情報システム科学実習II
第3回 再帰による繰り返し

担当: 山口 和紀・五十嵐 淳

2001年10月24日

keywords: **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とる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．

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．

OCaml 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 = 20.9439510233
```

“`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 をその結果に束縛して，
3. e_2 の評価

を行い，全体は e_2 の評価結果になる．変数 x の有効範囲は e_2 であるので，vol_cone の宣言以降では base は参照できない．

```
# base;;
Characters 1-5:
Unbound value base
```

また，e_1 は x の有効範囲に含まれないことに注意．

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)^2 *)
#   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．また複次的ではあるが，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;;
```

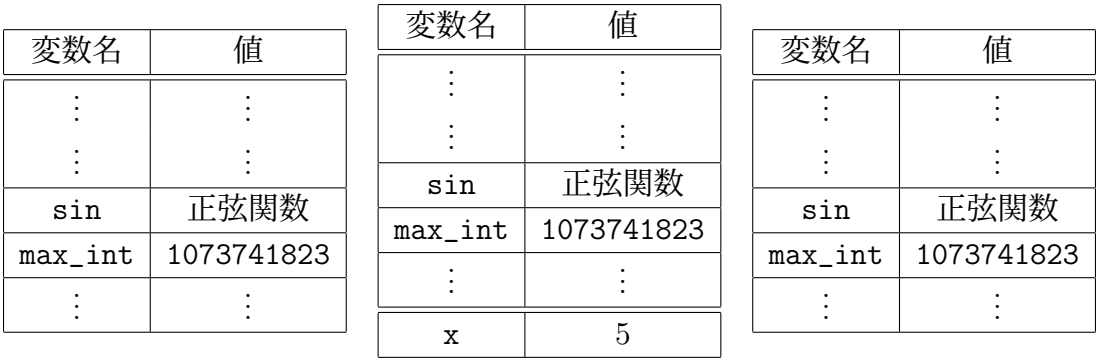


図 1: 式 `let x = 3 + 2 in x + 7` の実行．二重線以下が一時的に発生した束縛を表す．

```
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．

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 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．

1.1 練習問題

Exercise 3.1 `let` x `=` e_1 `in` e_2 という表記において，x は OCaml の変数を表すメタ変数である．`let x =` e_1 `in` e_2 との違いはなんだろうか？

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 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．

Exercise 3.2 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 3.3 トップレベルでの以下の 2 種類の宣言の違いは何か？

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

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．

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 (集合でいえば積集合) を定義することがある．それと同じように OCaml 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．また，結果の値に `()` がついていないが，実は OCaml では組の `()` を省略できる場合がある．この資料中ではいくつかの例外を除いて `()` を明示的に書くことにする．

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
```

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) から,
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・使うことができる.

ような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 (要素の型は何でもよい!) 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.

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．変数パターンは今まで使ってきた `let x = ...` などのもので，パターンとして 解釈すると,「何にでもマッチし，`x` をマッチした値に束縛する」ものである．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.

[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.

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;;
Characters 83-84:
This variable is bound several times in this matching
```

もうひとつ，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"
```

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>
```

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，(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 OCaml 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 (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).

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．

2.4 練習問題

Exercise 3.4 2 実数の相乗平均をとる関数 geo_mean を定義せよ．

Exercise 3.5 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 3.6 次のふたつの型

- `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 3.7 `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 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．

3.1 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[2]，
- $n! = n \cdot (n-1)!$，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 OCaml で定義すると，

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

となる．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[3]．

```
# let rec x = x + 1;;
Characters 13-18:
This kind of expression is not allowed as right-hand side of ‘let rec’
```

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．

[2] $0! = 1$ と定義することもある．
[3] 現在の OCaml 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．

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[4]，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．OCaml 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 4 * fact(4-1)
           →     4 * fact (4 - 1)
           →     4 * fact 3
           →···→ 4 * (3 * fact (3-1))
           →···→ 4 * (3 * fact 2)
           →···→ 4 * (3 * (2 * fact (2-1)))
           →···→ 4 * (3 * (2 * fact 1))
           →···→ 4 * (3 * (2 * 1))
           →···→ 4 * (3 * 2)
           →···→ 4 * 6
           →···→ 24
```

[4]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．

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)．

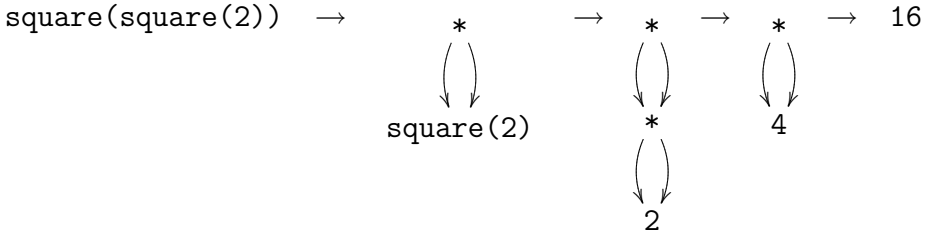
この欠点は，とにかく引数を先に評価していくこと (この性質を eagerness, strictness と呼ぶことがある) に起因する．これに対して，いまから述べるふたつの戦略は，lazy な評価と呼ばれ，「引数は使うまで評価しない」戦略である．

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，副作用との相性が悪い．実際これらの言語では副作用を伴う機能がない．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．)

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「あとで計算される部分」である 4 * (3 * (...)) といったものを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，$n \cdot ((n-1) \cdot (n-2)!) = (n \cdot (n-1)) \cdot (n-2)!$ が成立するため $(n-2)!$ の計算にとりかかる前に，$n \cdot (n-1)$ を先に計算することで,「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，先に計算するべき式の情報が必要であり，

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

のような定義になる．引数 p が，fact で発生していた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 (1, 4);;
- : int = 24
```

以下に，facti (1, 4) の評価の様子を示す．

```
facti (1, 4) → if 4 = 1 then 1 else facti(1 * 4, 4 - 1)
             → facti (4, 3)
             → if 3 = 1 then 4 else facti(4 * 3, 3 - 1)
             → facti (12, 2)
             → if 2 = 1 then 12 else facti(12 * 2, 2 - 1)
             → facti (24, 1)
             → if 1 = 1 then 24 else facti(24 * 1, 1 - 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．このような定義を，**反復的**(*iterative*) である，もしくは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．しかし，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．

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

のように，局所的に宣言するか，

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

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

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，線形再帰な定義になっていることに注意.

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

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

と定義することもできる.

3.5　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 は偶数である.
- 1 は奇数である.
- $n-1$ 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 = 1 then true else even(n - 1);;
val even : int -> bool = <fun>
val odd : int -> bool = <fun>
# even 6;;
- : bool = true
```

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 = 3.1440864153
# 4.0 *. pos 800;;
- : float = 3.14221726315
```

ゆっくりと $\frac{\pi}{4}$ に収束して行く.

3.6 練習問題

Exercise 3.8 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 3.9 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 3.10 if 式は OCaml 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 3.11 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 3.12 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先週のテキストを参照のこと．)

Exercise 3.13 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．

A　レポートその 1: 締切 11 月 6 日

必修課題:　2.1, 2.2, 2.6 (3 と 4), 3.2 (2 と 3), 3.5, 3.8(1), 3.9, 3.10, 3.12 から 2 つ.

オプション課題:　第 2 回，第 3 回の資料の残り全部の問題.

レポートの提出方法　レポートはメールで以下のフォーマットに従って提出してください.

- 提出先アドレス: `fp-report@graco.c.u-tokyo.ac.jp`
- subject: フィールドに `report 1` と書いて,
- 先頭行に (* 〈学籍番号〉〈名前〉 *)
- 各問題の最初に (* 〈問題番号〉 *)
- 説明・考察は全てコメント (* ... *) として書いてください. (レポートのファイルはコンパイラに通して大丈夫なようにしてください.)

レポートには受取りの返事をしますので，数日経っても返事が来ない場合は `igarashi@graco.c.u-tokyo.ac.jp` まで，問い合わせをお願いします. (正しいアドレスに送らないと，返事が遅れるかもしれません.)

注意事項　解答だけではなく，できるかぎり説明を加えてください. 特にプログラムを書く問題の場合，コードだけでなく，各部分の役割などを考察すること.