div m$ の剰余と m の最大公約数に等しい性質を用いて，二整数の最大公約数を求める方法である．

組み合わせ数　n 個のもののなかから m 個のものを選びだす組合わせの場合の数 $\left(\begin{array}{c} n \\ m \end{array} \right)$ は，

$$\left(\begin{array}{c} n \\ m \end{array} \right) = \frac{n \times \cdots \times (n-m+1)}{m \times \cdots \times 1}$$

で定義される．これを再帰的に

$$\begin{aligned}\left(\begin{array}{c} n \\ 0 \end{array}\right) &= \left(\begin{array}{c} n \\ n \end{array}\right) = 1 \\ \left(\begin{array}{c} n \\ m \end{array}\right) &= \left(\begin{array}{c} n-1 \\ m \end{array}\right) + \left(\begin{array}{c} n-1 \\ m-1 \end{array}\right) \qquad \text{ただし } 0 \leq m \leq n\end{aligned}$$

と定義することもできる．

3.3.5 相互再帰

最後に，二つ以上の関数がお互いを呼び合う**相互再帰**(*mutual recursion*) をみる．相互再帰関数は，一般的に

```
let rec f1 ⟨パターン1⟩ = e1
and f2 ⟨パターン2⟩ = e2
  ⋮
and fn ⟨パターンn⟩ = en
```

という形で定義される．各本体の式 e_i には自分自身である f_i だけでなく同時に定義される $f_1, \ldots, f_n$ 全てを呼ぶことができる．

非常に馬鹿馬鹿しい例ではあるが，次の関数 `even`, `odd` は

- 0 は偶数であり，奇数ではない．
- 1 は奇数である，偶数ではない．
- $n-1$ が偶数なら n は奇数である．
- $n-1$ が奇数なら n は偶数である．

という再帰的な定義に基づき，与えられた正の整数が偶数か奇数か判定する関数である．

```
# let rec even n = (* works for positive integers *)
#   if n = 0 then true else odd(n - 1)
# and odd n =
#   if n = 0 then false else even(n - 1);;
val even : int -> bool = <fun>
val odd : int -> bool = <fun>
# even 6;;
- : bool = true
# odd 14;;
- : bool = false
```

もう少し，現実的な例として，$\arctan 1$ の展開形

$$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} \cdots + \frac{1}{4k+1} - \frac{1}{4k+3} \cdots$$

を考える．途中までの和を求める関数は，正の項を足す関数と負の項を足す関数を相互再帰的に定義できる．

```
# let rec pos n =
#   neg (n-1) +. 1.0 /. (float_of_int (4 * n + 1))
# and neg n =
#   if n < 0 then 0.0
#   else pos n -. 1.0 /. (float_of_int (4 * n + 3));;
val pos : int -> float = <fun>
val neg : int -> float = <fun>
# 4.0 *. pos 200;;
- : float = 3.14408641529876087
# 4.0 *. pos 800;;
- : float = 3.14221726314786043
```

ゆっくりと $\frac{\pi}{4}$ に収束して行く.

3.3.6 練習問題

Exercise 3.7 x は実数, n は 0 以上の整数として, x^n を計算する関数 `pow (x,n)` を以下の 2 種類定義せよ.

1. `pow` の (再帰) 呼出しを n 回伴う定義
2. `pow` の (再帰) 呼出しは約 $\log_2 n$ 回ですむ定義. (ヒント: $x^{2n} = (x^2)^n$ である. では, $x^{2n+1} =$?)

Exercise 3.8 前問の `pow` の最初の定義を反復的にした `powi` を定義せよ. (もちろん引数の数は一つ増える. 呼出し方も説明せよ.)

Exercise 3.9 `if` 式は Objective Caml の関数で表現することはできない. 以下の関数はそれを試みたものである. `fact 4` の計算の評価ステップを考え, なぜうまく計算できないのか説明せよ.

```
# let cond (b, e1, e2) : int = if b then e1 else e2;;
val cond : bool * int * int -> int = <fun>
# let rec fact n = cond ((n = 1), 1, n * fact (n-1));;
val fact : int -> int = <fun>

# fact 4;;
????
```

Exercise 3.10 `fib 4` の値呼出しによる評価ステップをテキストに倣って示せ.

Exercise 3.11 以下の関数を定義せよ.

1. Euclid の互除法で二整数の最大公約数を求める関数 `gcd`.
2. テキストの再帰的な定義で $\begin{pmatrix} n \\ m \end{pmatrix}$ を求める関数 `comb`.
3. `fib_pair` を反復的に書き直した `fib_iter`.

4. 与えられた文字列のなかで ASCII コードが最も大きい文字を返す `max_ascii` 関数．文字列から文字を取出す方法は 2.2.5 節を参照のこと．(この問題は意図的に「なにかが足りない」ように設定してあります．欲しい機能・関数があればマニュアルを調べたり，プログラム上で工夫してください．)

Exercise 3.12 `neg` を単独で用いる必要がなければ，`pos` と `neg` は一つの関数にまとめることができる．一つにまとめて定義せよ．

第4章　高階関数，多相性，多相的関数

この章のキーワード: 高階関数，多相性，多相的関数，型推論

4.1　高階関数

Objective Caml を始めとする関数型言語では，関数を整数・文字列などのデータと同じように扱うことができる．すなわち，ある関数を，他の関数への引数として渡したり，組などのデータ構造に格納したりすることができる．このことを「Objective Caml では関数は**第一級の値**(*first-class value*) である」といい，また関数を引数/返り値とするような関数を**高階関数**(*higher-order function*) と呼ぶ.

4.1.1　関数を引数とする関数

まず，前回の復習をかねて，$1^2 + 2^2 + \cdots + n^2$ を計算する関数 sqsum と，$1^3 + 2^3 + \cdots + n^3$ を計算する関数 cbsum を定義してみよう.

```
# let rec sqsum n =
#   if n = 0 then 0 else n * n + sqsum (n - 1)
# let rec cbsum n =
#   if n = 0 then 0 else n * n * n + cbsum (n - 1);;
val sqsum : int -> int = <fun>
val cbsum : int -> int = <fun>
```

よく観察するまでもなく，この二つの関数の定義は酷似していることがわかるだろう．プログラミングの重要な作業の一つは，類似の計算手順を関数によって共有することである．この二つの関数の共通部分を吸収するような関数を定義できないだろうか.

このふたつの関数の似ている点は，(1) n が 0 ならば 0 を返すこと，(2) 再帰呼出しの結果と n に関する計算結果の和がとられていること，の二点で，違いは n に関する計算手順だけである．この「計算手順」という差異をパラメータとして表現するにはどうすればよいだろうか？もともと計算手順を抽象化したものが関数であるので,「関数をパラメータとする関数」を定義すればよさそうである．それを素直に表現したのが以下の定義である.

```
# let rec sigma (f, n) =
#   if n = 0 then 0 else f n + sigma (f, n-1);;
val sigma : (int -> int) * int -> int = <fun>
```

この関数 sigma は型が示しているように，整数上の関数 f と整数 n の組を受け取って，整数を返す. (型の読み方: 型構築子 * と -> は * の方が優先度が高い．) これを使って，sqsum と cbsum は,

```
# let square x = x * x
# let sqsum n = sigma (square, n)
# let cbsum n =
```

```
#   let cube x = x * x * x in sigma (cube, n);;
val square : int -> int = <fun>
val sqsum : int -> int = <fun>
val cbsum : int -> int = <fun>
```

と定義することができる．高階関数に渡すだけの補助的な関数は，他で必要ない場合は，cbsum の例のように let-式で局所束縛を行うか，次節で説明する匿名関数を使うのがよいだろう．

```
# sqsum 5;;
- : int = 55
# cbsum 5;;
- : int = 225
```

4.1.2　匿名関数

さて，関数 sigma は，$\sum_{i=0}^{n} f(i)$ のような計算を異なる f について行いたい場合に便利である．しかし，これまでにみた関数は let を用いて定義するしかなく，具体的な関数 f ひとつひとつについて，新しい名前をつけて定義をしなければならない，というやや面倒な手順[1]を踏まなければならない．

関数型言語では，名前のない関数，**匿名関数**(*anonymous function*) を扱う手段がたいてい用意されていて，Objective Caml もその例外ではない．Objective Caml では匿名関数は

fun ⟨パターン⟩ -> e

という形をとり，⟨パターン⟩で表される引数を受け取り式 e を計算する．この「fun 式」は関数が必要な場所どこにでも使用することができる．

```
# let cbsum n = sigma ((fun x -> x * x * x), n);;
val cbsum : int -> int = <fun>
# let sq5 = ((fun x -> x * x), 5) in
#   sigma sq5;;
- : int = 55
# (fun x -> x * x) 7;;
- : int = 49
```

2,3 番目の例のように，匿名関数は組の要素にもなり，また (あまり実用的な意味はないが) 直接適用することもできる．また，いずれの例でも fun のまわりの () が必要である (これがないと，“, n” が関数本体の一部と思われてしまい，x * x * x と n の組を返す関数として解釈されてしまう．一般的には，fun はできる限り先まで関数本体と思い込もうとするので，適宜 () を使ってどこまで関数本体か示してやる必要がある．)

実は，let による関数宣言

```
let f x = e
```

は

```
let f = fun x -> e
```

の略記法である．このことから，関数を構成すること (fun) と，それに名前をつけること (let) は，必ずしも関連していない別の仕組みであることがわかる．

[1] 関数名を考えるのって面倒じゃありませんか？

4.1.3 カリー化と関数を返す関数

Objective Caml の関数は全て一引数であるため，引数が二つ以上必要な関数を定義するには組を用いることをみてきた．ここでは,「関数を返す関数」を使って引数が複数ある関数を模倣できる方法をみる．このような「関数を返す関数」を有名な論理学者 Haskell Curry の名をとって**カリー化関数**(*curried function*) と呼ぶ[2].

基本的なアイデアは「x と y を受け取り e を計算する関数」を「x を受け取ると,『y を受け取って e を計算する関数』を返す関数」として表現することである．具体的な例として，二つの文字列 s_1, s_2 から s_1s_2 のような連結をした文字列を返す関数を考えてみよう．これまでにみてきた，組を使った定義では,

```
# let concat (s1, s2) = s1 ^ s2 ;;
val concat : string * string -> string = <fun>
```

と定義され，型はまさに「文字列を二つ (組にして) 受け取り文字列を返す」ことを表している．使う場合も二つの文字列を同時に指定して `concat ("abc", "def")` のように呼び出す.

さて，この関数を カリー化関数として定義してみよう.

```
# let concat_curry s1 = fun s2 -> s1 ^ s2;;
val concat_curry : string -> string -> string = <fun>
```

`concat_curry` は `fun` 式を用いて,「`s2` を受け取って (既に受け取り済の) `s1` と連結するような関数」を返している．`concat_curry` の型は `string -> (string -> string)` と同じで，そのことを示している．この関数を呼び出すには，2 回の関数適用を経て,

```
# (concat_curry "abc") "def";;
- : string = "abcdef"
```

のように行う．(...) 内の関数適用で,「"abc" と与えられた引数を連結するような関数」が返ってきており，外側の関数適用 (...) `"def"` で文字列の連結が行われる．

カリー化関数は，組を用いた定義と違って，最初のいくつかのの引数を固定したような関数を作りたい時に簡潔に実現できる．例えば，敬称 Mr. を名前 (文字列) に付加する関数を

```
# let add_title = concat_curry "Mr. ";;
val add_title : string -> string = <fun>
# add_title "Igarashi";;
- : string = "Mr. Igarashi"
```

と定義することができる．`add_title` は引数の置き換えモデルにしたがって, `fun s2 -> "Mr. " ^ s2` という関数に束縛されていると考えることができる．このように，カリー化された関数の一部の引数を与えて，特化した関数を作ることを**部分適用**(*partial application*) と呼ぶ.

カリー化関数の型は，読み方によって,「二引数の関数の型」と読むことも,「関数を返す関数」と読むこともできる.

[2] この考えは Schönfinkel という人が発見したらしいがなぜが Curry 化と呼ばれている.

関数定義の文法拡張　上のカリー化関数の定義方法を，fun を入れ子にすることによって，三引数，四引数の関数の表現に拡張していくことも可能である．

実は，Objective Caml では，fun を入れ子にする代りに，let や fun でのパラメータパターンを複数個並べることによって，カリー化関数をより簡潔に定義することができる．先の例は，

```
# let concat_curry s1 s2 = s1 ^ s2;;
val concat_curry : string -> string -> string = <fun>
```

と定義することも，

```
# let concat_curry = fun s1 s2 -> s1 ^ s2;;
val concat_curry : string -> string -> string = <fun>
```

と定義することも可能である．一般的には，

fun ⟨パターン$_1$⟩ -> fun ⟨パターン$_2$⟩ -> ... fun ⟨パターン$_n$⟩ -> e

は

fun ⟨パターン$_1$⟩⟨パターン$_2$⟩ ...⟨パターン$_n$⟩ -> e

と同じである．(let についても同様にパターンを空白で区切って並べることができる．)

また，関数適用も (((f x) y) z) と書く代りに f x y z と，括弧を省略することができる．(別の言い方をすると，関数適用式は**左結合する**．) 関数型構築子は，既に見たように，右結合し，t1 -> t2 -> t3 -> t4 は t1 -> (t2 -> (t3 -> t4)) を意味する．

中置/前置演算子　これまで，なんの説明もなしに使ってきたが，+, ^ などの**中置演算子**(*infix operator*) は，内部ではカリー化された関数 (int->int->int などの型をもつ) として定義されている．さらに，プログラマが新たな中置演算子を定義したり，(勧められないが) + などを再宣言することすらも可能である．

中置演算子は (と) で囲むことによって，通常の関数として (前置記法) で使うことができる．

```
# (+);;
- : int -> int -> int = <fun>
# ( * ) 2 3;;
- : int = 6
```

* の前後に空白が入っているのは，コメントの開始/終了と区別するためである．

中置演算子として使用可能な記号は，mod, lor, or などの前章までに中置演算子として紹介したキーワード，もしくは

- 1 文字目が =, <, >, @, ^, |, &, +, -, *, /, $, % のいずれかで，
- 2 文字目以降が !, $, %, *, +, -, ., /, :, <, =, >, ?, @, ^, |~ のいずれか

を満たす文字列である．

定義をするには (⟨ 中置演算子 ⟩) を普通の名前だと思って行う．

```
# let (^-^) x y = x * 2 + y * 3;;
val ( ^-^ ) : int -> int -> int = <fun>
# 9 ^-^ 6;;
- : int = 36
```

また，前置演算子といって，定義するときや単独で関数値として使うときは () が必要な，記号列からなる名前が用意されている．

```
# let ( !! ) x = x + 1;;
val ( !! ) : int -> int = <fun>
# !!;;
Characters 2-4:
  !!;;Syntax error
# (!!);;
- : int -> int = <fun>
# !! 3;;
- : int = 4
```

前置演算子は -, -. もしくは

- 1 文字目が !, ?, ~ のいずれかで，
- 2 文字目以降が !, $, %, *, +, -, ., /, :, <, =, >, ?, @, ^, |~ のいずれか

からなる文字列である．一見，前置演算子は関数の名前として記号が使うためのものだけのように見えるが，実際は構文解析で違いが現れる．前置演算子は通常の関数適用よりも結合の優先度が高く，`f !! x` は，`f (!! x)` と解釈される．(`f g x` が `(f g) x` と解釈されることと比較せよ．)

中置演算子同士の優先度は，名前から決まる．表 4.1 は優先度の高いものから並べたものである．コンストラクタなど，未出の概念，記号がでてきているがとりあえず無視しておいてよい．

表 4.1: 演算子の優先順位と結合

演算子	結合
前置演算子	–
関数適用	左
コンストラクタ適用	–
前置演算子としての -, -.	–
** で始まる名前	右
*, /, %, で始まる名前および mod	左
+, - で始まる名前	左
::	右
@, ^ で始まる名前	右
=, < など比較演算子, その他の中置演算子	左
not	–
&, &&	左
or, ||	左
,	–
<-, :=	右
if	–
;	右
let, match, fun, function, try	–

4.1.4 Case Study: Newton-Raphson 法

高階関数の有効な例として，方程式の近似解を求める Newton-Raphson 法をプログラムしてみよう．Newton-Raphson 法は，微分可能な関数 f に対して，方程式 $f(x) = 0$ の解を求める方法であり，

$$g(x) = x - \frac{f(x)}{f'(x)}$$

の不動点 ($g(a) = a$ なる a) を求める．(もしくは漸化式 $x_n = x_{n-1} - f(x_{n-1})/f'(x_{n-1})$ の極限を求める．)

これを解くプログラムを考えてみよう．まず，微分をどう表現するかであるが，近似的に，とても小さい定数 dx に対して

$$g'(x) = \frac{g(x + dx) - g(x)}{dx}$$

としよう．微分を求める関数は，自然に次のような高階関数として定義できる．

```
# let deriv f =
#   let dx = 0.1e-10 in
#     fun x -> (f(x +. dx) -. f(x)) /. dx;;
val deriv : (float -> float) -> float -> float = <fun>
```

例えば $f(x) = x^3$ の 3 における微分係数は，

```
# deriv (fun x -> x *. x *. x) 3.0;;
- : float = 26.999913416148047
```

と計算される．

次に不動点を求める関数を定義してみよう．この関数は，関数 f と初期値 x から，$f(x), f(f(x)), \ldots$ を計算していき，$f^n(x) = f^{n-1}(x)$ となったときの $f^n(x)$ を返す．実数の計算には誤差が伴うので実際には，ある時点で $|f^{n-1}(x) - f^n(x)|$ がある閾値以下になったときに終了とする．

```
# let fixpoint f init =
#   (* computes a fixed-point of f, i.e., r such that f(r)=r *)
#   let threshold = 0.1e-10 in
#   let rec loop x =
#     let next = f x in
#     if abs_float (x -. next) < threshold then x
#     else loop next
#   in loop init;;
val fixpoint : (float -> float) -> float -> float = <fun>
```

さて，これを使って，Newton-Raphson 法で用いる不動点を求めるべき関数は元の関数から

```
# let newton_transform f = fun x -> x -. f(x) /. (deriv f x);;
val newton_transform : (float -> float) -> float -> float = <fun>
```

で計算できる．

最終的に，Newton-Raphson 法で $f(x) = 0$ の解を求める関数は，関数 f と初期値として用いる $guess$ を受け取って

```
# let newton_method f guess = fixpoint (newton_transform f) guess;;
val newton_method : (float -> float) -> float -> float = <fun>
```

と定義できる.

```
# let square_root x = newton_method (fun y -> y *. y -. x) 1.0;;
val square_root : float -> float = <fun>
# square_root 5.0;;
- : float = 2.23606797750364272
```

4.1.5 練習問題

Exercise 4.1 実数上の関数 f に対して $\int_a^b f(x)dx$ を計算する関数 `integral f a b` を定義せよ. またこれを使って, $\int_0^\pi \sin x dx$ を計算せよ.

近似的な計算方法として, ここでは台形近似を説明するが他の方法でも良い. 台形公式では $b-a$ を n 分割した区間の長さを δ として, 台形の集まりとして 計算する. i 番目の区間の台形の面積は

$$\frac{(f(a+(i-1)\delta)+f(a+i\delta))\cdot\delta}{2}$$

として求められる.

Exercise 4.2 練習問題 3.7 の `pow` 関数をカリー化して定義せよ. 次に第一引数が指数になるよう (`pow n x`) に定義し, 3 乗する関数 `cube` を部分適用で定義せよ. 指数が第二引数であるように定義されている場合 (`pow x n`), `cube` を `pow` から定義するには どうすればよいか?

Exercise 4.3 以下の 3 つの型

- `int -> int -> int -> int`
- `(int -> int) -> int -> int`
- `(int -> int -> int) -> int`

の違いを説明せよ. また, 各型に属する適当な関数を定義せよ.

4.2 多相性

いろいろな関数を書いていくと, 引数の型に関わらず同じことをする関数が出現する場合がしばしば現れる. 例えば, 二つ組から 第一要素を取出す関数を考えてみよう. 例えば, `int * int` に対するこのような関数は,

```
# let fstint ((x, y) : int * int) = x;;
val fstint : int * int -> int = <fun>
```

と書ける. 明示的に型を宣言しているのは, 意図的である. また, `(int * float) * string` のような組と文字列の組に対して同様な関数を定義すると

```
# let fst_ifs ((x, y) : (int * float) * string) = x;;
val fst_ifs : (int * float) * string -> int * float = <fun>
```

と書ける．さて，ここまでくると生じる疑問は，組の要素の組み合わせごとにいちいち別の第一要素を取り出す関数をかかなければいけないのだろうか?ということである．これらの関数は引数の型を除いて同じ格好をしている．それならば，共通部分はひとつの定義におさめ，差異をパラメータ化できないだろうか? パラメータ化するとしたら，パラメータは一体なんであろうか?

注意深く考えると，この共通の定義は引数の「型」，より正確には第一要素と第二要素の型 (`int` と `int`，または `int * float` と `string`) に関してパラメータ化することになることがわかるだろう．また，このパラメータとしては今までの関数とは異なり,「型を表現する変数」のようなものが必要なことがわかる．これを明示的に書き表したのが下の関数宣言である．

```
# let fst ((x, y) : 'a * 'b) = x;;
val fst : 'a * 'b -> 'a = <fun>
```

`'a` と `'b` が**型変数**(*type variable*) と呼ばれるものである．このような「型に関して抽象化された関数」を**多相的関数**(*polymorphic function*) と呼ぶ．この `fst` の型 `'a * 'b -> 'a` は「任意の型 `T1`, `T2` に対して，`T1 * T2 -> T1`」と読むことができる．(論理記号を使うならば ∀'a.∀'b.(`'a * 'b -> 'a`) と考えるのがより正確である．) 実は，Objective Caml の型推論機能はこのような多相的関数の宣言に際しても，明示的に型変数を導入する必要はない．

```
# let fst (x, y) = x;;
val fst : 'a * 'b -> 'a = <fun>
```

通常の「式に関して抽象化された関数」を適用する際に，実引数，つまりパラメータの実際の値を渡すのと同様，多相的関数には「型の実引数」を渡すと考えられるが，実際のプログラムでは，型引数を明示的に書き下す必要はない．(書き下すための文法は用意されていない．)

```
# fst (2, 3);;
- : int = 2
# fst ((4, 5.2), "foo");;
- : int * float = (4, 5.2)
```

概念的には，最初の例では `'a`, `'b` に `int` が，次の例では `'a` に `int * float`，`'b` に `string` が渡っていると考えることができる．別の見方をすると，`fst` という式が，二つの違った型の式 `int * int -> int` と `(int * float) * string -> int * float` として使われていると見ることができる．このように一つの式が複数の型を持つことを言語に**多相性**(*polymorphism*) がある，という．また，特に，関数の型情報の一部をパラメータ化することによって発生する多相性を**パラメータ多相**(*parametric polymorphism*) と呼ぶ．パラメータ多相の関数は型変数に何が渡されようともその振舞いは同じであるという特徴がある．多相性は，ひとつの定義の (再) 利用性を高めるのに非常に役立つ．

いくつか多相性のある関数の例をみてみよう．以下の `id` は**恒等関数**(*identity function*) とよばれ，与えられた引数をそのまま返す関数である．また，関数適用関数 `apply` は関数とその引数を引数と受け取って，関数適用を行う．

```
# let id x = x;;
val id : 'a -> 'a = <fun>
# let apply f x = f x;;
val apply : ('a -> 'b) -> 'a -> 'b = <fun>
```

`apply` の型では，(`'a -> 'b`) が `f` の型を，`'a` が `x` の型を示す．型変数 `'a` が，ふたつの引数 `f` と `x` の間の制約，つまり `f` は `x` に適用できなければならないことを表現していることに注意したい．

```
# abs;;
- : int -> int = <fun>
# apply abs (-5);;
- : int = 5
# apply abs "baz";;
  apply abs "baz";;
            ^^^^^
This expression has type string but is here used with type int
```

関数の多相性はどこに由来するものであろうか? fst の定義をよく見ると，本質的に，(x, y) というパターンの要請する引数に関する制約は,「二つ組であること」だけで，各要素が整数であろうが，文字列であろうが，構わないはずである．また， 各要素 x, y に対して何の操作も行われていないため，それが唯一の制約である．また，apply の定義からは f に対する要請は「x に適用できる関数であること」だけである．このように，パラメータ多相は，関数が引数の部分的な構造 (組である，関数であること) のみで計算が行われることに起因している．また型変数は，関数自身は操作しない部分の構造を抽象化していると考えることができる．つまり 'a * 'b -> 'a という型を見れば，その関数が引数の第一要素も第二要素も使わないことがわかるのである．

> **様々な多相性** その他の多相性としては，多くの言語に見られる + という一つの記号が整数同士もしくは実数同士の足し算どちらでも使える，といった**アドホックな多相性**(*ad-hoc polymorphism*) と呼ばれるもの，オブジェクト指向言語で見られる親子クラス関係など，型上に定義された二項関係によって，式が複数の型を持ちうる**部分型多相**(*subtyping polymorphism*) といったものがある．

4.2.1 let 多相と値多相

Objective Caml では多相性を持てるのは，let(宣言もしくは式) で導入された変数のみである．関数のパラメータを多相的に使うことはできない．具体的には，下の例で x を (id が来るとわかっていても) 異なる型の値への適用は許されない．

```
# (fun x -> (x 1, x 2.0)) id;;
  (fun x -> (x 1, x 2.0)) id;;
                    ^^^
This expression has type float but is here used with type int
```

このエラーメッセージは，x 1 を型推論した時点で x の引数の型は int として決定されてしまったのに，float に対して適用している，という意味である．

また，任意の let 宣言/式でよいわけではなく，定義されるもの (右辺) が「値」でなければいけない，という制限[3]がある．値として扱われるものは，関数の宣言，定数，など計算を必要としない式である．逆に許されないものは，関数適用などの，値に評価されるまでに計算を伴う式である．以下の例は f を x に二度適用する double 関数である．

```
# let double f x = f (f x);;
val double : ('a -> 'a) -> 'a -> 'a = <fun>
# double (fun x -> x + 1) 3;;
- : int = 5
# double (fun s -> "<" ^ s ^ ">") "abc";;
- : string = "<<abc>>"
```

[3]実は，現在演習で使用しているバージョン 3.08.1 では，もう少し制限が緩くなっていて，値とある種の型を持つ式に多相性が許されるようになっているが，ここでは細かくは触れない．

さらに，この関数を組み合わせると，同じ関数を 4 度適用する関数として用いることができる．

```
# double double (fun x -> x + 1) 3;;
- : int = 7
# double double (fun s -> "<" ^ s ^ ">") "abc";;
- : string = "<<<<abc>>>>"
```

ところがこの double double を let でそのまま宣言しても，右辺が関数適用であり，値ではないので，多相的につかうことはできない．

```
# let fourtimes = double double in
# (fourtimes (fun x -> x+1) 3,
#  fourtimes (fun s -> "<" ^ s ^ ">") "abc");;
   fourtimes (fun s -> "<" ^ s ^ ">") "abc");;
                              ^
This expression has type int but is here used with type string
```

これを let 式ではなくて let 宣言した場合も同様な制限が課せられる．

```
# let fourtimes = double double;;
val fourtimes : ('_a -> '_a) -> '_a -> '_a = <fun>
```

アンダースコアのついた型変数 '_a は,「一度だけ」任意の型に置換できる型変数であり，一度決まってしまうと二度と別の型として使うことはできない．このため，多相的に fourtimes を使用することはできない．

```
# fourtimes (fun x -> x + 1) 3;;
- : int = 7
# fourtimes;;
- : (int -> int) -> int -> int = <fun>
# fourtimes (fun s -> "<" ^ s ^ ">") "abc";;
  fourtimes (fun s -> "<" ^ s ^ ">") "abc";;
                              ^
This expression has type int but is here used with type string
```

fourtimes の型変数が一度目の適用で int に固定されてしまっている．

この制限を**値多相**(*value polymorphism*) と呼ぶ．値多相に制限しなければならない理由は副作用 (7 章参照) をもつ言語機構と深く関係があるのだが，ここでは説明しない．fourtimes のような定義に多相性を持たせるためには，

```
# let fourtimes' f = double double f
#   (* equivalent to "let fourtimes' = fun f -> double double f" *) ;;
val fourtimes' : ('a -> 'a) -> 'a -> 'a = <fun>
```

のように，明示的にパラメータを導入して，関数式として (つまり fun を使って) 定義してやればよい．

```
# fourtimes' (fun x -> x + 1) 3;;
- : int = 7
# fourtimes' (fun s -> "<" ^ s ^ ">") "abc";;
- : string = "<<<<abc>>>>"
```

このように，関数を計算する式 (ここでの double double) に fun x -> ... x をつけて値とするような (プログラムの) 変換を η**-展開**(η-*expansion*) という．

4.2.2 多相型と型推論

さて，これまで Objective Caml では型推論の機能があり，プログラマは関数の引数の型を特に明示的に宣言しなくてもよいことには触れたが，それがどのような仕組みで実現されているかについては詳しく述べなかった．ここでは，簡単に型推論の仕組みを見る．

まず簡単な例から考えてみよう．`fun x -> x + 1` という式はもちろん `int -> int` 型を持つわけだがこれがなぜか考えてみると，ひとつの考え方として

- `+` は `int -> int -> int` 型だから両辺に `int` が来なければならない．
- `x` の型は与えられていないが，`+` の左側にあるから，`x` は `int` でなければならないはずだ．`1` も同様に `int` でなければならないが，これは OK だ．
- `x` が `int` であると仮定すれば，`x + 1` は `int` になる．
- `x + 1` を `int` でなければならない `x` で抽象化するから，全体の型は `int -> int` である．

という，推論が成り立つ．Objective Caml はまさにこのように型推論を行っている．より具体的には，式をボトムアップに見ていく．定数 `1` などはその型が自明に与えられ，変数に関してはとりあえず，未知の型を表す型変数を割り当てる．関数適用や `if` 式など，部分式から構成される個所で，部分式の型に関する制約 (「`x` の型 `'a` は `int` でなければならない」など) を構成し，それを解く．もしも制約が解けない場合は，その式は型があっていないと結論づけられる．制約が解けない例は

```
fun x -> if x then 1 else x
```

を考えるとわかる．`if` 式の型規則は `if` 直後の式は `bool` に等しく，`then` 節，`else` 節の型も等しい，というものであるから，`x` に割り当てられた型変数を `'a` とすると，`int = 'a = bool` という制約が得られる．これは明らかに解くことができないので，この式は型エラーとなる．

制約を解いても型変数が残る場合がある．この式が値で `let` で宣言されるものであれば，この型変数は型パラメータとして (暗黙の内に) 抽象化される．例えば，先ほどの `apply` の宣言は，`f`, `x` の未知の型をそれぞれ `'a`, `'b` とすると `f x` という式から，`f x` の型を `'c` として，`'a = 'b -> 'c` という制約と `fun f x -> f x` 全体の型 `'a -> 'b -> 'c` が導かれる．これを書き換えて，`('b -> 'c) -> 'b -> 'c` となる．

ひとつの式には (型エラーを起こさない範囲で) 様々な型が割当てられる可能性があるわけだが，それらの候補のうちには「一般性・汎用性」の高いものと低いものがある．例えば，`fst` 関数に対しては，`int * int -> int`，`float * int -> float`，`'a * 'a -> 'a` といった型が与えられるが，これらは `'a * 'b -> 'a` よりも (適用できる場所が少ないという意味で) 一般性の低い型である．つまり `'a * 'a -> 'a` という型は引数の組の要素の型が等しいような引数にしか適用できないという，本来なら不必要な制限がついている分，`'a * 'a -> 'a` という型よりも汎用性が低いと考えられる．与えられた式の最も一般的な型 (が存在する場合，その型) のことを**主要な型**(*principal type*) と呼ぶ．Objective Caml (や Standard ML, Haskell) では，与えられた式には常に主要な型が存在することが保証されており，また，型推論アルゴリズムは，上で述べたような方法で，主要な型を**常に**求めることができるという，良い性質を持っている．

> **型推論の長所・短所** ところで，型推論は変数の型宣言を省ける一方，型宣言そのものはプログラムを読む上でも有用なものである．どのような場合に宣言をすべき/しないべきかは，趣味の問題である部分もあるが，高階関数/匿名関数を使うようになると，型が文脈から自明な場合が多く現れる．たとえば，先ほどの Newton-Raphson 法で，

```
let square_root x = newton_method (fun y -> y *. y -. x) 1.0;;
```

のように高階関数 newton_method に，匿名関数を渡している．newton_method の型から匿名関数のパラメータ y の型が float であることはすぐわかる．ここで，わざわざ fun (y : float) -> ... と書かなければならないとしたら結構面倒である．

また，主要な型を求める一番確実な方法は，型宣言をしないことである．先ほどの fst を

```
# let fst ((x, y) : 'a * 'a) = x;;
val fst : 'a * 'a -> 'a = <fun>
```

と宣言しても，型チェックを通過するが，組の要素が同じ型を持たなくてはならないという，一般性を欠く定義になってしまう．

```
# fst (true, 3.2);;
  fst (true, 3.2);;
      ^^^^^^^^^^^
This expression has type bool * float but is here used with type bool * bool
```

この反面，ML 型推論は，型検査によるエラーが思わぬところで報告される，という欠点がある．例えば，前の例の変形だが，

```
# fun x -> (x 1, x true);;
  fun x -> (x 1, x true);;
                  ^^^^
This expression has type bool but is here used with type int
```

で，x true でエラーが報告されているが，もしかすると，x 1 で 1 を渡している部分で間違えたのかもしれない．特に定義が大きくなり x 1 と x true が離れているような場合，「本当に間違えたところ」を一目で探すのが難しい場合がある．これは，しばしば ML の型推論の欠点として指摘される点である．(熟練した Objective Caml プログラマは，型推論がどのようにして行なわれるかを理解しているため，それほど惑わされずに本当のエラー箇所を探しあてることができる．) これに対し，x の型を宣言し，

```
# fun (x: bool -> 'a) -> (x 1, x true);;
  fun (x: bool -> 'a) -> (x 1, x true);;
                          ^
This expression has type int but is here used with type bool
```

とすれば，x の取る引数の型が予めわかっているので，x 1 の部分でエラーが報告される．

4.2.3 Case Study: コンビネータ

「計算」という概念のモデルとして基本的なものにチューリング機械がある．チューリング機械は，CPU がメモリの読み書きをしながら計算を進めて行く様子を単純化したものである．これに対して，関数型プログラミングの計算をモデル化したものに λ 計算，コンビネータ理論といったものがある．λ 計算の体系は関数適用におけるパラメータの引数の置換をモデル化したもので，複雑な計算も関数抽象式 (Objective Caml の fun 式) と関数適用の組み合わせのみで表すことができる．コンビネータ理論は，λ 計算の理論と密接に関わっていて，コンビネータと呼ばれるいくつかの「関数を組み合わせて新しい関数を構成する」高階関数の組合わせで，複雑な計算を表現するものである．ここでは簡単にコンビネータについてみていく．

最もよく知られたコンビネータのひとつは数学で使う関数合成 $f \circ g$ (但し，$(f \circ g)(x) = f(g(x))$ とする) を表す $\circ$ である．これは，関数 f, g からそれらを合成した関数を構成するコンビネータと考えられる．$\circ$ を中置演算子 $ で表現することにすると，

```
let ($) f g x = f (g x);;
val ( $ ) : ('a -> 'b) ->  ('c -> 'a) -> 'c -> 'b = <fun>
```

と表現される．例えば，二つの関数の組合わせからなる関数

```
fun x -> ~-. (sqrt x)
```

は，

```
# let f = ( ~-. ) $ sqrt;;
val f : float -> float = <fun>
# f 2.0;;
- : float = -1.41421356237309515
```

という式で表現できる．(`~-.` は実数値の符号を逆転させる単項演算子である．) コンビネータのポイントは，明示的にパラメータを導入することなく，単純な関数を組み合わせてより複雑な関数を構成できるところにある．

もっとも単純なコンビネータは先に見た `id` である．(コンビネータ理論では I コンビネータと呼ばれる．) `id f` は `f` と同じ関数を表現する．また I コンビネータは関数合成と組み合わせても何も起らない．

```
# (sqrt $ id) 3.0
#    (* Without (), it would be equivalent to sqrt $ (id 3.0) *)
# ;;
- : float = 1.73205080756887719
# (id $ sqrt) 3.0;;
- : float = 1.73205080756887719
```

K コンビネータは定数関数を構成するためのコンビネータであり，以下の関数で表現される．

```
# let k x y = x;;
val k : 'a -> 'b -> 'a = <fun>
```

`k x` は何に適用しても `x` を返す関数になる．

```
# let const17 = k 17 in const17 4.0;;
- : int = 17
```

次の S コンビネータは関数合成の $\circ$ を一般化したものである．

```
# let s x y z = x z (y z);;
val s : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c = <fun>
```

Objective Caml で `fun` 式と関数適用の組合わせ「のみ」で表現できる関数 (`fun x -> x`, `fun x y -> x (x y)` など) は S と K の組合わせのみで表現できることが知られている．これらは，単純型付き λ 計算で表現できる関数のクラスと同じである．例えば I コンビネータは $S\ K\ K$ として表せる．

```
# s k k 5;;
- : int = 5
```

(不動点コンビネータを導入した) コンビネータ，(型なしの) λ 計算はチューリング機械と同程度の表現力がある計算モデルであることが知られている．

4.2.4　練習問題

Exercise 4.4 以下の関数 curry は，与えられた関数をカリー化する関数である．

```
# let curry f x y = f (x, y);;
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
# let average (x, y) = (x +. y) /. 2.0;;
val average : float * float -> float = <fun>
# let curried_avg = curry average;;
val curried_avg : float -> float -> float = <fun>
# average (4.0, 5.3);;
- : float = 4.65
# curried_avg 4.0 5.3;;
- : float = 4.65
```

この逆，つまり (2 引数の) カリー化関数を受け取り，二つ組を受け取る関数に変換する uncurry 関数を定義せよ．

```
# let avg = uncurry curried_avg in
#   avg (4.0, 5.3);;
- : float = 4.65
```

Exercise 4.5 以下の関数 repeat は double, fourtimes などを一般化したもので，f を n 回，x に適用する関数である．

```
# let rec repeat f n x =
#   if n > 0 then repeat f (n - 1) (f x) else x;;
val repeat : ('a -> 'a) -> int -> 'a -> 'a = <fun>
```

これを使って，フィボナッチ数を計算する関数 fib を定義する．以下の ... の部分を埋めよ．

```
let fib n =
  let (fibn, _) = ...
  in fibn;;
```

Exercise 4.6 次の関数 funny がどのような働きをするか説明せよ．

```
# let rec funny f n =
#   if n = 0 then id
#   else if n mod 2 = 0 then funny (f $ f) (n / 2)
#   else funny (f $ f) (n / 2) $ f;;
val funny : ('a -> 'a) -> int -> 'a -> 'a = <fun>
```

Exercise 4.7 s k k が恒等関数として働く理由を s k k 1 が評価される計算ステップを示すことで，説明せよ．

また，fun x y -> y と同様に働く関数を，コンビネータ s と k を関数適用のみで (fun や let による関数定義を使わずに) 組み合わせた形で表現せよ．

```
# ( (* s, k を 関数適用で組み合わせた式 *) ) 1 2;;
- : int = 2
```

Exercise 4.8 double double f x が f (f (f (f x))) として働く理由を前問と同様にして説明せよ．

第5章　再帰的多相的データ構造: リスト

これまで，構造のあるデータを表現するための手段としては組を用いてきた．ここでは**リスト**(*lists*)という,「データの (有限) 列」を表現するデータ構造をみていく．リストは構造自体が再帰的である．また，格納するデータの種類が変わりうるという意味で多相的であるという特徴をもっている．

5.1　リストの構成法

まずは簡単なリストの例を見てみよう．リストはデータ列を構成する要素を ; で区切り，[] で囲むことで構成される．

```
# [3; 9; 0; -10];;
- : int list = [3; 9; 0; -10]
# let week = ["Sun"; "Mon"; "Tue"; "Wed"; "Thu"; "Fri"; "Sat"];;
val week : string list = ["Sun"; "Mon"; "Tue"; "Wed"; "Thu"; "Fri"; "Sat"]
```

リストの式には “⟨ 要素の型 ⟩ list” という型が与えられる．これが意味するのは「(Objective Caml では) 一つのリストに並ぶ要素の型は同じ」でなければならない，ということである．また別の見方をすると，リスト式はその列の長さに関わらず，要素の型が同じであれば，同じリスト型に属するということである．

```
# [1; 'a'];;
  [1; 'a'];;
      ^^^
This expression has type char but is here used with type int
# (* compare with the type of [3; 9; 0; -10] *)
# [-3];;
- : int list = [-3]
```

このことは組とリストの決定的な違いであるので注意すること．組の型は各要素の型を並べることで記述されるため，各要素の型は異ってもよいが，大きさの違う組は同じ型に属し得ない (すなわち，組の大きさの情報が型に現れている).

リストの構造は，組のように「要素を並べたもの」と思う代りに，以下のように再帰的に定義されている構造とも考えられる．つまり

- [] は，**空リスト**(*empty list, null list*) と呼ばれ，リストである．
- リスト l の先頭に (l の要素と同じ類いの) 要素 e を追加したもの (e :: l と書く) もリストである．

と定義することもできる．この定義は二番目の節が再帰的になっている．:: のことを **cons オペレータ**(または単に cons) と呼ぶこともある．このようにリストを構築する例を見てみよう．

```
# let oddnums = [3; 9; 253];;
val oddnums : int list = [3; 9; 253]
# let more_oddnums = 99 :: oddnums;;
val more_oddnums : int list = [99; 3; 9; 253]
# (* :: is right associative, that is, e1 :: e2 :: l = e1 :: (e2 :: l) *)
# let evennums = 4 :: 10 :: [256; 12];;
val evennums : int list = [4; 10; 256; 12]
# [];;
- : 'a list = []
```

要素を列記する方法と :: を用いる方法を混ぜることもできる．evennums の例に見るように，:: は右結合する (e1 :: e2 :: l は e1 :: (e2 :: l) と読む)． 空リスト [] は要素がなく，どのような要素のリストとも見なせること ができるため，'a list という多相型が与えられている．もちろん，空リストには様々な型の要素を追加することができる．

```
# let boollist = true :: false :: [];;
val boollist : bool list = [true; false]
# let campuslist = "Yoshida" :: "Uji" :: "Katsura" :: [];;
val campuslist : string list = ["Yoshida"; "Uji"; "Katsura"]
```

ちなみに Objective Caml では「要素を並べる」定義は，(内部的には) 再帰的な定義の略記法である．つまり，

$$[e_1;\ e_2;\ \cdots\ ;\ e_n] = e_1 :: e_2 :: \cdots :: e_n :: []$$

である．

cons オペレータ :: は，あくまで「一要素の (先頭への) 追加」を行うもので，リストにリストを追加 (連結) するという操作や，リストの最後尾へ要素を追加するといった操作は :: で行えない．

```
# [1; 2] :: [3; 4];;
  [1; 2] :: [3; 4];;
            ^

This expression has type int but is here used with type int list
# [1; 2; 3] :: 4;;
  [1; 2; 3] :: 4;;
               ^

This expression has type int but is here used with type int list list
```

:: は文法キーワードなので，:: の型を見ることはできないが，敢えて型を考えるなら 'a -> 'a list -> 'a list と思うのがよいだろう．リストはどんな値でも要素にでき，関数のリスト，リストのリスト等を考えることも可能である．

```
# [(fun x -> x + 1); (fun x -> x * 2)];;
- : (int -> int) list = [<fun>; <fun>]
# [1; 2; 3] :: [[4; 5]; []; [6; 7; 8]];;
- : int list list = [[1; 2; 3]; [4; 5]; []; [6; 7; 8]]
```

2 番目の例は，整数リストのリストに整数リストを :: で追加している．

5.2 リストの要素へのアクセス: match 式とリストパターン

さて，リストの要素にアクセスするときには組と同様にパターンを用いる．リストパターンは

[⟨ パターン$_1$⟩; ⟨ パターン$_2$⟩; ...; ⟨ パターン$_n$⟩]

で n 要素のリスト (n が 0 なら空リスト) にマッチさせることができる．また，:: を使ってパターンを記述することもできる．

⟨ パターン$_1$⟩ :: ⟨ パターン$_2$⟩

と書いて，先頭の要素を ⟨ パターン$_1$⟩ に，**残りのリストを**⟨ パターン$_2$⟩ にマッチさせることができる．次の関数は，三要素の整数リストの要素の和を計算する関数である．

```
# (* equivalent to
#    let sum_list3 (x :: y :: z :: []) = x + y + z *)
# let sum_list3 ([x; y; z]) = x + y + z;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
  let sum_list3 ([x; y; z]) = x + y + z;;
                ^^^^^^^^^^^^^^^^^^^^^^^
val sum_list3 : int list -> int = <fun>
```

リスト式の場合と同様，[x; y; z] というパターンは :: と [] を使ったパターンで表すことができる．ここで，重要なことはこの関数をコンパイルをするとコンパイラから警告が発せられていることである．この “nonexhaustive pattern” と呼ばれる警告は,「パターンの表す型に属する値でありながら，パターンにマッチしない値が存在する」という意味であり，コンパイラはマッチしない値の例を示してくれる．たしかにこの例では引数の型は int list であるにも関わらず三要素のリストにしかマッチしない．実際，マッチしない値に関数を適用するとマッチングに失敗したという例外が発生する．

```
# sum_list3 [4; 5; 6];;
- : int = 15
# sum_list3 [2; 4];;
Exception: Match_failure ("", 27, 14).
```

では，任意の長さの整数リストの要素の和を取る関数を書くにはどうすればよいのだろうか? 例え，二要素のリストの和を取る関数，四要素リストの和を取る関数などなどを定義し，それを何らかの手段で組み合わせることができたとしても，それでは不十分である．関数定義の大きさが無限に長くなってしまう! ここで，リストが再帰的な定義をされた構造であることが非常に重要な意味を持ってくる．つまり，リストの再帰的な定義から，要素の和を取る定義を導くことができるのである．それが以下の定義である．

- 空リストの全要素の和は 0 である．
- 先頭要素が n で，残りのリスト l に対する全要素の和を s とすると，n :: l の全要素の和は $n + s$ である．

ポイントは，長いリストの全要素の和はより短いリストの全要素の和から計算できることである．(fact などの定義と比べてみよ．) これを Objective Caml の定義とするためには，与えられたリストが空かそうでないかを判断する手段が必要であるが，ひとまず例を見て，その判断をどのように行うか見てみよう．

```
# let rec sum_list l =
#   match l with
#     [] -> 0
#   | n :: rest -> n + (sum_list rest);;
val sum_list : int list -> int = <fun>
```

まず，sum_list は再帰関数になっている． match 以下が l が空リストかそうでないかを判断し，分岐処理を行っている部分で match 式と呼ばれる．match 式の一般的な文法は，

match ⟨式$_0$⟩ with ⟨パターン$_1$⟩ -> ⟨式$_1$⟩ | ... | ⟨パターン$_n$⟩ -> ⟨式$_n$⟩

という形で与えられ，⟨式$_0$⟩ を評価した結果を，⟨パターン$_1$⟩ から**順番に**マッチさせていき，⟨パターン$_i$⟩ でマッチが成功したら，⟨式$_i$⟩ を評価する．全体の値は ⟨式$_i$⟩ の値である．各パターンで導入された変数 (上の例では n, rest) は対応する式の中だけで有効である．上の Objective Caml による定義が，自然言語による定義に対応していることを確かめよ．多くのリストを操作する関数は sum_list のように，空リストの場合の処理と，cons の場合の処理を match で組み合わせて書かれる．

match 式についての注意　match 式がマッチングを順番に行う，というのは非常に重要な点である．もしも，同じ値が複数のパターンにマッチする場合は先に書いてあるパターンにマッチしてしまう．このような例は特に**定数パターン**を使用すると発生しやすい．定数パターンは整数，文字列定数をパターンとして用いるもので，その定数にのみマッチする．また，前に書いてあるパターンのせいで，パターンにマッチする値がない場合，コンパイラは “this match case is unused.” と警告を発する．

```
# let f x =
#   match x with (1, _) -> 2 | (_, 1) -> 3 | (1, 1) -> 0 | _ -> 1;;
Warning: this match case is unused.
    match x with (1, _) -> 2 | (_, 1) -> 3 | (1, 1) -> 0 | _ -> 1;;
                                             ^^^^^^
val f : int * int -> int = <fun>
# f (1, 1);;
- : int = 2
```

(1, 1) は最初のパターンにマッチする．また，3 番目のパターンは決して使われないので警告が出ている．

さて，一般的なリスト操作関数の例を見ていく前に，もうひとつ例をみていこう．(空でない) 整数リストのなかから最大値を返す関数 max_list は

```
# let rec max_list l =
#   match l with
#     [x] -> x
#   | n1 :: n2 :: rest ->
#     if n1 > n2 then max_list (n1 :: rest) else max_list (n2 :: rest);;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
```

```
[]
  ..match l with
      [x] -> x
    | n1 :: n2 :: rest ->
       if n1 > n2 then max_list (n1 :: rest) else max_list (n2 :: rest)..
val max_list : 'a list -> 'a = <fun>
```

のように定義される．

5.3 リスト操作の関数

さて，リストに対する基本的な操作 (構成と要素アクセス) をみたところで，様々なリスト操作を行う関数を見ていこう．多くの関数がリストの構造に関して再帰的に定義される．また，ほとんどすべての関数が要素型によらない定義をしているため，多相型が与えられることに注意しよう．

hd, tl, null リストの先頭要素，リストの先頭を除いた残りを返す関数 hd (head の略), tl (tail の略) は以下のように定義され，

```
# let hd (x::rest) = x
# let tl (x::rest) = rest;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
Characters 29-45:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
  let hd (x::rest) = x
         ^^^^^^^^^^^^^
  let tl (x::rest) = rest;;
         ^^^^^^^^^^^^^^^^
val hd : 'a list -> 'a = <fun>
val tl : 'a list -> 'a list = <fun>
```

空リストに対しては働けない nonexhaustive な関数である．null は与えられたリストが空かどうかを判定する関数である．

```
# let null = function [] -> true | _ -> false;;
val null : 'a list -> bool = <fun>
```

function キーワードは fun と match を組み合わせて匿名関数を定義するもので，

function ⟨パターン$_1$⟩ -> ⟨式$_1$⟩ | ... | ⟨パターン$_n$⟩ -> ⟨式$_n$⟩

で

fun x -> match x with ⟨パターン$_1$⟩ -> ⟨式$_1$⟩ | ... | ⟨パターン$_n$⟩ -> ⟨式$_n$⟩

を示す．最後に受け取った引数に関して即座にパターンマッチを行うような関数定義の際に便利である．

nth, take, drop　次は，n 番目の要素を取り出す nth, n 番目までの要素の部分リストを取り出す take，n 番目までの要素を抜かした部分リストを取り出す drop である．リストの要素は先頭を一番目とする．

```
# let rec nth n l =
#   if n = 1 then hd l else nth (n - 1) (tl l)
# let rec take n l =
#   if n = 0 then [] else (hd l) :: (take (n - 1) (tl l))
# let rec drop n l =
#   if n = 0 then l else drop (n - 1) (tl l);;
val nth : int -> 'a list -> 'a = <fun>
val take : int -> 'a list -> 'a list = <fun>
val drop : int -> 'a list -> 'a list = <fun>
```

これらの関数はリストの構造でなく，n に関しての再帰関数になっている．

```
# let ten_to_one = [10; 9; 8; 7; 6; 5; 4; 3; 2; 1; 0];;
val ten_to_one : int list = [10; 9; 8; 7; 6; 5; 4; 3; 2; 1; 0]
# nth 4 ten_to_one;;
- : int = 7
# take 8 ten_to_one;;
- : int list = [10; 9; 8; 7; 6; 5; 4; 3]
# drop 7 ten_to_one;;
- : int list = [3; 2; 1; 0]
# take 19 ten_to_one;;
Exception: Match_failure ("", 48, 7).
```

length　次はリストの長さを返す関数 length である．

```
# let rec length = function
#     [] -> 0
#   | _ :: rest -> 1 + length rest;;
val length : 'a list -> int = <fun>
```

length の型は _ パターンを使って先頭要素を使用してないことからわかるように，どんなリストに対しても使うことができる．実際，型をみると入力の要素型が型変数になっている．

```
# length [1; 2; 3];;
- : int = 3
# length [[true; false]; [false; false; false;]];;
- : int = 2
```

また length はふたつめの結果にみられるように，一番外側のリストの長さを計算するものである (結果は 5 ではない)．

append, reverse　次に示す append 関数はリスト同士を連結する関数である．append l1 l2 の再帰的定義は

- 空リストに l_2 を連結したものは l_2 である．
- 先頭が v で，残りが l_1' であるリストに l を連結したものは，l_1' と l の連結の先頭に v を追加したものである．

と考えることができる．

```
# let rec append l1 l2 =
#   match l1 with
#     [] -> l2
#   | x :: rest -> x :: (append rest l2);;
val append : 'a list -> 'a list -> 'a list = <fun>
# append [1; 2; 3] [4; 5; 6];;
- : int list = [1; 2; 3; 4; 5; 6]
```

この append の定義は，l1 の長さが長くなるほど再帰呼び出しが深く行われ，l2 の長さには関係がない．ちなみに append 関数は，もともと Objective Caml 起動時に中置オペレータ @ として定義されている．

```
# [1; 2; 3] @ [4; 5; 6];;
- : int list = [1; 2; 3; 4; 5; 6]
```

また append を使ってリストを反転させる reverse 関数を定義できる．

```
# let rec reverse = function
#     [] -> []
#   | x :: rest -> append (reverse rest) [x];;
val reverse : 'a list -> 'a list = <fun>
```

リストの最後に要素を追加することは直接はできないので，一要素リストを作って append している．しかし，この関数はあまり効率的ではない．なぜなら，reverse の呼び出し一度につき，append が一度呼ばれるが，この時 append の第一引数の長さは「反転させようとする引数の長さ -1」であり append を計算するのにその長さ分の計算量を必要とする．reverse の再帰呼び出し回数は与えたリストの長さなので，リストの長さの自乗に比例した計算時間がかかってしまう．

これを改善したのが次の定義である．

```
# let rec revAppend l1 l2 =
#   match l1 with
#     [] -> l2
#   | x :: rest -> revAppend rest (x :: l2)
# let rev x = revAppend x [];;
val revAppend : 'a list -> 'a list -> 'a list = <fun>
val rev : 'a list -> 'a list = <fun>
```

最初の再帰関数 revAppend が第一引数を先頭から順に l2 に追加して行く関数である．先頭から追加していくため，l1 の順が逆になって l2 に連結される．

```
# revAppend [1; 2; 3] [4; 5; 6];;
- : int list = [3; 2; 1; 4; 5; 6]
```

この関数も append と同じく，第一引数の長さだけに比例した時間がかかる．リストの反転は revAppend の第二引数が空である特別な場合である．

```
# rev ['a'; 'b'; 'c'; 'd'];;
- : char list = ['d'; 'c'; 'b'; 'a']
```

map　map はリストの各要素に対して同じ関数を適用した結果のリストを求めるための高階関数である.

```
# let rec map f = function
#     [] -> []
#   | x :: rest -> f x :: map f rest;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

たとえば, 整数リストの各要素を 2 倍する式は map を使って,

```
# map (fun x -> x * 2) [4; 91; 0; -34];;
- : int list = [8; 182; 0; -68]
```

と書くことができる. map の型は今まで見た中でかなり複雑である. まず, 'a -> 'b で「何らかの関数」が第一引数であることがわかる. カリー化関数とみるならば, 第二引数は「何らかの関数」の定義域の値を要素とするリストで, 結果が「何らかの関数」の値域の値を要素とするリストとなる. または「何らかの関数」を与えた時点でリストからリストへの関数が返ってきていると解釈してもよい.

forall, exists　forall はリストの要素に関する述語 (要素から bool への関数) と, リストをとり, 全要素が述語を満たすかどうかを, exists は同様に述語とリストをとって, 述語を満たす要素があるかどうかを返す関数である.

```
# let rec forall p = function
#     [] -> true
#   | x :: rest -> if p x then forall p rest else false
# let rec exists p = function
#     [] -> false
#   | x :: rest -> (p x) or (exists p rest);;
val forall : ('a -> bool) -> 'a list -> bool = <fun>
val exists : ('a -> bool) -> 'a list -> bool = <fun>
# forall (fun c -> 'z' > c) ['A'; ' '; '+'];;
- : bool = true
# exists (fun x -> (x mod 7) = 0) [23; -98; 19; 53];;
- : bool = true
```

畳み込み関数 fold　上で見た sum_list, append はリストの要素すべてを用いた演算をするものである. 実はこの二つの関数は共通の計算の構造を持っている. sum_list は [i1; i2; ...; in], つまり

```
i1 :: i2 :: ... :: in :: []
```

から

```
i1 + (i2 + (... + (in + 0)...))
```

を計算し, append [e1; e2; ...; en] l2 は

```
e1 :: (e2 :: ... :: (en :: l2)...)
```

を計算している. このふたつの計算の共通点は,

- 引数リストの cons を, sum_list では + で, append では :: 自身で置換え,

- 末尾の空リストも，sum_list では 0 で，append では l2 で置換え，
- 右から演算を順次行っていく

ことである．このような「右から畳み込む」計算構造を一般化した高階関数を fold_right と呼ぶ．逆に左から畳み込むのを fold_left と呼ぶ．rev は fold_left の例である．何故なら，rev [e1; e2; ...; en] は l ::: x を x :: l として，

```
(...(([] ::: e1) ::: e2) ... ::: en)
```

と表現できるからである．

以下が fold_right, fold_left の定義である．

```
fold_right f [e1; e2; ...; en] e ⟹ f e1 (f e2 (... (f en e)...))

fold_left f e [e1; e2; ...; en] ⟹ f (... (f (f e e1) e2) ...) en
```

を計算する．

```
# let rec fold_right f l e =
#   match l with
#     [] -> e
#   | x :: rest -> f x (fold_right f rest e)
# let rec fold_left f e l =
#   match l with
#     [] -> e
#   | x :: rest -> fold_left f (f e x) rest;;
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
# fold_right (fun x y -> x + y) [3; 5; 7] 0;;
- : int = 15
# fold_left (fun x y -> y :: x) [] [1; 2; 3];;
- : int list = [3; 2; 1]
```

fold_left, fold_right は要素はそのままで cons を適当な演算子に読み替えて，計算をするものと思うことができる．一方，map 関数はリストの構造はそのままで，要素だけを操作するような計算構造を抽象化した高階関数であった．実はリストに関する再帰関数はほとんど map と fold_left または fold_right を組み合わせて定義することができる．例えば，length は全要素をまず map を使って 1 に置換えて，足し算による畳み込みを行えばよいので，

```
# let length l = fold_right (fun x y -> x + y) (map (fun x -> 1) l) 0;;
val length : 'a list -> int = <fun>
```

と定義することも可能である．

連想リスト　辞書における見出しと説明文，データベースにおけるデータ検索のためのキーとデータ，のような「関係」を表現するのに**連想リスト**(*association list*) というデータ構造を用いる．連想リストは構造的にはペアのリストで表現される．各要素 (a, b) はキー a の データ b への関連付けを表現する．

例えば，以下は，都市の名前をキー，市外局番をデータとした連想リストの例である．

```
# let city_phone = [("Kyoto", "075"); ("Osaka", "06"); ("Tokyo", "03")];;
val city_phone : (string * string) list =
  [("Kyoto", "075"); ("Osaka", "06"); ("Tokyo", "03")]
```

このような連想リストとキーから，関連づけられたデータを取り出す関数 assoc は以下のように定義できる．

```
# let rec assoc a = function
#    (a', b) :: rest -> if a = a' then b else assoc a rest;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
  ..................function
     (a', b) :: rest -> if a = a' then b else assoc a rest..
val assoc : 'a -> ('a * 'b) list -> 'b = <fun>
# assoc "Osaka" city_phone;;
- : string = "06"
# assoc "Nara" city_phone;;
Exception: Match_failure ("", 124, 18).
```

連想リストにないキーで問い合わせを行うと，再帰呼び出しがリストの末端まで到達した末にマッチングに失敗して例外を発生する．

5.4 Case Study: ソートアルゴリズム

リストを使ったソートアルゴリズムをいくつかみていこう．以下ではリストを < に関する昇順 (小さい方から順) に並べ替えることを考える．(比較演算子 < は多相的であるため，ソート関数も多相的に使える．)

まずは準備として，ソートの対象として用いる，疑似乱数列を生成するための関数を定義する．

```
# let nextrand seed =
#   let a = 16807.0 and m = 2147483647.0 in
#   let t = a *. seed
#   in t -. m *. floor (t /. m)
# let rec randlist n seed tail =
#   if n = 0 then (seed, tail)
#   else randlist (n - 1) (nextrand seed) (seed::tail);;
val nextrand : float -> float = <fun>
val randlist : int -> float -> float list -> float * float list = <fun>
# randlist 10 1.0 [];;
- : float * float list =
(2007237709.,
 [1458777923.; 1457850878.; 101027544.; 470211272.; 1144108930.; 984943658.;
  1622650073.; 282475249.; 16807.; 1.])
```

挿入ソート(*insertion sort*) は，既にソート済のリストに新しい要素をひとつ付け加える操作を基本として，各要素を順に付け加えていくものである．基本操作 insert は

```
# let rec insert (x : float) = function
#     (* Assume the second argument is already sorted *)
```

```
#     [] -> [x]
#   | (y :: rest) as l -> if x < y then x :: l else y :: (insert x rest);;
val insert : float -> float list -> float list = <fun>
# insert 4.5 [2.2; 9.1];;
- : float list = [2.2; 4.5; 9.1]
```

と書くことができる．パターン中に出現する

〈パターン〉 as 〈変数〉

は as **パターン**と呼ばれるもので，パターンにマッチした値全体を〈変数〉で参照できるものである．ここでは x :: y :: rest と書く代りに x :: l としている．この insert を使ってソートを行う関数は

```
# let rec insertion_sort = function
#     [] -> []
#   | x :: rest -> insert x (insertion_sort rest);;
val insertion_sort : float list -> float list = <fun>
```

と定義できる．

挿入ソートは insert, insertion_sort ともに入力に比例する回数の再帰呼出しを行うため，計算には与えられたリストの長さの自乗に比例した時間がかかる．**クイックソート**(*quick sort*) は C.A.R. Hoare が発明した効率の良いソートアルゴリズムで，**分割統治**(*divide and conquer*) 法に基づき，下のような要領でソートを行う．

- 要素から適当にピボットと呼ばれる値を選びだす．
- 残りの要素をピボットに対する大小でふたつの集合に分割する．
- それぞれの部分集合にたいしてソートを行い，その結果を結合する．

```
# let rec quick = function
#     [] -> []
#   | [x] -> [x]
#   | x :: xs ->  (* x is the pivot *)
#       let rec partition left right = function
#           [] -> (quick left) @ (x :: quick right)
#         | y :: ys -> if x < y then partition left (y :: right) ys
#                      else partition (y :: left) right ys
#       in partition [] [] xs;;
val quick : 'a list -> 'a list = <fun>
```

この quick の定義は append を使用しているのでまだ効率の悪い点が残っている．(append を使用しない定義は練習問題．) クイックソートはリストの長さを n として，平均で $n \log n$ に比例した時間で行えることが知られている．

```
insert_sort (snd (randlist 10000 1.0 []))
```

と

```
quick (snd (randlist 10000 1.0 []))
```

を試してみよ．(snd はペアから第二要素を取り出す定義済の関数である．)

5.5 練習問題

Exercise 5.1 次のうち，正しいリスト表現はどれか．コンパイラに入力する前に，正しい場合と思う場合は式の型を，間違っていると思う場合はなぜ誤りか，を予想してから実際に確認せよ．

1. `[[]]`
2. `[[1; 3]; ["hoge"]]`
3. `[3] :: []`
4. `2 :: [3] :: []`
5. `[] :: []`
6. `[(fun x -> x); (fun b -> not b)]`

Exercise 5.2 `sum_list`，`max_list` を，`match` を使わず `null`, `hd`, `tl` の組合わせのみで定義せよ．`match` を使うテキストの定義と比べ，記述面などの利害得失を議論せよ．

Exercise 5.3 次の関数を定義せよ．

1. 正の整数 n から 0 までの整数の降順リストを生成する関数 `downto0`.
2. 与えられた正の整数のローマ数字表現 (文字列) を求める関数 `roman`．(I = 1, V = 5, X = 10, L = 50, C = 100, D = 500, M = 1000 である．) ただし，`roman` はローマ数字の定義も引数として受け取ることにする．ローマ数字定義は，単位となる数とローマ数字表現の組を大きいものから並べたリストで表現する．例えば

   ```
   roman [(1000, "M"); (500, "D"); (100, "C"); (50, "L");
          (10, "X"); (5, "V"); (1, "I")] 1984
   ⟹ "MDCCCCLXXXIIII"
   ```

   4, 9, 40, 90, 400, 900 などの表現にも注意して，

   ```
   roman [(1000, "M"); (900, "CM"); (500, "D"); (400, "CD");
          (100, "C"); (90, "XC"); (50, "L"); (40, "XL");
          (10, "X"); (9, "IX"); (5, "V"); (4, "IV"); (1, "I")] 1984
   ⟹ "MCMLXXXIV"
   ```

   となるようにせよ．
3. 与えられたリストのリストに対し，内側のリストの要素を並べたリストを返す関数 `concat`.

   ```
   concat [[0; 3; 4]; [2]; [5; 0]; []] = [0; 3; 4; 2; 5; 0]
   ```

4. 二つのリスト `[a1; ...; an]` と `[b1; ...; bn]` を引数として，`[(a1, b1); ...; (an, bn)]` を返す関数 `zip`．(与えられたリストの長さが異なる場合は長いリストの余った部分を捨ててよい．)

5. リストと，リストの要素上の述語 (**bool** 型を返す関数) p をとって，p を満たす全ての要素のリストを返す関数 filter.

```
# let positive x = (x > 0);;
val positive : int -> bool = <fun>
# filter positive [-9; 0; 2; 5; -3];;
- : int list = [2; 5]
# filter (fun l -> length l = 3) [[1; 2; 3]; [4; 5]; [6; 7; 8]; [9]];;
- : int list list = [[1; 2; 3]; [6; 7; 8]]
```

6. リストを集合とみなして，以下の集合演算をする関数を定義せよ.

   (a) belong a s で a が s の要素かどうかを判定する関数 belong.

   (b) intersect s1 s2 で s1 と s2 の共通部分を返す関数 intersect.

   (c) union s1 s2 で s1 と s2 の和を返す関数 union.

   (d) diff s1 s2 で s1 と s2 の差を返す関数 diff.

   但し，集合なので，要素は重複して現れてはならないことに気をつけよ．(関数の引数には重複してないものが与えられるという仮定を置いてもよい．)

Exercise 5.4 f, g を適当な型の関数とする．map f (map g l) を map を一度しか使用しない同じ意味の式に書き換えよ．map (fun x -> ...) l の ... 部分は?

Exercise 5.5 forall, exists を fold_right, map を組み合わせて定義せよ.

Exercise 5.6 quick 関数を @ を使わないように書き換える．quicker は未ソートのリスト l と，sorted というソート済でその要素の最小値が l の要素の最大値より大きいようなリストを受け取る．定義を完成させよ.

```
let rec quicker l sorted = ...
```

Exercise 5.7 与えられた自然数 r に対して $x^2 + y^2 = r$ であるような，(x, y) *(*ただし $x \geq y \geq 0$*)* の組を全てリストとして列挙する関数 squares r を定義せよ．*(*検算用資料*:* $r = 48612265$ の時 *32* 個の解があるそうです．*)*

Exercise 5.8 map の定義は末尾再帰的ではないため，入力リストの長さが長くなるとそれに比例した量のメモリ *(*スタック*)* が必要になる．map と同機能で，必要なメモリ量が定数オーダーである map2 を定義せよ．*(*ヒント*:* 末尾再帰的 *(iterative)* な関数を使う．*)*

第6章　レコード型/ヴァリアント型とその応用

レコード型やヴァリアント型は，組型，リスト型と同様に，そのような名前の一つの型が存在するわけではなく，構造のあるデータを導入するための，似たような構造を持った型の種類である．レコードは組と同様にいくつかの値に名前をつけて並べることで構成されるデータであるのに対し，ヴァリアントは異なる種類の値を同じ型の値として混ぜて扱うためのものである．また，ヴァリアント型はリストなどの再帰的な構造を持つデータ構造を導入するために使われる．どちらも，使用するためにはプログラマが具体的な型の構造を宣言をする必要がある．問題領域に応じて適切なデータ型をデザイン・定義することは (Objective Caml に限らず) プログラミングにおいて非常に重要な技術である．

6.1　レコード型

レコード型の値は，組と同様に，本質的には値をいくつか並べたものである．組との違いは，それぞれの要素に名前を与えて，その名前でレコードの要素にアクセスできることである．この名前と値の組を**フィールド**(*field*)，フィールドの名前をフィールド名，と呼ぶ．

例えば，学生のデータベースを作ることを考えよう．学生一人一人のデータは名前を表す `name` フィールド，学生証番号を表す `id` フィールドの並びとする．新しいレコード型は，`type` 宣言をつかって定義することができる．

```
# type student = {name : string; id : int};;
type student = { name : string; id : int; }
```

`student` が新しい型の識別子である．一般的には，各フィールドに格納される値の型をフィールド名とともに並べ，`{}` で囲むことでレコード型を示す．

`type` ⟨ 型名 ⟩ `=` `{`⟨ フィールド名$_1$⟩ `:` ⟨ 型$_1$⟩`;` `...``;` ⟨ フィールド名$_n$⟩ `:` ⟨ 型$_n$⟩`}`

定義されたレコードの値は，

`{`⟨ フィールド名$_1$⟩ `=` ⟨ 式$_1$⟩`;` `...``;` ⟨ フィールド名$_n$⟩ `=` ⟨ 式$_n$⟩`}`

で構成することができる．[1]

```
# let st1 = {name = "Taro Yamada"; id = 123456};;
val st1 : student = {name = "Taro Yamada"; id = 123456}
```

レコードの要素は，組と同様にパターンマッチでアクセスする方法と，フィールドひとつの値のみをドット記法と呼ばれる方法でアクセスする方法がある．レコードパターンは，

`{`⟨ フィールド名$_1$⟩ `=` ⟨ パターン$_1$⟩`;` `...``;` ⟨ フィールド名$_n$⟩ `=` ⟨ パターン$_n$⟩`}`

[1] 最後の ⟨ 型$_n$⟩ や ⟨ 式$_n$⟩ の後に `;` をつけてもよい．

という形で表される.

```
# let string_of_student {name = n; id = i} = n ^ "'s ID is " ^ string_of_int i;;
val string_of_student : student -> string = <fun>
# string_of_student st1;;
- : string = "Taro Yamada's ID is 123456"
```

または ⟨ レコード ⟩.⟨ フィールド名 ⟩ という形で ⟨ レコード ⟩ から ⟨ フィールド名 ⟩ に対応する要素を取り出すことができる.

```
# let string_of_student st = st.name ^ "'s ID is " ^ string_of_int st.id;;
val string_of_student : student -> string = <fun>
```

またレコードの一部のフィールドを変更しただけの新しいレコードを生成したい場合には,

{⟨ レコード式 ⟩ with
⟨ フィールド名$_1$⟩ = ⟨ 式$_1$⟩; ...; ⟨ フィールド名$_n$⟩ = ⟨ 式$_n$⟩}

という式でできる. 長いレコードの一部だけ変更したいときに便利である.

```
# type teacher = {tname : string; room : string; ext : int};;
type teacher = { tname : string; room : string; ext : int; }
# let t1 = {tname = "Atsushi Igarashi"; room = "140"; ext = 4953};;
val t1 : teacher = {tname = "Atsushi Igarashi"; room = "140"; ext = 4953}
# let t2 = {t1 with room = "142"};;
val t2 : teacher = {tname = "Atsushi Igarashi"; room = "142"; ext = 4953}
```

この時, 気をつけなければいけないのが, with はフィールドの更新を行うのではなく, 新しいレコードを生成しているということである. t1 が束縛されたレコードの room フィールドの値は t2 の定義後も変っていない. つまり,

```
# t1;;
- : teacher = {tname = "Atsushi Igarashi"; room = "140"; ext = 4953}
```

t1 の値は t2 の定義前後で変化しない.

組型とレコード型の違いについて レコード型が組型と違う点は, 各要素に名前がつき, その名前でパターンマッチを介さず要素にアクセスできること, 一部の要素だけを変更したレコードを容易に作ることができることの他に, 型宣言をしないと使えないということがある. また, レコード型は常に名前で参照され, {...} という表現は型そのものとしては 使えない. つまり, 引数の型として

```
let f (x : {name : string; id : int}) = ...
```

としたり, 入れ子になったレコードを宣言するのに,

```
type student_teacher =
  {s : {name : string; id : int};
   t : {tname : string; room : string; ext : int}};;
```

と宣言することはできない. 正しくは内側のレコード型を宣言しておいて,

```
# type student_teacher = {s : student; t : teacher};;
type student_teacher = { s : student; t : teacher; }
```

と行う．ただし，ネストしたレコードの値やパターンは直接構成可能である．

```
# let st = {s = {name = "Taro Yamada"; id = 123456}; t = t1};;
val st : student_teacher =
  {s = {name = "Taro Yamada"; id = 123456};
   t = {tname = "Atsushi Igarashi"; room = "140"; ext = 4953}}
```

型名/フィールド名についての注意・名前空間について　型名・フィールド名に使用できるのは変数名に使用できるのと同様な文字列である．フィールド名は type 宣言でレコード型を宣言することで，はじめて使用することができる．同じフィールド名を持つ別の型を宣言すると，変数の再宣言と同様，古いフィールド名の定義は隠されてしまうので注意が必要である．つまり，

```
# type foo = {name : bool};;
type foo = { name : bool; }
```

などと name フィールドを持つ別のレコード型を宣言してしまうと，新たに student の値を構成したり，name フィールドにアクセスなどができなくなってしまう．

```
# {name = "Ichiro Suzuki"; id = 51};;
  {name = "Ichiro Suzuki"; id = 51};;
          ^^^^^^^^^^^^^^^
This expression has type string but is here used with type bool
# st1.name;;
  st1.name;;
  ^^^
This expression has type student but is here used with type foo
```

ただし，変数名，フィールド名，型名は別の**名前空間**(*name space*) に属しているので，例えば，宣言済のフィールド名と同じ変数を用いても，フィールド名が隠されたりすることはない．

6.2 ヴァリアント型

ヴァリアント型が何かを説明する前に，動機付けの例として，様々な図形のデータを扱うことを考えてみよう．扱うのは以下の 4 種類の図形で，それぞれ形状を決定するためのデータをもつとする．

- 点はどれもみな同じ形・大きさをしている．
- 円は，形はどれも同じで，半径の長さで大きさを一意に決定できる．つまり整数 4, 9 などが 円を特徴づけるために必要なデータである．
- 長方形は長辺と短辺の長さで大きさ・形が決まる．
- 正方形は形はどれも同じで一辺の長さで大きさが決まる．

ここで，このような図形データを扱おうとすると，

- (2，4) のような長方形を示す値と，4 という円を示す値を混ぜて使いたい
- さらに悪いことに，7 という整数は，円も正方形も表しうる

という問題がある．ヴァリアント型はこのように，異なるデータ (表現が同じでも意味が異なる場合も含めて) を混ぜて，一つの型として扱いたい際に使うもので，ひとつひとつの値は，データの本体 ((2, 4) や 8 など) に，そのデータはなにを表すためのものか (円，長方形，正方形) の情報を付加したもので表される．

では，図形を表すヴァリアント型を定義してみよう．

```
# type figure =
#   Point
# | Circle of int
# | Rectangle of int * int
# | Square of int;;
type figure = Point | Circle of int | Rectangle of int * int | Square of int
```

ヴァリアント型もレコード型と同様 type を使って宣言する．figure は新しい型の名前である．Point, Circle, Rectangle, Square は**コンストラクタ**(*constructor*) と呼ばれ，上で述べた「そのデータは何を表すためのものか」という情報に相当する．of の後には，コンストラクタが付加される値の型を記述する． of 以下は省略可能で，省略した場合コンストラクタ単独でその型の値となる．(この例では，点同士は区別する必要がないので 0 引数コンストラクタとして宣言している．) ヴァリアント型の値は，コンストラクタを関数のように対応する型の値に「適用」することによって構成される．

```
# let c = Circle 3;;
val c : figure = Circle 3
# let figs = [Point; Square 5; Rectangle (4, 5)];;
val figs : figure list = [Point; Square 5; Rectangle (4, 5)]
```

次に，与えられた図形の面積を求める関数を考えてみよう．まずは，図形がどの種類のものであるかを調べて，それによって場合わけをする必要がある．この場合わけは match 式で行うことができる．ヴァリアント型の値に対するパターンは

⟨ コンストラクタ ⟩⟨ パターン ⟩

という形で，コンストラクタが適用された値が ⟨ パターン ⟩ にマッチすることになる．⟨ パターン ⟩ が省略された場合には引数のないコンストラクタにマッチすることになる．

```
# let area = function
#     Point -> 0
#   | Circle r -> r * r * 3  (* elementary school approximation :-) *)
#   | Rectangle (lx, ly) -> lx * ly
#   | Square l -> l * l;;
val area : figure -> int = <fun>
```

引数に対しすぐにマッチングを行うので，function を使用している．(第 5 章参照)

```
# area c;;
- : int = 27
# map area figs;;
- : int list = [0; 25; 20]
```

その他のパターン　ヴァリアント型に対するパターンマッチでは，基本的にコンストラクタの数だけ場合分けをかかなければならないが，いくつかの場合で処理が共通している場合には **or パターン** と呼ばれるパターンでまとめることができる．次の関数は，与えられた図形を囲むことができる正方形を返す関数である．(長方形は最初の辺が短辺と仮定する．)

```
# let enclosing_square = function
#     Point -> Square 1
#   | Circle r -> Square (r * 2)
#   | Rectangle (_, l) | Square l -> Square l;;
val enclosing_square : figure -> figure = <fun>
```

`Rectangle (_, l) | Square l` が or パターンと呼ばれるもので，一般的には

〈パターン$_1$〉 `|` 〈パターン$_2$〉

と書かれ，どちらかにパターンにマッチするものが全体にマッチする．各部分パターンで導入される変数 (群) は，(どちらのパターンにマッチしても `->` 以降の処理がうまくいくように) 同じ名前/型でなければならない．

また，複数の値に対してマッチをとるときには組を利用してマッチをとるのがよい．次の関数は，二つの図形が相似であるかを判定する関数である．or パターンと，組の利用法に注意．

```
# let similar x y =
#   match (x, y) with
#     (Point, Point) | (Circle _, Circle _) | (Square _, Square _) -> true
#   | (Rectangle (l1, l2), Rectangle (l3, l4)) -> (l3 * l2 - l4 * l1) = 0
#   | _ -> false;;
val similar : figure -> figure -> bool = <fun>
# similar (Rectangle (2, 4)) (Rectangle (1, 2));;
- : bool = true
```

コンストラクタの名前について　コンストラクタの名前は，変数名，型名と違って，一文字目が英大文字でなければならない．より正確には

1. 一文字目が英大文字またはアンダースコア (`_`) で，
2. 二文字目以降は英数字 (`A...Z`, `a...z`, `0...9`)，アンダースコアまたはプライム (`'`)

であるような任意の長さの文字列である．

コンストラクタもレコード型のフィールド名と同じように，同じ名前のものを再宣言してしまうと，前に定義されたコンストラクタを伴う型は使い物にならなくなってしまうので注意すること．

6.3　ヴァリアント型の応用

列挙型　Pascal, C, C++ の列挙型 (enum 型) は引数を取らないコンストラクタだけからなるヴァリアント型で表現することができる．

```
# type color = Black | Blue | Red | Magenta | Green | Cyan | Yellow | White;;
type color = Black | Blue | Red | Magenta | Green | Cyan | Yellow | White
```

enum の値が実は整数でしかない C とは違い，コンストラクタ同士の足し算などは当然定義されない．(そういう関数を書かない限り．) この型上の関数は 8 つの場合わけを行わなければいけない．

```
# let reverse = function
#     Black -> White | Blue -> Yellow | Red -> Cyan | Magenta -> Green
#   | Green -> Magenta | Cyan -> Red | Yellow -> Blue | White -> Black;;
val reverse : color -> color = <fun>
```

bool 型も列挙型の一種と見なすことができる．

```
type bool = true | false
```

そして，if 式は match 式の変形と見なすことができる．

```
if ⟨式1⟩ then ⟨式2⟩ else ⟨式3⟩
∼ match ⟨式1⟩ with true -> ⟨式2⟩ | false -> ⟨式3⟩
```

コンストラクタの名前の制限などもあるため，上の type 宣言を実際に行なうことはできないが，**bool** 型も概念的にはヴァリアント型の一種と考えるのは理解の助けになるだろう．

再帰ヴァリアント型　ヴァリアント型は of 以下に今宣言しようとしている型名を参照することで再帰的なデータ型を定義することも可能である．ここでは，もっとも単純な再帰的データの例として，自然数を表す型を考えてみよう，自然数は以下のように再帰的に定義できる．

- ゼロは自然数である．
- 自然数より 1 大きい数は自然数である．

「ゼロ」「1 大きい」という部分をコンストラクタとして考えると次のような型定義が導かれる．

```
# type nat = Zero | OneMoreThan of nat;;
type nat = Zero | OneMoreThan of nat
# let zero = Zero and two = OneMoreThan (OneMoreThan Zero);;
val zero : nat = Zero
val two : nat = OneMoreThan (OneMoreThan Zero)
```

この自然数上での加算は，データ自体の再帰的定義に従って，

- ゼロに自然数 n を足したものは n である．
- m より 1 大きい数に n を足したものは，m と n を足したものに 1 大きい数である．

と定義できる．

```
# let rec add m n =
#   match m with Zero -> n | OneMoreThan m' -> OneMoreThan (add m' n);;
val add : nat -> nat -> nat = <fun>
# add two two;;
- : nat = OneMoreThan (OneMoreThan (OneMoreThan (OneMoreThan Zero)))
```

実は，この構造はよく見てみると，リストに良く似ていることに気づく．丁度 Zero は空リスト [], OneMoreThan は cons に対応している．本質的な違いは格納できる要素の有無だけである．nat の構造を拡張すれば，例えば，整数リストを表現する型を簡単に定義することができる．

```
# type intlist = INil | ICons of int * intlist;;
type intlist = INil | ICons of int * intlist
```

(多相的なリストの定義は下で行う．)

またヴァリアント型定義はふたつの型定義を and で結ぶことによって，相互再帰的にすることも可能である．下の例は，実数と文字列が交互に現れるリストを表現したものである．

```
# type fl_str_list = FNil | FCons of float * str_fl_list
# and str_fl_list = SNil | SCons of string * fl_str_list;;
type fl_str_list = FNil | FCons of float * str_fl_list
and str_fl_list = SNil | SCons of string * fl_str_list
# let fslist = FCons (3.14, SCons ("foo", FCons (2.7, SNil)));;
val fslist : fl_str_list = FCons (3.14, SCons ("foo", FCons (2.7, SNil)))
```

この型上の関数は自然に相互再帰の二つの関数で定義される．次の関数は，この 2 種類のリスト (実数から始まるものと，文字列から始まるもの) の長さを計算する関数である．

```
# let rec length_fs = function
#     FNil -> 0
#   | FCons (_, rest_sf) -> 1 + length_sf rest_sf
# and length_sf = function
#     SNil -> 0
#   | SCons (_, rest_fs) -> 1 + length_fs rest_fs;;
val length_fs : fl_str_list -> int = <fun>
val length_sf : str_fl_list -> int = <fun>
# length_fs fslist;;
- : int = 3
```

多相的ヴァリアント型 intlist と同様にして，stringlist などを定義してゆくと，定義が酷似していることがわかる．結局，

- 末端を表すコンストラクタ
- 要素を ...list に連結するためのコンストラクタ

という構造が共通しているためである．また違いは要素の型だけである． Objective Caml では，多相型関数が，(概念的に) 関数中の型情報をパラメータ化することで様々な型の値に適用できるのと同様，型定義の一部分をパラメータ化することも可能である．ただし，関数とは違い，型パラメータを明示的に型宣言中に示してやる必要がある．多相型リストの型をヴァリアント型で定義するとすれば，

```
# type 'a list = Nil | Cons of 'a * 'a list;;
type 'a list = Nil | Cons of 'a * 'a list
```

といった定義になる．'a が型パラメータを表している．

その他，頻繁に使われる多相的データ型としては，オプション型という以下で定義される型がある．

```
# type 'a option = None | Some of 'a;;
type 'a option = None | Some of 'a
```

典型的には None が例外的な「答えがない」値を表し，正常に計算が行われた場合に Some v という形で v という値が返ってくる．(Java, C などで，null もしくは NULL を用いて例外的な値を示しているのと似ている．) オプション型は ocaml 起動時に定義済の型である．

ヴァリアント型宣言のまとめ　ヴァリアント型は，一般的には以下のような文法で宣言される．[] で囲まれた部分は省略可能である．

```
type [⟨型変数1⟩] ⟨型名1⟩ =
    ⟨コンストラクタ11⟩ [of ⟨型11⟩] | ··· | ⟨コンストラクタ1n⟩ [of ⟨型1n⟩]
and [⟨型変数2⟩] ⟨型名2⟩ =
    ⟨コンストラクタ21⟩ [of ⟨型21⟩] | ··· | ⟨コンストラクタ2m⟩ [of ⟨型2m⟩]
and [⟨型変数3⟩] ⟨型名3⟩ = ···
 ⋮
```

6.4　Case Study: 二分木

木(*tree*) 構造はリスト，配列などと並ぶ代表的なデータ構造で様々なところに応用が見られる．木は，ラベルと呼ばれるデータを格納するための**ノード**(*node*) の集まりから構成され，各ノードは，0 個以上の**子ノード**(*child node*) を持つような階層構造をなしている．階層構造の一番「上」のノードを**根**(*root*) と呼ぶ．木は様々なところに応用されていて，UNIX のファイルシステムは木構造の一例である．木構造のうち各ノードの子供の (最大) 数 n が決まっているものを n 分木と呼ぶ．$n = 1$ のものはリストと同様な構造になる．ここでは最も単純な二分木を扱っていく．

二分木構造は再帰的に次のように定義することができる．

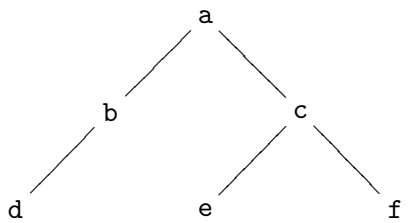
- (ラベルをもたない) 空の木 (ここでは，葉または leaf と呼ぶ) は二分木である．
- ふたつの二分木 left と right を子ノードとするノードは二分木を構成する．

これを Objective Caml の型定義に直したものが以下の定義である．

```
# type 'a tree = Lf | Br of 'a * 'a tree * 'a tree;;
type 'a tree = Lf | Br of 'a * 'a tree * 'a tree
```

`Lf` は葉，`Br` (枝/branch) はノードのことで，左右の部分木とそのラベルから `tree` を構成する．また，リストと同様，データ構造を記述するデータ型であるのでラベルの型は特定していない (多相的である)．

実際の木の例を見てみよう．例えば，



のような文字からなる木は，

```
# let chartree = Br ('a', Br ('b', Br ('d', Lf, Lf), Lf),
#                          Br ('c', Br ('e', Lf, Lf), Br ('f', Lf, Lf)));;
val chartree : char tree =
  Br ('a', Br ('b', Br ('d', Lf, Lf), Lf),
   Br ('c', Br ('e', Lf, Lf), Br ('f', Lf, Lf)))
```

と表現できる．子ノードのないノードは Br（.., Lf, Lf）で表している．

木の大きさを計る指標としては，(ラベルつき) ノードの数や，根から一番「深い」ノードまでの深さを用いる．以下の関数はそれらを求めるものである．

```
# let rec size = function
#     Lf -> 0
#   | Br (_, left, right) -> 1 + size left + size right;;
val size : 'a tree -> int = <fun>
# let rec depth = function
#      Lf -> 0
#    | Br (_, left, right) -> 1 + max (depth left) (depth right);;
val depth : 'a tree -> int = <fun>
```

明らかに，二分木 t に対しては常に，$size(t) \leq 2^{depth(t)} - 1$ が成立する．また，$size(t) = 2^{depth(t)} - 1$ であるような二分木を **完全二分木**(*complete binary tree*) と呼ぶ．

```
# let comptree = Br(1, Br(2, Br(4, Lf, Lf),
#                            Br(5, Lf, Lf)),
#                      Br(3, Br(6, Lf, Lf),
#                            Br(7, Lf, Lf)));;
val comptree : int tree =
  Br (1, Br (2, Br (4, Lf, Lf), Br (5, Lf, Lf)),
   Br (3, Br (6, Lf, Lf), Br (7, Lf, Lf)))
```

は，深さ 3 の完全二分木の例である．

```
# size comptree;;
- : int = 7
# depth comptree;;
- : int = 3
```

さて，木構造から要素を列挙する方法を考えてみよう．列挙するためには，ラベルデータに適当な順序をつける必要がある．この順序の付け方の代表的なもの 3 つ，**行きがけ順**(*preorder*), **通りがけ順**(*inorder*)，**帰りがけ順**(*postorder*) を紹介する．列挙する関数はいずれもリストを返す関数として定義されるが，説明としては木のノードを訪れる順番のつけかたとして説明する．

行きがけ順は，訪れたラベルをまず取り上げてから，左の部分木，右の部分木の順でノードを列挙する．

```
# let rec preorder = function
#     Lf -> []
#   | Br (x, left, right) -> x :: (preorder left) @ (preorder right);;
val preorder : 'a tree -> 'a list = <fun>
# preorder comptree;;
- : int list = [1; 2; 4; 5; 3; 6; 7]
```

通りがけ順は，まず左の部分木から始めて，右の木に行く前に，ラベルを取り上げる．

```
# let rec inorder = function
#     Lf -> []
#   | Br (x, left, right) -> (inorder left) @ (x :: inorder right);;
val inorder : 'a tree -> 'a list = <fun>
# inorder comptree;;
- : int list = [4; 2; 5; 1; 6; 3; 7]
```

帰りがけ順は，まず，部分木を列挙してから，最後にノードラベルを取り上げる．

```
# let rec postorder = function
#     Lf -> []
#   | Br (x, left, right) -> (postorder left) @ (postorder right) @ [x];;
val postorder : 'a tree -> 'a list = <fun>
# postorder comptree;;
- : int list = [4; 5; 2; 6; 7; 3; 1]
```

これらの定義は，@ を使っているせいで，サイズに比べて深さが大きいような (アンバランスな) 木に対して効率が良くない．これを改善したのが次の定義である．列挙済要素を引数として追加し，各ノードではそのリストに要素を追加 (cons) することだけで，計算を行っている．(この類いの定義は，効率を良くするためにわりとよくとられる手であるので，上の素直な定義を理解したうえで，マスターする価値はある．)

```
# let rec preord t l =
#   match t with
#     Lf -> l
#   | Br(x, left, right) -> x :: (preord left (preord right l));;
val preord : 'a tree -> 'a list -> 'a list = <fun>
# preord comptree [];;
- : int list = [1; 2; 4; 5; 3; 6; 7]
```

二分探索木　二分探索木(*binary search tree*) は，順序のつけられるデータを格納するときに，あとで検索がしやすいように，一定の規則にしたがってデータを配置した木である．具体的には，あるノードの左の部分木にはそのノード上のデータより小さいデータだけが，右の部分木には大きいデータだけが並んでいるような木である．例えば

```
Br (4, Br (2, Lf, Br (3, Lf, Lf)), Br (5, Lf, Lf))
```

の表す木は二分探索木であるが，

```
Br (3, Br (2, Br (4, Lf, Lf), Lf), Br (5, Lf, Lf))
```

はそうではない．

二分探索木上で，ある要素が木の中にあるかを問い合わせる mem，要素を木に追加する add 関数はそれぞれ以下のようになる．

```
# let rec mem t x =
#   match t with
#     Lf -> false
#   | Br (y, left, right) ->
#       if x = y then true
#       else if x < y then mem left x else mem right x
# let rec add t x =
#   match t with
#     Lf -> Br (x, Lf, Lf)
#   | (Br (y, left, right) as whole) ->
#       if x = y then whole
#       else if x < y then Br(y, add left x, right) else Br(y, left, add right x);;
val mem : 'a tree -> 'a -> bool = <fun>
val add : 'a tree -> 'a -> 'a tree = <fun>
```

定義からわかるようにどちらも計算の構造は同じで,

- 空の木であれば，見つからないという false，もしくは加えようとする要素だけからなる木を返す.
- ノードであれば，ノード上のデータと比較を行う．等しければ，true を返すか，そのままの木 whole を返すし，等しくない場合は大小関係に応じて，左か右の部分木にむかって探索，追加を続行する.

同じデータの集合でも，様々な形の木が考えられる．探索の効率は，木の深さによるので，深さがなるべく浅い木を維持するのが二分探索木を扱う際の目標のひとつとなる.

6.5 Case Study: 無限リスト

もうひとつ，応用例として，無限のデータ列を表すデータ型をみてみよう．無限列を有限のメモリ上にそのまま表現することはできないので，工夫が必要である．ここでは無限列は，先頭要素と,「以降の (無限) 列を計算するための関数」の組で表現する．この「関数」は特に引数となるべき情報を必要としないので，unit からの関数として表現しよう．これを書いたのが以下の type 宣言である.

```
# type 'a seq = Cons of 'a * (unit -> 'a seq);;
type 'a seq = Cons of 'a * (unit -> 'a seq)
```

seq は list や tree のように要素型に関して多相的である． Cons は要素を先頭に追加するためのコンストラクタであり， of 以下の型が示すように，tail 部分は関数型になっている．このヴァリアント型はコンストラクタが一つしかないため，ヴァリアント型をわざわざ使わずに，要素と関数の組で表現することも原理的に可能である．このような型を宣言するのは，データ抽象の手段のひとつであると考えられる．つまり，プログラマがデータにこめた意味をプログラム上で読み取ることができ，たまたま同じ表現をもつ違う意味のデータと混乱する可能性が少なくなる，という意義がある.

次の関数 from は，整数 n から 1 ずつ増える無限列を生成する.

```
# let rec from n = Cons (n, fun () -> from (n + 1));;
val from : int -> int seq = <fun>
```

fun () -> に注目したい．無限列は，fun () -> ... 使って関数を構成し，tail 部の評価を，必要なときまで遅らせているから実現できているのである．同様な関数をリストを使って定義しても,

```
let rec list_from n = n :: list_from (n + 1)
```

ひとたび呼び出されると，list_from の呼び出しを無限に行ってしまうのでうまくいかない.

from の定義にみられるように，式の評価を遅らせるために導入される関数を**サンク**($thunk$) と呼ぶ．余談であるが，サンクの明示的な導入は遅延評価機構を持つ言語であれば必要のないところである (もともと部分式の評価は必要になるまで遅らされる) ため，このような無限の構造は lazy な言語の得意とするところである.

列から先頭要素，後続の列を返す関数，先頭から n 要素をリストとして取出す関数は,

```
# let head (Cons (x, _)) = x;;
val head : 'a seq -> 'a = <fun>
```

```
# let tail (Cons (_, f)) = f ();;
val tail : 'a seq -> 'a seq = <fun>
# let rec take n s =
#   if n = 0 then [] else head s :: take (n - 1) (tail s);;
val take : int -> 'a seq -> 'a list = <fun>
# take 10 (from 4);;
- : int list = [4; 5; 6; 7; 8; 9; 10; 11; 12; 13]
```

と定義される．tail 関数の本体ではサンクに () を適用して，後続列を計算している． コンストラクタがひとつのヴァリアント型に対しては，場合わけをする必要がない ので，match を使わずに引数にパターンを直接書いても支障がない． take はリストに対してのものと同様に定義できる．

さて，無限列に対する map を定義してみよう．

```
# let rec mapseq f (Cons (x, tail)) =
#   Cons (f x, fun () -> mapseq f (tail ()));;
val mapseq : ('a -> 'b) -> 'a seq -> 'b seq = <fun>
# let reciprocals = mapseq (fun x -> 1.0 /. float_of_int x) (from 2);;
val reciprocals : float seq = Cons (0.5, <fun>)
# take 5 reciprocals;;
- : float list = [0.5; 0.33333333333333315; 0.25; 0.2; 0.16666666666666657]
```

注目すべき点は，reciprocals の値を見るとわかるように，0.5 以降の要素は take をするまで実際には計算されていないことである．take の中で，tail を呼び出す度に逆数が計算されていく．

余談であるが，複雑なデータ型にたいする関数を書いていればいるほど型情報がデバグに役立つことが実感できる．上の関数定義で，つい，サンク導入のための fun () -> を書き忘れてしまったり，サンクの評価 (tail ()) をし忘れたりするのだが，型のおかげで (エラーメッセージになれれば) すぐに間違いを発見することができる．

エラトステネスのふるい　さて，もうすこし面白い無限列の応用例をみてみたい．エラトステネスのふるいは素数を求めるための計算方法で，2 以上の整数の列から，

- 先頭の数字 2 は素数である．
- 残りの列 (3, 4, 5, . . .) から 2 の倍数 (4, 6, 8, . . .) を消す．
- 残った数字列の先頭 3 は素数である．
- 残りの列 (5, 7, 9, . . .) から 3 の倍数 (9, 15, . . .) を消す．
- 残った数字列の先頭 5 は素数である．
- 以下同様

として素数の列を求める方法である．ここでは，seq 型を用いてエラトステネスのふるいを実現してみよう．まず必要なのは，整数列から n の倍数を取り去った列を返す関数 sift である．

```
let rec sift n ... = ...;;
```

これを使うとエラトステネスのふるいは，次のように定義できる．

```
# let rec sieve (Cons (x, f)) = Cons (x, fun () -> sieve (sift x (f())));;
val sieve : int seq -> int seq = <fun>

# let primes = sieve (from 2);;
val primes : int seq = Cons (2, <fun>)
# take 20 primes;;
- : int list =
[2; 3; 5; 7; 11; 13; 17; 19; 23; 29; 31; 37; 41; 43; 47; 53; 59; 61; 67; 71]
# let rec nthseq n (Cons (x, f)) =
#   if n = 1 then x else nthseq (n - 1) (f());;
val nthseq : int -> 'a seq -> 'a = <fun>
# nthseq 1000 primes;;
- : int = 7919
```

6.6 練習問題

Exercise 6.1 以下のレコード型 loc_fig は，図形に xy 平面上での位置情報をもたせたものである．正方形，長方形は，各辺が軸に並行であるように配置されていると仮定 (長方形に関しては，Rectangle (x, y) の x の表す辺が x 軸に並行，とする．) し，二つの図形が重なりを持つか判定する関数 overlap を定義せよ．

```
type loc_fig = {x : int; y : int; fig : figure};;
```

Exercise 6.2 nat 型の値をそれが表現する int に変換する関数 int_of_nat, nat 上の掛け算を行う関数 mul，nat 上の引き算を行う関数 (ただし $0 - n = 0$) monus (モーナス) を定義せよ．(mul, monus は *, - などの助けを借りず，nat 型の値から「直接」計算するようにせよ．)

Exercise 6.3 上の monus 関数を変更して, $0-n$ $(n > 0)$ は None を返す nat -> nat -> nat option 型の関数 minus を定義せよ．

Exercise 6.4 深さ n で全てのノードのラベルが x であるような完全二分木を生成する関数 comptree x n を定義せよ．

Exercise 6.5 preord と同様な方法で，通りがけ順，帰りがけ順に列挙する関数 inord, postord を定義せよ．

Exercise 6.6 二分木の左右を反転させた木を返す関数 reflect を定義せよ．

```
# reflect comptree;;
- : int tree =
Br (1, Br (3, Br (7, Lf, Lf), Br (6, Lf, Lf)),
 Br (2, Br (5, Lf, Lf), Br (4, Lf, Lf)))
```

また，任意の二分木 t に対して成立する，以下の方程式を完成させよ．

$$\begin{aligned} \texttt{preorder(reflect(t))} &= ? \\ \texttt{inorder(reflect(t))} &= ? \\ \texttt{postorder(reflect(t))} &= ? \end{aligned}$$

Exercise 6.7 以下は，足し算と掛け算からなる数式の構文を表した型定義である．

```
# type arith =
#   Const of int | Add of arith * arith | Mul of arith * arith;;
type arith = Const of int | Add of arith * arith | Mul of arith * arith
# (* exp stands for (3+4) * (2+5) *)
# let exp = Mul (Add (Const 3, Const 4), Add (Const 2, Const 5));;
val exp : arith = Mul (Add (Const 3, Const 4), Add (Const 2, Const 5))
```

数式の文字列表現を求める関数 string_of_arith, 分配則を用いて数式を $(i_{11} \times \cdots \times i_{1n_1}) + \cdots + (i_{m1} \times \cdots \times i_{mn_m})$ の形に変形する関数 expand を定義せよ．

```
# string_of_arith exp;;
- : string = "((3+4)*(2+5))"
# string_of_arith (expand exp);;
- : string = "(((3*2)+(3*5))+((4*2)+(4*5)))"
```

(オプションとして) string_of_arith の出力結果の括弧を減らすように工夫せよ．(上の出力例では何も工夫していない．)

Exercise 6.8 1, 2, 3, 4 からなる可能な二分探索木の形を列挙し，それぞれの形を作るためには空の木から始めて，どの順番で要素を add していけばよいか示せ．

Exercise 6.9 関数 sift を定義し，$\langle$ 自分の学籍番号 $+ 3000\rangle$ 番目の素数を求めよ．

Exercise 6.10 以下で定義される ('a, 'b) sum 型は,「α 型の値もしくは β 型の値」という和集合的なデータの構成を示す型である．

```
# type ('a, 'b) sum = Left of 'a | Right of 'b;;
type ('a, 'b) sum = Left of 'a | Right of 'b
# let float_of_int_or_float = function
#     Left i -> float_of_int i
#   | Right f -> f;;
val float_of_int_or_float : (int, float) sum -> float = <fun>
# float_of_int_or_float (Right 3.14);;
- : float = 3.14
# float_of_int_or_float (Left 2);;
- : float = 2.
```

これを踏まえて，次の型をもつ関数を定義せよ．

1. 'a * ('b, 'c) sum -> ('a * 'b, 'a * 'c) sum
2. ('a, 'b) sum * ('c, 'd) sum -> (('a * 'c, 'b * 'd) sum, ('a * 'd, 'b * 'c) sum) sum
3. ('a -> 'b) * ('c -> 'b) -> ('a, 'c) sum -> 'b
4. (('a, 'b) sum -> 'c) -> ('a -> 'c) * ('b -> 'c)
5. ('a -> 'b, 'a -> 'c) sum -> ('a -> ('b,'c) sum)

第7章　参照，例外処理，入出力

本章はObjective Camlに備わっている副作用を伴う機能を概観する。Objective Camlプログラムの実行は式を値に評価する過程であった．副作用とは、式の評価中に起こる「なにか」であり、代入、ファイルへの入出力などがその一例である。副作用を伴わない式は何度評価しても結果は不変であり，また，式をその評価結果の値で置換えてもプログラムの挙動は変化しない．一方，副作用を伴う式は，評価する度に違った結果が得られる可能性があり，評価結果でその式を置換えてしまうとプログラムの挙動が変わってしまう。そのため，副作用を伴うプログラムの挙動はしばしば推論しにくくなる。

副作用を伴う関数の一例は，端末への表示を行う関数 `print_string` である．

```
# print_string "Hello, World!\n";;
Hello, World!
- : unit = ()
```

ここでは，副作用として引数の文字列が標準出力 (端末画面) へ書き出されている．また，この関数は返り値として `()` を返している．このように，副作用自体に意味があり計算結果としては重要な値を伴わない関数は，典型的には **unit** 型を返す関数として表現される．(Objective Caml では基本的に全てのプログラム中の式は何らかの値を返す．命令型言語におけるコマンドもこのような関数として表現されている．)

また，関数 `read_line` は，引数 `()` で呼ばれると，端末からの入力を待ち，その結果を文字列として返す．

```
# read_line ();;
foo              ⇐ キーボードからの入力
- : string = "foo"
```

キーボードからの入力がかわる度に同じ式 `read_line ()` の返す値は変っていく．そのため，副作用を伴う式については，いつ評価されるかを常に頭にいれてプログラムしなければならない．

7.1　参照、更新可能レコードと配列

ここまで見たプログラムでは一度作成したデータを更新する手段は与えられていない．Objective Caml にはいくつかの更新可能なデータ構造が用意されている．**参照**(*references*)，**更新可能レコード**(*mutable records*)，**配列**(*array*) である．

7.1.1　参照

Objective Caml における参照の概念はほぼメモリアドレスと考えることができ，C における変数，ポインタなどに対応する．参照の値はそのアドレスが指す先にデータを格納しており，代入操作によってその内容を書き換えることができる．t 型の値への参照値は t `ref` 型が与えられる．

参照の生成は ref 関数で行う．ref 関数は参照先に格納する初期値を引数としてとり，それを格納したアドレスを返す．(ただし，実際のアドレスの数値が観察できるわけではない．) この関数は通常の数学的な意味の関数とは違い，read_line のように呼び出す度に新しい値 (アドレス) を返す．参照から格納された値を取出すには前置オペレータ ! を使用する．また，代入式

⟨ 式$_1$⟩ := ⟨ 式$_2$⟩

は ⟨ 式$_1$⟩ を評価した結果の参照に ⟨ 式$_2$⟩ の評価結果を代入するものである．右辺の型が t であるときには，左辺の型は ref t でなければならない．代入式自体は代入という副作用を起すだけで，その結果は常に () である．以下は簡単な参照を使った例である．

```
# let p = ref 5 and q = ref 2;;
val p : int ref = {contents = 5}
val q : int ref = {contents = 2}
# (!p, !q);;
- : int * int = (5, 2)
# p := !p + !q;;
- : unit = ()
# (!p, !q);;
- : int * int = (7, 2)
```

参照を理解する上で大事なのは，格納されている値とそれのアドレスを区別することである．代入式は左辺のアドレスの**内容**を右辺の値で書き換える．! が必要なのも，この区別をはっきりさせるものと思えば良い．C などの代入文では，i = j と書いたときに，左辺は暗黙のうちに変数 i のアドレスをさし，右辺の j は変数 j の内容を指し示している．

参照値はデータ構造に格納することもできる．

```
# let reflist = [p; q; p];;
val reflist : int ref list = [{contents = 7}; {contents = 2}; {contents = 7}]
# p := 100;;
- : unit = ()
# reflist;;
- : int ref list = [{contents = 100}; {contents = 2}; {contents = 100}]
```

代入操作が reflist の出力結果に変化を与えていることに注意．ただし，reflist の 3 要素の値 (アドレス) が変化したのではなく，その指し示す値が変っただけであることに留意せよ．このように同じ参照が複数個所で使用されることによって，ある場所での代入が別の場所に影響をおよぼすことを**エイリアシング**(*aliasing*) といい，命令型言語のプログラミングで頭を悩ませる種の一つである．

また，参照の参照を考えることもでき，C におけるポインタのように扱うことができる．

```
# let refp = ref p and refq = ref q;;
val refp : int ref ref = {contents = {contents = 100}}
val refq : int ref ref = {contents = {contents = 2}}
# !refq := !(!refp);;
- : unit = ()
# (!p, !q);;
- : int * int = (100, 100)
```

7.1.2 更新可能レコード

これまでみたレコードではフィールドの値は更新できなかった．with を用いても，新しいフィールドを使った新しいレコードを生成していた．実は，Objective Caml のレコードはフィールド毎に更新可能かどうかを指定することができる．更新可能フィールドには型宣言時に mutable キーワードを用いる．

```
# type teacher = {name : string; mutable office : string};;
type teacher = { name : string; mutable office : string; }
# let t = {name = "Igarashi"; office = "140"};;
val t : teacher = {name = "Igarashi"; office = "140"}
```

フィールドの更新は，

⟨ 式$_1$⟩.⟨ フィールド名 ⟩ <- ⟨ 式$_2$⟩

という形で，⟨ 式$_1$⟩ の値であるレコードの ⟨ フィールド名 ⟩ フィールドの内容を ⟨ 式$_2$⟩ で置換える．

```
# t.office <- "142";;
- : unit = ()
# t;;
- : teacher = {name = "Igarashi"; office = "142"}
```

7.1.3 配列

配列は同じ種類の値の集合を表すデータという意味ではリストと似通っているが，長さは生成時に固定であり，どの要素にも直接 (先頭から順に辿ることなく) アクセスできる．また，各要素を更新することができる．配列には，要素型を t として t array という型が与えられる．配列の生成，要素の参照，更新はそれぞれ，

[|⟨ 式$_1$⟩; ⟨ 式$_2$⟩; ...; ⟨ 式$_n$⟩|]
⟨ 式$_1$⟩.(⟨ 式$_2$⟩)
⟨ 式$_1$⟩.(⟨ 式$_2$⟩) <- ⟨ 式$_3$⟩

という形で行い，n 要素の配列で初期値がそれぞれ ⟨ 式$_i$⟩ であるもの，配列 ⟨ 式$_1$⟩ の ⟨ 式$_2$⟩ 番目の要素，配列 ⟨ 式$_1$⟩ の ⟨ 式$_2$⟩ 番目の要素を ⟨ 式$_3$⟩ で更新，という意味である．配列の大きさを越えた整数で要素にアクセスすると Invalid_argument という例外が発生する．

7.1.4 多相性と参照

第 4 章で多相性について延べたときに，Objective Caml では多相性は値多相，すなわち let で名前が与えられる式 (let x = ... の ... 部分) が値であるときのみ，その変数 (x) が多相的に使えることを見た．ここでは，その制限の理由を説明する．

まずは，値多相の制限がなかったと仮定してみよう．以下は仮想の Objective Caml セッションである．まずは恒等関数への参照を定義する．

```
# let x = ref (fun y -> y);;
val x : ('a -> 'a) ref = {contents=<fun>}
```

参照 x に格納されているのは多相型関数であるので，その内容は様々な引数に適用できる．

```
# (!x 2, !x true);;
- : int * bool = 2, true
```

また，(引数と返り値の方が同じである) 様々な型の関数が代入できるはずである．

```
# x := fun y -> y + 1;;
- : unit = ()
```

しかし，先ほどは正しかった式 !x true が，なんと今回は true に 1 を足そうとしてしまう．

```
# !x true;;
??
```

この現象は，ただ単に let で定義される右辺の式の型に ref が入っているから，というわけではない．例えば，以下のようなプログラムで上と同様なことができてしまう．

```
# let (get, set) =
    let r = ref [] in
    ((fun () -> !r), (fun x -> r:=x));;
val get : unit -> 'a list = <fun>
val set : 'a list -> unit = <fun>
# 1 :: get ();;
- : int list = [1]
# "abc" :: get ();;
- : string list = ["abc"]
# put ["abc"];;
- : unit = ()
# 1 :: get ();;
??
```

このようなプログラムが良くない理由は，右辺が参照を生成するような式であり，かつ，生成される参照の型に型変数が含まれているためである．(最初の例であれば 'a -> 'a，ふたつめの例であれば 'a list である．) このような場合，let の後で型変数が複数の型に具体化されてしまうことで，参照に格納した値が別の型として取り出されてしまうことになる．似たような議論が更新可能レコード・配列に対しても可能である．

この事態を防ぐために Objective Caml (及び Standard ML) では，右辺の式が参照 (更新可能レコード・配列) を生成しないことがわかっている時に，多相性を許すことにしている．与えられた式が実行の過程で参照を生成するかどうかは (チューリングマシンの停止性判定と同じく) 決定不能なので，保守的な十分条件として「右辺が，参照を生成しないどころか，そもそも計算自体発生することのない，値であること」という条件を採用している．これが値多相の由来である．ここで，値とは

- 変数
- 整数や文字列などの定数
- fun, function などの関数 (匿名関数に限らず let f x = ... として定義されたものも関数である．)

- コンストラクタを値に適用したもの，各要素が値であるレコード/組

などである．値ではないものは

- 関数適用，特に `ref` の適用
- `if`, `match` 式

などである．

多相性と参照の問題は，ML の発明以来，値多相に限らずいろいろな解決策が議論されてきているが，未だに皆が納得するものは得られていない．

7.1.5 Case Study: オブジェクト指向風プログラミング

Java などのオブジェクト指向プログラミング言語におけるオブジェクトとは

内部状態が隠蔽されたデータ と それを操作するメソッド (群)

の組であると考えることができる．Objective Caml 自体に，そのようなオブジェクトを定義する仕組みがあるが，ここでは，これまでに学んだ機構を使ってオブジェクトを模倣してみよう．

まず，以下のような (一次元) 点オブジェクトを考える．

- 内部状態は (整数) 座標値 x.
- 座標値を取り出すための `get` メソッド
- 座標値を更新するための `set` メソッド
- 座標値を $+1$ するための `inc` メソッド

このために，オブジェクトのインターフェースとなる型をメソッドのレコード型として定義する．名前の最後の `I` はインターフェースであることを示している．

```
# type pointI = {get: unit -> int; set: int -> unit; inc: unit->unit};;
type pointI = { get : unit -> int; set : int -> unit; inc : unit -> unit; }
```

点オブジェクトは以下のように定義できる．“メソッド呼び出し”はレコードの要素を適用することで，実現できる．

```
# let p =
#   let x = ref 0 in
#   let rec this () =
#     {get=  (fun () -> !x);
#      set=  (fun newx -> x:=newx);
#      inc= (fun () -> (this ()).set ((this ()).get () + 1))} in
#   this ();;
val p : pointI = {get = <fun>; set = <fun>; inc = <fun>}
# p.get();;
- : int = 0
# p.inc();;
- : unit = ()
# p.get();;
- : int = 1
```

inc メソッドは自分自身の set と get を組み合わせて実装している．自分自身のメソッドの呼び出しは再帰を使って実装しており，this は「自分自身のメソッドの組」を表現している．その意味からは this は pointI 型のレコードとなるのが理想だが，ここでは再帰的に定義している関係で unit -> pointI という関数として定義している．よってメソッド内で使用するときには this () という形でレコードを求めてから .get などでメソッドを得ている．

さて，クラスはオブジェクトを生成するためのメソッド実装であるから，近似的には初期状態を引数としてオブジェクトを返す関数と思える．

```
# let pointC x =
#   let rec this () =
#     {get=  (fun () -> !x);
#      set=  (fun newx -> x:=newx);
#      inc= (fun () -> (this ()).set ((this ()).get () + 1))} in
#   this ();;
val pointC : int ref -> pointI = <fun>
# let new_point x = pointC (ref x);;
val new_point : int -> pointI = <fun>
# let p = new_point 2;;
val p : pointI = {get = <fun>; set = <fun>; inc = <fun>}
# p.inc(); p.get();;
- : int = 3
```

関数 new_point はコンストラクタのような働きをしている．

このようにして，継承などもある程度プログラミングできる．例えば，点オブジェクトを継承により拡張して色のついた点オブジェクト (新しいメソッドとして，getcolor というメソッドを考える) を，継承するメソッドを二度記述することなく，再利用するように定義することも可能である．ただし， Objective Caml の制限から同じフィールド名を別のレコード型のものとして同時には使えない[1]ので，別のメソッド名にする必要があり，煩雑になってくる．

7.2　制御構造

最初の説明でも触れたように，副作用を伴う計算は，副作用の起こる順番に気をつけてプログラミングしなければならない．例えば，以下のようなプログラム

```
# let x = print_string "Hello, " in
# print_string "World!\n";;
Hello, World!
- : unit = ()
```

で，思ったとおりの出力が得られるのは，x が束縛される値の計算 (と，それに伴う副作用) が in 以下の計算よりも前に起こるためである．Objective Caml で特に注意が必要なのは，関数呼出しなどの複数の引数の評価順序である．

```
# let f x y = 2 in
# f (print_string "Hello, ") (print_string "World\n");;
World
```

[1] Objective Caml のオブジェクトはレコードではないので，そのような制限はもちろんない

```
Hello, - : int = 2
# (print_string "Hello, ", print_string "World\n");;
World
Hello, - : unit * unit = ((), ())
```

現在の実装では，後ろの引数から順に評価が行われるが，Objective Caml の仕様としては未定義なので，引数の計算に副作用を伴うときには，let などを用いて計算順序をプログラム中に明示的にするべきである．

Objective Caml では，命令型言語に見られるような制御構造がいろいろ用意されている．まず，上のプログラムのような「コマンド列」を表現するための機能として，

⟨ 式$_1$⟩; ⟨ 式$_2$⟩; ···; ⟨ 式$_n$⟩

という形の式が用意されている．この式全体は，⟨ 式$_1$⟩ から順番に評価を行い，⟨ 式$_n$⟩ の値を全体の値とする．また，途中の式の結果は捨てられてしまうので，⟨ 式$_{n-1}$⟩ までの式は通常は副作用を伴うものである．Objective Caml では，⟨ 式$_i$⟩ ($i < n$) が **unit** 型でない場合には warning を発行する．() でなく，意味のある値を返す式の値を捨ててしまうのはバグであることが多いのからである．プログラマが，値が要らないことを確信している場合は，ignore 関数をつかって，明示的にそのことを示すことが推奨される．

```
# print_string "Hello, "; print_string "World\n";;
Hello, World
- : unit = ()
```

また，ループのための式として，while 式，for 式が用意されている．while 式は

while ⟨ 式$_1$⟩ do ⟨ 式$_2$⟩ done

という形で，**bool** 型の式 ⟨ 式$_1$⟩ の評価結果が false になるまで ⟨ 式$_2$⟩ の評価を行う．式全体は () を返す．for 式は

for ⟨ 変数名 ⟩ = ⟨ 式$_1$⟩ to ⟨ 式$_2$⟩ do ⟨ 式$_3$⟩ done

もしくは

for ⟨ 変数名 ⟩ = ⟨ 式$_1$⟩ downto ⟨ 式$_2$⟩ do ⟨ 式$_3$⟩ done

という形で，まず ⟨ 式$_1$⟩，⟨ 式$_2$⟩ を整数 n, p に評価する．その後，⟨ 変数名 ⟩ を $n, n+1, \ldots, p$ (downto の場合は $n, n-1, \ldots, p$) の順に束縛して ⟨ 式$_3$⟩ を評価する．⟨ 変数名 ⟩ の有効範囲は ⟨ 式$_3$⟩ である．式全体の値は () である．

これらのループを表現する式は，副作用を伴わないとまったく意味がないことに注意すること．副作用がなければ，while の繰り返し条件の式つねに同じ値を返すし，for 式も計算をいくらか繰り返して () を返すだけである．

以下はキーボードから入力された行を二つつなげて画面に出力し返す関数である．終了にはピリオドだけからなる行を入力する．

```
# let parrot () =
#   let s = ref "" in
#     while (s := read_line (); !s <> ".") do
#       print_string !s;
#       print_endline !s;
#     done;;
val parrot : unit -> unit = <fun>
```

print_endline 関数は引数の文字列に改行をつけて出力するための関数である．

7.3　例外処理

例外(*exception*) は、計算を進めていく過程でなんらかの理由で計算を中断せざるをえない状況を表現するための仕組みである。なんらかの理由の例としては、0 での除算、パターンマッチの失敗、ファイルのオープンの失敗、など典型的には実行時エラーの発生した状況が多い。基本的には例外が発生すると残りの計算をせずに中断して値を返さず実行を終了してしまう。(インタラクティブコンパイラでは入力プロンプトに戻る。) 例外発生は Objective Caml 式がその型の値を返さない唯一の例である。

以下は例外を発生する式である。2 番目の式はファイルを開くための関数が呼び出されているが，ファイルがないという例外が発生している．最後の式では、4 / 0 の結果の出力だけでなく残りの計算である文字列の出力が行われずに実行が中断されていることがわかるだろう。

```
# hd [];;
Exception: Match_failure ("", 63, 7).
# open_in "";;
Exception: Sys_error ": No such file or directory".
# print_string (string_of_int (4 / 0)); print_string "not printed";;
Exception: Division_by_zero.
```

それぞれ、パターンマッチングが失敗したことを示す Match, OS に対するシステムコール関連のエラーが発生したことを示す Sys_error, 0 での除算が発生したことを示す Division_by_zero, が発生している。また、いくつかの例外では例外の名前のあとに Objective Caml の値が付加されて例外に関する詳しい情報を提供している。

Objective Caml では、プログラマが新しい例外を宣言、その例外を発生させることができる。また、例外の発生をプログラム中で検知し、中断される前に処理を再開することができる。

7.3.1　exception 宣言と raise 式

新しい例外の宣言は exception 宣言で行う。

```
# exception Foo;;
exception Foo
```

例外の名前は、ヴァリアント型のコンストラクタと同様、英大文字で始まらなければならない。例外を発生させるのは raise 式で行う。

```
# raise Foo;;
Exception: Foo.
```

raise 式は、値を返さずに例外を発生させるので、任意の場所で用いることができる。別の言い方をすると raise 式には任意の型が与えられる。下の関数定義の例では、raise Foo 式が真偽値や整数の必要な場所で使われていることがわかるだろう。

```
# let f () = if raise Foo then raise Foo else 3;;
val f : unit -> int = <fun>
```

先に見た、Sys_error のような値を伴う例外は、宣言時に何の型の値を伴うかも宣言しなければならない。以下は整数を伴う例外の宣言と raise の例である。

```
# exception Bar of int;;
exception Bar of int
# raise (Bar 2);;
Exception: Bar 2.
```

Objective Caml では (raise で発生させる前の) 例外には exn 型が与えられ、第一級の値として関数に渡したり、データ構造に格納することができる。別の見方をすると，exception 宣言により，新しいコンストラクタが exn 型に追加されることになる．またヴァリアント形と同様にパターンマッチで、例外コンストラクタが適用された値を取り出すことができる。

```
# let exnlist = [Bar 3; Foo];;
val exnlist : exn list = [Bar 3; Foo]
# let f = function
#     Foo -> 0
#   | x -> raise x;;
val f : exn -> int = <fun>
# f Foo;;
- : int = 0
# f (Bar 4);;
Exception: Bar 4.
```

標準ライブラリの例外　その他の，いくつかの定義済の例外を紹介しておく．Invalid_argument, Failure は文字列を伴う例外で，いくつかのライブラリ関数で発生する．前者は関数引数が意味をなさない場合，後者は関数引数は正しいが何らかの理由で計算が失敗した場合に発生する．Not_found は探索を行う関数で，探索対象が見つからなかった場合に発生させるものである．End_of_file はファイルからの入力関数がファイルの終端に到達した場合に発生する．

例えば，assoc 関数 (5.3 節参照) は以下のように定義するのが，より「Objective Caml らしい」定義である．

```
# let rec assoc a = function
#     [] -> raise Not_found
#   | (a', b) :: rest -> if a = a' then b else assoc a rest;;
val assoc : 'a -> ('a * 'b) list -> 'b = <fun>
# assoc "Osaka" city_phone;;
- : string = "06"
# assoc "Nara" city_phone;;
Exception: Not_found.
```

また定義済の関数として，例外を発生させるだけの関数，invalid_arg, failwith なども用意されている．

```
# failwith "foo";;
Exception: Failure "foo".
```

7.3.2　例外の検知

式の評価中に起こる例外は、その種類がわかる場合には、try 式で、検知して後処理を行うことができる。try 式の一般的な形は、match 式と似ていて、

```
try ⟨ 式 ⟩ with
    ⟨ パターン1 ⟩ -> ⟨ 式1 ⟩
  | ...
  | ⟨ パターンn ⟩ -> ⟨ 式n ⟩
```

となる。まず、⟨ 式 ⟩ を評価し、例外が発生しなければその値を全体の値とする。評価途中で例外が raise された場合には、順にパターンマッチを行っていき、マッチした時点で ⟨ 式$_i$ ⟩ を評価し、try 式全体の値となる。何もマッチするものがなかった場合には、もともと発生した例外が再発生する。try 式の値は、⟨ 式 ⟩, ⟨ 式$_1$ ⟩, . . . , ⟨ 式$_n$ ⟩ のいずれかになるので、これらの式の型は一致していなければならない。

```
# try 4 + 3 with Division_by_zero -> 9;;
- : int = 7
# try 1 + (3 / 0) with Division_by_zero -> 9;;
- : int = 9
# try 1 + (3 / 0) with Sys_error s -> int_of_string s;;
Exception: Division_by_zero.
# let query_city_phone c =
#   try assoc c city_phone with Not_found -> "999";;
val query_city_phone : string -> string = <fun>
# query_city_phone "Osaka";;
- : string = "06"
# query_city_phone "Nara";;
- : string = "999"
```

例外の検知は，エラー処理の観点からだけではなく，より効率的な実行のために用いることもできる．たとえば，整数のリストの要素の積を計算するプログラムを考えてみる．もっとも素朴には，

```
# let rec prod_list = function
#     [] -> 1
#   | n :: rest -> n * prod_list rest;;
val prod_list : int list -> int = <fun>
# prod_list [2; 3; 4];;
- : int = 24
# prod_list [4; 0; 2; 3; 4];;
- : int = 0
```

と定義することができる．しかし，この定義は，リストに 0 が含まれていた場合でも，リストの最後まで律儀に計算してしまい無駄である．次の定義は 0 が出現したら 0 を返すようにして，0 以降の積を計算しないようにしているが，0 以前の要素と 0 との積を取っているという無駄がある．

```
# let rec prod_list = function
#     [] -> 1
#   | 0 :: _ -> 0
#   | n :: rest -> n * prod_list rest;;
val prod_list : int list -> int = <fun>
```

0 が出現した場合に，一度も掛け算をしないようにする方法として，option 型を用いる方法がある．

```
# let rec prod_list_aux = function
#     [] -> Some 1
#   | 0 :: _ -> None
#   | n :: rest ->
#       (match prod_list_aux rest with None -> None | Some m -> Some (m * n));;
val prod_list_aux : int list -> int option = <fun>
# let prod_list l = match prod_list_aux l with None -> 0 | Some n -> n;;
val prod_list : int list -> int = <fun>
# prod_list [2; 3; 4];;
- : int = 24
# prod_list [4; 0; 2; 3; 4];;
- : int = 0
```

ここでは，リストに 0 が出現したことを None で表現している．たしかに，0 が出現したら，残りのリストの積を計算せずに終了しているのだが，プログラム的にはあまり美しくない．これを例外を利用して，None を返す代わりに例外を発生させるようにしたのが最後の定義である．

```
# exception Zero_found;;
exception Zero_found
# let rec prod_list_aux = function
#     [] -> 1
#   | 0 :: _ -> raise Zero_found
#   | n :: rest -> n * prod_list_aux rest;;
val prod_list_aux : int list -> int = <fun>
# let prod_list l = try prod_list_aux l with Zero_found -> 0;;
val prod_list : int list -> int = <fun>
# prod_list_aux [2; 3; 4];;
- : int = 24
# prod_list_aux [4; 0; 2; 3; 4];;
Exception: Zero_found.
# prod_list [2; 3; 4];;
- : int = 24
# prod_list [4; 0; 2; 3; 4];;
- : int = 0
```

このように例外は大域ジャンプに相当することを実現することができ，うまく使うとプログラムの最適化，可読性の向上にも役に立つ反面，乱用するとプログラムの制御の流れが分かりにくくなる危険性もあるので注意して使おう．

7.4 チャネルを使った入出力

Objective Caml には、最初に見た print_string 以外にも様々な入出力用の関数が用意されている．

標準入出力，標準エラー出力関連の関数 print_char, print_string, print_int, print_float, print_endline, print_newline は標準出力 (通常は端末画面) に引数を書き出す関数である．それ

ぞれ，引数の型が異なったり，末尾の改行の出力機能がついていたりするが，基本的な動作は同じである．

print を prerr に変えれば，標準エラー出力に書き出す関数になる．

また，標準入力 (通常はキーボード) から読み込みを行う関数に，read_line, read_int, read_float がある．

ファイル操作とチャネル　Objective Caml では入出力先を表現するのにチャネルという抽象的な概念を用いている．チャネルは通信路のことで，ここに向かって書き出したり，ここから読み込みを行うことで，その先につながった何か (ファイル，ディスプレイ) に書き込んだりすることができる．チャネルはさらに，入力用，出力用のものにわかれており，Objective Caml ではそれぞれ，in_channel 型，out_channel 型として定義されている．また，標準入力などは定義済の値として用意されている．

```
# (stdin, stdout, stderr);;
- : in_channel * out_channel * out_channel = (<abstr>, <abstr>, <abstr>)
```

チャネルに対する入出力には，input_char, input_line, output_char, output_string などの関数で読み書きを行う．

また，ファイル名から新たなチャネルを生成することもできる．出力用のチャネルは open_out，入力用のチャネルは open_in 関数で得ることができる．どちらも，ファイル名の文字列を引数として受け取る．また，使い終わったチャネルは close_out, close_in 関数で閉じなければならない．

その他の入出力関数については，マニュアルの 18 章を参照されたい．

7.5　Objective Caml の文法について補足

いくつか，文法について補足しておく．以前に見たように演算子には優先順位がついていて強いものから結合し部分式を構成する．例えば ; は if よりも結合が弱いので，

```
if false then print_string "a"; print_string "b";;
```

は

```
(if false then print_string "a"); print_string "b";;
```

と同じ意味である．また，

```
if false then print_string "a"; print_string "b" else ();;
```

は文法エラーになってしまう．(; まで読んだ時点で if 式が終わったと判断してしまうためで先にある else はみてくれない．)

分かりにくいのは，if など複数のキーワードから構成され，同じ優先度を持つ式が入れ子になった場合である．基本的には Objective Caml の文法で let, if, match, try, function, fun のように，終端を示すキーワードがないものは，できる限り先まで含めて式の一部であるように読まれる．例えば，

```
if a then if b then c else d
```

は

```
if a then (if b then c else d)
```

のことである．内側の if が右にできる限り伸びようとしているのがわかるだろう．
また，match の入れ子，練習問題にある match と try の組合わせは要注意である．

```
match e with
  A -> match e' with B -> ... | C -> ...
| D -> ...
```

は D の分岐も内側の match の一部として考えられてしまうので，D が外側の match であるようにするには以下のように括弧が必要である．

```
match e with
  A -> (match e' with B -> ... | C -> ...)
| D -> ...
```

練習問題の try 式も括弧がないと最後の | _ -> 以降が try 式の一部と見なされてしまうのである．
第 4 週の優先順位の表と上のルールで，一見して明らかでない部分式の結合はわかるはずである．(プログラムが見にくくならない程度に括弧をつけるのもよい習慣である．)

7.6 練習問題

Exercise 7.1 ref 型の値の表示を見て気づいている人もいるかもしれないが，ref 型は以下のように定義された 1 フィールドの更新可能なレコードである．

```
type 'a ref = { mutable contents : 'a };;
```

関数 ref, 前置オペレータ !，中置オペレータ := の定義を，レコードに関連した操作で書け．

Exercise 7.2 与えられた参照の指す先の整数を 1 増やす関数 incr を定義せよ．

```
# let x = ref 3;;
val x : int ref = {contents = 3}
# incr x;;
- : unit = ()
# !x;;
- : int = 4
```

Exercise 7.3 以下で定義する funny_fact は再帰的定義 (rec) を使わずに階乗を計算している．どのような仕組みで実現されているか説明せよ．

```
# let f = ref (fun y -> y + 1)
# let funny_fact x = if x = 1 then 1 else x * (!f (x - 1));;
val f : (int -> int) ref = {contents = <fun>}
val funny_fact : int -> int = <fun>
# f := funny_fact;;
- : unit = ()
# funny_fact 5;;
- : int = 120
```

Exercise 7.4 参照を使って階乗関数を定義してみよう．... 部分を埋めよ．

```
let fact_imp n =
  let i = ref n and res = ref 1 in
    while (...) do
      ...;
      i := !i - 1
    done;
    ...;;
```

Exercise 7.5 階乗関数 fact を負の引数に対して Invalid_argument を発生させるように改良せよ．

Exercise 7.6 7.1.4 節で述べた恒等関数への参照の例を実際にインタラクティブ・コンパイラで試し，テキストに書いた挙動との違い，特に，参照の型を説明し，どのようにして true に 1 を足すような事態の発生が防がれているか説明せよ．

Exercise 7.7 上でみたオブジェクト指向風プログラミングの延長で継承などを表現してみようと思う．以下は色を表す型と色付き点オブジェクトのインターフェースである．

```
# type color = Blue | Red | Green | White;;
type color = Blue | Red | Green | White
# type cpointI = {cget: unit -> int;
#                 cset: int -> unit;
#                 cinc: unit->unit;
#                 getcolor: unit-> color};;
type cpointI = {
  cget : unit -> int;
  cset : int -> unit;
  cinc : unit -> unit;
  getcolor : unit -> color;
}
```

色付き点オブジェクトは座標に加え，色を状態としてもつとする．また，cget, cinc は pointC のメソッドを継承し，cset は座標のセットとともに色を白にセットするように実装したい．以下がその試みである．

```
# let cpointC x col=
#   let super = pointC x in
#   let rec this =
#     {cget=  super.get;
#      cset=  (fun x -> super.set x; col := White);
#      cinc= super.inc;
#      getcolor = (fun () -> !col)} in
#   this;;
val cpointC : int ref -> color ref -> cpointI = <fun>
# let new_cpoint x col = cpointC (ref x) (ref col);;
val new_cpoint : int -> color -> cpointI = <fun>
```

しかし，この実装はうまく働かない．

```
# let cp = new_cpoint 0 Red;;
val cp : cpointI =
  {cget = <fun>; cset = <fun>; cinc = <fun>; getcolor = <fun>}
```

```
# cp.cinc();;
- : unit = ()
# cp.cget();;
- : int = 1
# cp.getcolor();;
- : color = Red
```

cinc 中では，座標をセットするので色は白になっていてほしいのに元のままである．この理由をプログラムの挙動とともに説明し，うまく cinc が作動するようにプログラムを書き換えよ．ただし，継承を模倣したいので，同じことをするメソッドに相当する関数は一度書くだけで (上の super.get のような形で) 再利用すること．

(ヒント: pointC の定義を以下のように変更する．)

```
# let pointC x this () =
#   {get=  (fun () -> !x);
#    set=  (fun newx -> x:=newx);
#    inc= (fun () -> (this ()).set ((this ()).get () + 1))};;
val pointC : int ref -> (unit -> pointI) -> unit -> pointI = <fun>
# let new_point x =
#   let x = ref x in
#   let rec this () = pointC x this () in
#   this ();;
val new_point : int -> pointI = <fun>
```

Exercise 7.8 以下の関数 change は，お金を「くずす」関数である．

```
# let rec change = function
#     (_, 0) -> []
#   | ((c :: rest) as coins, total) ->
#       if c > total then change (rest, total)
#       else c :: change (coins, total - c);;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
([], 1)
  ................function
      (_, 0) -> []
    | ((c :: rest) as coins, total) ->
        if c > total then change (rest, total)
        else c :: change (coins, total - c)..
val change : int list * int -> int list = <fun>
```

与えられた (降順にならんだ) 通貨のリスト coins と合計金額 total からコインのリストを返す．

```
# let us_coins = [25; 10; 5; 1]
# and gb_coins = [50; 20; 10; 5; 2; 1];;
val us_coins : int list = [25; 10; 5; 1]
val gb_coins : int list = [50; 20; 10; 5; 2; 1]
# change (gb_coins, 43);;
- : int list = [20; 20; 2; 1]
# change (us_coins, 43);;
- : int list = [25; 10; 5; 1; 1; 1]
```

しかし，この定義は先頭にあるコインをできる限り使おうとするため，可能なコインの組合わせがあるときにでも失敗してしまうことがある．

```
# change ([5; 2], 16);;
Exception: Match_failure ("", 198, 17).
```

これを，例外処理を用いて解がある場合には出力するようにしたい．以下のプログラムの，2 個所の ... 部分を埋め，プログラムの説明を行え．

```
let rec change = function
    (_, 0) -> []
  | ((c :: rest) as coins, total) ->
      if c > total then change (rest, total)
      else
        (try
          c :: change (coins, total - c)
         with Failure "change" -> ...)
  | _ -> ...;;
```

Exercise 7.9 print_int 関数を stdout, output_string などを用いて定義せよ．

Exercise 7.10 二つのファイル名を引数にとって，片方のファイルの内容をもう片方にコピーする関数 cp を定義せよ．とくに一度開いたファイルは最後に閉じることを忘れないように．

第8章　単純なモジュールとバッチコンパイル

Objective Caml では，バッチコンパイラ ocamlc を用いて Objective Caml プログラムを書いたファイルから，実行可能ファイルを生成することができる．また，プログラムを複数のファイルに分割してコンパイルすることもできる．このとき，分割された単位をモジュールと呼ぶ．Objective Caml ライブラリもモジュールの形で提供されており，その使用方法は同じである．

8.1　ライブラリモジュールの使い方

まずは，ライブラリの使用法を学びながら，既存のモジュールの使い方をみてゆく．Objective Caml のライブラリは，List, Array, Sort モジュールなどのデータ構造に関するもの，Printf などの入出力に関連するもの，Sys モジュールなどの，OS や Objective Caml 処理系とのインターフェースをとるためのもの等が豊富に用意されている．ひとつひとつの詳しい機能などはマニュアルの 19 章をみてほしい．ここでは List モジュールと Queue モジュールを題材にする．

モジュール内の関数は，⟨ モジュール名 ⟩.⟨ 関数名 ⟩ と言う形で呼び出すことができる．

```
# List.length [5; 6; 8];;
- : int = 3
# List.concat [[4; 35; 2]; [1]; [9; -4]];;
- : int list = [4; 35; 2; 1; 9; -4]
```

以前にみてきたリスト操作のための関数 rev, append, map, fold_left, fold_right 等は，ほとんど List モジュールに定義されている．

Queue モジュールは，いわゆる待ち行列のデータ構造を実装したもので，リストのように同種のデータをまとめて格納するのに用いる．add, take という要素を追加，取り出す関数が用意されているが，特徴は「先入れ先出し」であり，add した順番でしか take できない．Queue モジュールはモジュール内で t という名前の独自の型を定義している．この場合には，その型にもモジュール名がついた形 'a Queue.t で表される．

```
# let q = Queue.create ();;
val q : '_a Queue.t = <abstr>
# Queue.add 1 q; Queue.add 2 q;;
- : unit = ()
# Queue.take q;;
- : int = 1
# Queue.take q;;
- : int = 2
# Queue.take q;;
Exception: Queue.Empty.
```

最後に発生した例外 Empty は Queue モジュール内で定義されたものである．

open 宣言　同じモジュールを使い続けると，いちいちモジュール名をつけるのが面倒になってくる．open 宣言は文字通りモジュールを「開く」もので，モジュール内の定義がモジュール名なしでアクセスできるようになる．

```
# open List;;

# length [3; 9; 10];;
- : int = 3
```

この open 宣言は，open した時点ですでに宣言されている同名の定義を隠してしまうので，同名の定義を提供する複数のモジュールを開くときには順番に注意した方がよい．隠されてしまった名前は，結局モジュール名つきの記法でアクセスすることになる．

8.2　バッチコンパイラによる実行可能ファイルの生成

もっとも単純な ocamlc の使用法は，Objective Caml の宣言を書いたファイル (拡張子は .ml) を用意して，シェルのコマンドラインから

ocamlc -o〈出力ファイル名〉〈ソースファイル名〉

としてコンパイラを 起動すると，〈出力ファイル名〉という実行可能なファイルが生成される． -o オプションを省略すると a.out という名前で生成される．

```
igarashi@zither:text> cat hello.ml
let _ = print_string "Hello, World!\n"
igarashi@zither:text> ocamlc hello.ml
igarashi@zither:text> a.out
Hello, World!
igarashi@zither:text> cat fact.ml
let rec fact n =
  if n = 0 then 1 else n * fact (n - 1)
let _ = print_int (fact 10)
igarashi@zither:text> ocamlc -o fact10 fact.ml
igarashi@zither:text> ./fact10
3628800igarashi@zither:text>
```

バッチコンパイルされるファイルの中には宣言の並びだけが許され，インタラクティブコンパイラで見たような，式だけからなるものははじかれるので，let _ = ... のような，式を評価して結果を捨てる宣言として記述している．(C の main 関数に相当するものがなく，ただ単に上から評価を行っていく．)

さて，最初に述べたようにソースファイルは複数のファイルに分割することができる．Objective Caml システムでは，一つ一つのソースファイルがモジュールに対応し，UNIX 上でのファイル名の先頭を大文字にしたものが，Objective Caml でのモジュール名になる．例えば，foo.ml のソースファイル中の宣言は，他のファイルからは，モジュール Foo にあるものとしてアクセスされる．下は，fact.ml と main.ml である．

```
igarashi@zither:samples> cat fact.ml
let rec fact n =
```

```
  if n = 0 then 1 else n * fact (n - 1)
igarashi@zither:samples> cat main.ml
(* main.ml *)
let _ =
  print_int (Fact.fact 10);
  print_newline();
```

main.ml では Fact モジュール内の fact 関数を Fact.fact という名前で使用している．コンパイラには，二つのファイル名を，依存されているものから順に並べる．

```
igarashi@zither:samples> ocamlc -o fact10 fact.ml main.ml
igarashi@zither:samples> fact10
3628800
```

また，-c オプションを用いると，各モジュールを個別にコンパイルすることができる．中間的なオブジェクトファイルとして，.cmi という拡張子を持つ，モジュールのインターフェース情報 (モジュール内にどんな名前の関数がどんな型で宣言されているかの情報・シグネチャとも呼ぶ) をコンパイルしたファイルと，.cmo という拡張子を持つモジュール自体をコンパイルしたファイルが生成される．.cmi ファイルは，そのモジュールを使用するファイルがコンパイルされるときに必要であり，(.cmo 自体は必要ではない．) 下の例で，main.ml を先にコンパイルすることはできない．そして，.cmo はあとでリンクして実行可能ファイルを生成することができる．

```
igarashi@zither:samples> ocamlc -c fact.ml
igarashi@zither:samples> ocamlc -c main.ml
igarashi@zither:samples> ocamlc -o fact10 fact.cmo main.cmo
```

各モジュールのインターフェースは -i オプションで標準出力に書き出すことができる．

```
igarashi@zither:samples> ocamlc -i -c fact.ml
val fact : int -> int
```

プログラマはこれを見て，各関数に意図通りの型が与えられているかどうかを確認することができる．

関連図書

[1] Guy Cousineau and Michel Mauny. *The Functional Approach to Programming*. Cambridge University Press, 1997. 現在，関数型プログラミングの教科書の中で Caml を直接対象にした (英語では) 唯一のもの.

[2] Matthias Felleisen and Daniel P. Friedman. *The Little MLer*. The MIT Press, 1998. プログラミングにおける再帰・型の概念を Standard ML を使って解説．内容は OCaml にも，ほぼそのままあてはまる.

[3] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system release 3.06: Documentation and user's manual*, 2002. `http://pauillac.inria.fr/caml/ocaml/htmlman/index.html`.

[4] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, MA, 1997. Standard ML の形式的定義．数学的な定義が並んでいるもので解説はないので読むのは困難．コンパイラ実装者など言語仕様を正確に知りたい人向け.

[5] L. C. Paulson. *ML for the Working Programmer*. Cambridge University Press, second edition, 1996. Standard ML の教科書.

[6] Jeffrey D. Ullman. *Elements of ML Programming*. Prentice Hall, ML97 edition, 1998. Standard ML の教科書．初版は和訳がある.

[7] 大堀 淳. **プログラミング言語** *Standard ML*. 共立出版, 2001. 日本語で書かれた Standard ML の数少ない教科書.