情報システム科学実習II
第2回 基本データ型，変数の宣言，簡単な関数

担当: 山口 和紀・五十嵐 淳

2001年10月17日

keywords: **インタラクティブコンパイラ，型，型安全性，有効範囲, 環境，関数**

1 インタラクティブコンパイラを使う

OCaml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，Lisp, Scheme など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，主に ocaml の方を用いて進めていく．

1.1 簡単な使い方

起動方法は (前回の設定がうまくいっていれば) シェルのプロンプトで ocaml と打つだけでよい．(quena% がシェルプロンプトである．)

```
quena% ocaml
        Objective Caml version 3.02

#
```

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

```
# 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の-)，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．

いくつか誤った入力例についてもみていこう．

```
# 2 + 3 - ;;
Characters 9-11:
Syntax error
# 5 + "abc";;
Characters 4-9:
This expression has type string but is here used with type int
# 4 / 0;;
Uncaught exception: Division_by_zero.
```

(端末上ではエラーの発生した位置が数字ではなく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，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．OCaml 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．エラーメッセージ自体は，エラーの発生した個所は文字列であるのに，整数が必要な個所 (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 OCaml 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 Control-D を，もしくは #quit;; と入力することで行う．

```
quena% ocaml
        Objective Caml version 3.02
```

```
# #quit;;
quena%
```

以上が基本的な使い方であるが，シェルから直接 ocaml を実行すると入力途中で間違いに気づいても，矢印キー，backspace などで編集することができず，Control-C を使うしかなく不便なので，付録 Aに示す方法で Emacs, Mule を通して操作するのが便利だろう．

1.2 ディレクティブによるコンパイラへの指示

ocaml 内では，コンパイラの動作を制御するためのいくつかのディレクティブと呼ばれる命令が利用できる．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 [1] 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．

```
quena% cat two.ml
1 + 1;;
quena% ocaml
        Objective Caml version 3.02

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

#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．

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";;
Characters 5-6:
Syntax error
```

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．(" で囲まれた，文字列定数，詳しくは後述，を除く．) また，コメントは入れ子になってもよい．

EUC でエンコードされている限り，コメント，文字列として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．

2 基本データ型と演算

OCaml 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．この値を計算する過程を**評**

価(*evaluation*) という．最も簡単な式は，整数などの，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，OCaml 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．

> **参照透明性と副作用について** プログラム中の `1 + 2` という式は，`2 + 1` や，その値 `3` を代わりに用いても，プログラムの意味 (実行結果) が変ることはない．このように，部分式を「等価な」式と置き換えられる性質を**参照透明性**(*referential transparency*) という．とくに，式をその値で置き換えても良い．
>
> ところが,「`1 + 2` を計算して，ディスプレイにその結果を表示し，結果は `1 + 2` の値」であるような式は，その評価結果つまり値である `3` で置き換えることはできない．(ディスプレイへの出力がなくなってしまう．) このように，ひとたび値を計算すると失われてしまう「効果」を**副作用**(*side-effect*) という．副作用を伴う式は，その評価結果で置き換えることはできない．(その意味で，副作用を伴う式とその結果は等価でない，ともいえる．)
>
> 関数型言語では副作用のない式がプログラムの主要な要素である．副作用のない式は，式だけでその意味が決まるために，プログラムの性質の推論がおこないやすい．一方，BASIC, C など多くの**命令型言語**(*imperative language*) では，プログラムの主要な構成要素は，変数への代入，ファイルの入出力などを行う，副作用を伴う文 (もしくは命令) である．

> **メタ変数について** 資料中，OCaml 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 OCaml の式の一部の文字であることに対して，i, j は (この資料中では) 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 OCaml 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．

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，例えば副作用を起こすだけの式は，**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]ただし C の void 型はそれに属する値をもたない．

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．)

2.3 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 10 を基底とする指数表現 `31.415e-1` $(= 31.415 \times 10^{-1})$ 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;;
Characters 0-1:
This expression has type int but is here used with type float
# float_of_int(1) +. 3.4;;
- : float = 4.4
# 1 + (int_of_float 3.4);;
- : int = 4
```

関数の引数のまわりの `()` は省略可能である[3]．これ以外にも三角関数 `sin`, `cos`, `tan`, 自乗根 `sqrt` などが用意されている．詳しくはマニュアルを参照のこと．

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 1 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．

[3]ML の流儀としては引数のまわりの `()` は (可能な場合は) 省略される．ひとつには数学的な記法 (例 $\sin\theta$ など) により近いということからである．式が複雑でどう結合するかわかりにくい場合には，上の最後の例のように関数呼び出し全体を `()` で括る．

表 1: 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

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

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 1 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!"
```

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;;
Characters 0-1:
This expression has type int but is here used with type float
```

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"
```

分岐の二つの式 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 `then`-節には **unit** 型の式が来なければならない．)

2.7 型システムと安全性

OCaml 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[4]．OCaml 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[5] 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 OCaml 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 (OCaml, 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．

2.8 練習問題

Exercise 2.1 次の式の型と評価結果は?

[4] 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．

[5] 先に見た 0 での除算は，型エラーではないとしたが，(OCaml 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. `sin 3.14 /. 2.0 ** 2.0 +. cos 3.14 /. 2.0 ** 2.0` ⇒ `1.0`
3. `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 のみをもちいて書き直せ．式 b_2 `||` b_2 も同様に書き直せ．

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．

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．

```
# 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，変数が束縛されるのは計算結果にであって，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．

OCaml 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"
```

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: OCaml キーワード

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.

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.

OCaml 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の OCaml 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.

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

図 1: ocaml 起動時の大域環境

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

図 2: let 宣言実行後の大域環境

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 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) ．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．

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_    _
```

Exercise 2.6 上の `let` 宣言の,「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.」という説明における，x はプログラム変数を表すメタ変数である．`let x =` e というときの違いを述べよ.

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．OCaml 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，何度も，違った半径に対して「同じような式」を入力するのは，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.

OCaml 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．() 内の `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，その具体的表現 ("3" など) がないことに注意.

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.

OCaml 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，後程みていくことにする．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 = の前にいれる．

ここでの，関数宣言の文法をまとめると，

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

となる[6]．[ ] 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，関数宣言の直前に宣言されたものが参照される．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．

4.1 練習問題

Exercise 2.7 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$ = 121.04 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$ = 121.04 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')

[6] 以降を通じて関数宣言の文法は拡大されていく．

A caml-mode in emacs

`ftp://ftp.inria.fr/lang/caml-light/bazar-ocaml/` では OCaml を Emacs (XEmacs, Mule を含む) 上で開発するための Emacs Lisp プログラム `ocaml-mode-xxxx.tar.gz` が配布されている．これを使うことで，

- プログラムのインデント (字下げ) 付け
- Emacs のバッファ内での `ocaml` の実行 (inferior process) および，他のバッファからのプログラムの入力

などを行うことができる．

先週の環境設定が正しく行われていれば，`.ml` という拡張子を持つファイルを読み込むと自動的に caml mode に入ってくれる．

主なキーバインディングは表3の通り．`M-x run-caml` で新たなバッファが開いて，emacs の編集機能を使いながら，コンパイラを使うことができる．コンパイラとの対話用バッファでは `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．コンパイラの終了は `C-d` (一文字消去) ではなくて `C-c C-d` になる．

参考文献

[1] Xavier Leroy, Damien Doligez, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The Objective Caml system release 3.02: Documentation and user's manual*, 2001. `http://pauillac.inria.fr/caml/ocaml/htmlman/index.html`.

表 3: caml-mode キーバインディング

TAB	現在の行のインデント
M-C-q および	
C-c C-q	現在の行を囲むフレーズ (式として意味のあるまとまり) のインデント
M-C-h	フレーズにマーク
C-c w	バッファに while 式を挿入
C-c t	try 式を挿入
C-c m	match 式を挿入
C-c l	let 式を挿入
C-c i	if 式を挿入
C-c f	for 式を挿入
C-c b	begin 式を挿入
M-x run-caml	ocaml を起動．起動中には以下のコマンドが使用可能
M-C-x および	
C-c C-e	フレーズを caml プロセスに送る
C-c C-r	リージョンを caml プロセスに送る
C-c C-s	caml プロセスのバッファを表示
C-c ‘	caml プロセスに送った式のコンパイルエラーを順次表示
	コンパイラの起動されたバッファでは以下のコマンドが使用可能
C-c C-c	入力途中で中断しプロンプトに戻る
C-c C-d	caml の終了
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