16 関数型プログラミング

16.1 なぜ関数型プログラミングなのか?

セクション1.8では、チャーチとチューリングの両者が、最初のコン ピュータが作られるずっと前に計算のモデルを提唱していたことを述べ ました。チューリング・マシンは、更新可能なストア、つまり命令が実行されるとその内容が変更されるメモリ・セルのセットに基づいているという点で、現代のコンピュータと非常によく似ている。これはフォン・ノイマン・アーキテクチャとしても知られている。

チャーチの計算モデルの定式化（ラムダ計算と呼ばれる）は、関数の数学的概念に基づいており、まったく異なる。この定式化は、記述可能な計算という点ではチューリングのものと完全に等価であるが、計算の実用的な形式論としては、関数型アプローチはあまり普及していない。1956年に開発された言語Lispは、ラムダ計算モデルに似た関数型アプローチを使っていますが、命令型プログラミングスタイルを推奨する多くの機能を含んでいます。

1980年代、関数型プログラミングのさらなる研究により、非常にきれいな理論的基礎を持ちながら効率的に実装できる言語が生まれた。現代の関数型プログラミング言語とLispの主な違いは、型と型チェックがこれらの言語の基本概念であるため、プログラムの信頼性と効率の両方が大幅に向上していることです。

信頼性の高いプログラムを書く上での問題の多くは、更新可能なストアを使用することに直接起因しています：

- 単に値を計算するのではなく、（配列のインデックスやポインタを使用して）メモリセルを直接変更するため、メモリが「汚れる」可能性がある。
- サブプログラムが副作用を持つ可能性があるため、複雑なプログラムを コンポーネントから構成するのは難しい。そのため、プログラム全体から切り離してサブプログラムの効果を理解することは不可能である。

強力な型チェックとオブジェクト指向プログラミングのカプセル化技術によって、これらの問題を軽減することはできますが、完全に排除することはできません。関数型アプローチを採用することで、これらの問題は両方とも解消される。

16.1 なぜ関数型プログラミングなのか？

16.2 Functions

MLでは，関数の名前とその正式なパラメータを等式化することで関数を 定義します.

$$\text{fun even n} = (\text{n mod 2} = 0)$$

16.1.なぜ関数型プログラミングなのか？ # 他の言語でも概念は同じですが、よく使われる言語である標準MLに基づいて議論 します。関数が定義されると、その関数を値に適用することができます：

$$\text{even 4} = \text{true}$$
$$\text{even 5} = \text{false}$$

プログラミング言語で型が変数に関連付けられるように、型はすべての関数に関連付けられます。evenの型は

$$\text{even: int} -> \text{bool}$$

meaning that it maps a value of integer type into a value of Boolean type.

Expressions used in ML can contain conditions:

$$\text{fun min (x,y)} = \text{if x} < \text{y then x else y}$$

関数の適用例を評価すると，次のようになる.

$$\text{min (4,5)} =$$
$$(\text{if x} < \text{y then x else y}) \text{ (4,5)} =$$
$$\text{if 4} < \text{5 then 4 else 5} =$$
$$\text{if true then 4 else 5} =$$
$$4$$

これはif文ではなく，条件式であることに注意.

$$x < y\ ?\ x : y$$

アプリケーションの評価は結果を生成する。関数型プログラミングでは、関数は正確に1つの引数を持つとみなされます。[2] 複数の引数が必要な場合は、デカルト積関数を使用してタプル（ペア、トリプルなど）を作成する必要があります。従って，(4,5)はint×int型であり，関数minはint型である.

$$\text{min: (int} \times \text{int)} -> \text{int}$$

MLには命令的代入がありますが、ここでは無視します。 the

関数型プログラミングでは、プログラミング用語のパラメータではなく、数学用語の引数を使うのが一般的です。

タプルを使用する代わりに、各引数に1つずつ適用される関数を定義することが # できます：

```
fun min_c x y = if x < y then x else y
```

これは数学者H.B.カリーにちなんでカリー関数と呼ばれます。この関数を一連の引数に適用すると、最初の適用で別の関数が生成され、それが2つ目の引数に適用される。

The function min_c takes one integer argument and creates a new function, also of one argument:

```
min_c 4 = if 4 < y then 4 else y
```

この関数は、次に別の単一の引数に適用することができる：

```
min_c 4 5 =
(if 4 < y then 4 else y) 5 =
if 4 < 5 then 4 else 5 =
if true then 4 else 5 =
4
```

この関数は、その後、別の単一の引数に適用することができる # キュリー関数は、部分評価を使って新しい関数を定義することができる：

```
fun min_4 = min_c 4

min_4 5 =
(if 4 < y then 4 else y) 5 =
if 4 < 5 then 4 else 5 =
if true then 4 else 5 =
4
```

16.3 複合型

Lists

特にintegerやBooleanのようなあらかじめ定義された型はそうです. リストは

```
[2, 3, 5, 7, 11]          [true, false, false]
```

はそれぞれint list型とbool list型です。リストコンストラクタは、空のリストには [ ] を、既存のリストに要素を追加して作成される空でないリストには element :: list を使用します。コンストラクタは、パターン・マッチで関数を定義する際に使用できます：

```
fun    member [] e = false
|      member [e :: tail] e = true
|      member [e1 :: tail] e = member tail e
```

メンバの型は[3]である。

```
member: int list × int -> boolean
```

であり、以下のように読むことができる：

memberがリストLに適用され、(その後)要素eに適用される場合、引数に従っ て評価される：(1) Lが空の場合、eはLのメンバではない (2) eがLの最初の要素である場合、eはLのメンバである (3) そうでない場合、Lの最初の要素であるe1はeと同じではないので、eがLの末尾のメンバであるかどうかを（再帰的に）チェックする。

コンパイラは，引数の型と結果の型から関数の型を自動的に推測します．もしコンパイラが関数の型を推論できない場合には，式を曖昧に しないために十分な型宣言をする必要があります．型チェックは静的なものであり、関数が値に適用されるときに、関数の型が引数の型と一致するかどうかがコンパイル時にチェックされます。

この関数は再帰的であることに注意。再帰は関数型プログラミング言語において非常に重要である。「ステートメント 」がない場合、式評価のループを作る唯一の方法である。

最後の例として、挿入ソートアルゴリズムをMLで書く方法を示します。このアルゴリズムは，トランプで手札を並べ替えるときに使います．

```
fun    insertion_sort [] = []
|      insertion_sort head :: tail =
           insert_element head insertion_sort tail
and
fun    insert_element x [] = [x]
|      insert_element x head :: tail =
           if x < head then x :: head :: tail
           else head :: (insert_element x tail)
```

これらの関数の型は以下の通りです：

```
insertion_sort: int list -> int list
insert_element: int -> int list -> int list
```

この表記法に慣れれば、このようなプログラムを読むのは簡単である：

[3] Actually, the type is 't list × 't boolean, but type variables are not introduced until the next section.

ソートされた空リストは空リストである。空でないリストは、最初の要素xを取り出し、リストの残りを末尾にソートし、ソートされたリストの適切な位置にxを挿入することでソートされる。

この関数は再帰的であることに注意。xを空でないリストに挿入するには、xとリストの先頭を比較する：(1)xがheadより小さい場合、xをリストの新しい先頭要素にする。(2)そうでない場合、headと、xを挿入して作成されたリストの残りで 構成される新しいリストを作成する。

というような新しい関数を作ることができる：

```
insert_element: int -> (int list -> int list)
```

この関数は整数を別の関数にマップし、整数リストを整数リストにマップします。部分評価を使って、次のような新しい関数を作ることができる：

```
fun insert_4 = insert_element 4
```

これは整数リストに4を挿入する関数である。

同じアルゴリズムの命令型プログラムと比較すると、インデックスもフォー ループもない。さらに、演算子 「i」をその型の2つの値を比較する適切なブール関数に置き換えるだけで、他の型のオブジェクトのソートにも即座に一般化できる。リストの作成に明示的なポインターは必要なく、ポインターはデータ表現に暗黙的に含まれている。もちろん、どのような言語でもリストのソートは配列のインプレース・ソートよりも効率的ではありませんが、多くのアプリケーションではリストの使用は実用的です。

Defining new types

本書を通して、プログラミング言語が現実世界をモデル化するためには、新しい型の定義が不可欠であることを見てきた。現代の関数型プログラミング言語もこの機能を持っている。ノードが整数でラベル付けされた木の（再帰的な）型を定義してみよう：

```
datatype int tree =
        Empty
|       T of (int tree × int × int tree)
```

This is read:

int treeは新しいデータ型で、その値は以下の通りである：(1)新しい定数値Empty、または(2)木、整数、別の木からなるトリプルに適用されるコンストラクタTによって形成される値。

新しい型を定義した後は、ツリーを処理する関数を書くことができます。例えば

```
fun    sumtree Empty = 0
|      sumtree T(left, value, right) =
           (sumtree left) + value + (sumtree right)
```

ツリーのノードにラベル付けされた値を追加します：

```
fun    mintree Empty = maxint
|      mintree T(left, value, right) =
           min left (min value (mintree right))
```

ノードにラベル付けされたすべての値の最小値を計算し、空のツリー上の最大の整数maxintを返す。

木構造にマッチする新しいデータ型を定義し、そのデータ型に関数を書く。明示的なポインタやループは必要なく、再帰とパターンマッチングだけです。

16.4 高階関数

関数型プログラミングでは、関数は型を持つ普通のオブジェクトなので、他の関数の引数になることができます。例えば、compare関数を追加の引数として追加するだけで、insert要素の汎用的な形式を作成できます：

```
fun    一般的な挿入要素の比較 x [ ] = [x] 一般的な挿入要
|      素の比較 x head :: tail = # これは事実上不可能です。
           if compare x head
           then x :: head :: tail
           else head :: (general_insert_element compare x tail)
```

If string_compare is a function from strings to Boolean:

```
string_compare: (string × string) − > bool
```

applying general_insert_element to this argument:

```
fun string_insert = general_insert_element   string_compare
```

gives a function of type:

```
string − > string list − > string list
```

命令型言語とは異なり、ジェネリックやテンプレートのような追加の構文やセマンティクスがなくても、この一般化は自然に達成されることに注意。

16.4 高階関数 # しかし、一般的なinsert要素の型は何でしょうか? 最初の引数は 「function from a pair of anything to Boolean 」型、2番目の引数は同じ 「anything 」型、3番目の引数は 「anything 」のリストでなければならない。型変数は 「anything 」の省略形として使用され、関数の型はこのようになる：

general_insert_element: (('t × 't) −> bool) −> 't −> 't list

型変数はMLではアポストロフィで始まる識別子として記述されます。

高階関数、つまり関数を引数に持つ関数の使用は、ジェネリックスのような静的な構文に 限定されません。非常に便利な関数に map がある：

```
fun    map f [] = []
|      map f head :: tail = (f head) :: (map f tail)
```

この関数は最初の引数を値のリストに適用し、結果のリストを生成する： results. For

```
map even [1, 3, 5, 2, 4, 6] = [false, false, false, true, true, true]
map min [(1,5), (4,2), (8,1)] = [1, 2, 1]
```

せいぜい関数へのポインタを引数として受け取るサブプログラム を書くことができるくらいで、関数の引数のシグネチャの可能性ごとに # 異なるサブプログラムが必要になります。

Note that the construction is safe. The type of map is:

map: ('t1 −> 't2) −> 't1 list −> 't2 list

つまり、引数リスト 't1 list の要素はすべて関数 't1' の引数と互換性がなければならず、結果リスト 't2 list は関数結果型 't2' の要素のみから構成される。

高階関数は、命令型言語で必須とされる制御構造のほとんどを抽象化します。別の例を挙げると、関数accumulateは、mapのように結果のリストを作成する代わりに、関数アプリケーションを複合化する：

```
fun    accumulate f initial [] = initial
|      accumulate f initial head :: tail = accumulate f (f initial head) tail
```

accumulateを使うと、様々な便利な関数を作成することができます。関数

```
fun minlist = accumulate min maxint
fun sumlist = accumulate "+" 0
```

整数リストの最小値と和をそれぞれ計算する。例えば：

```
minlist [3, 1, 2] =
accumulate min maxint [3, 1, 2] = ac
cumulate min (min maxint 3) [1, 2] =
 accumulate min 3 [1, 2] = accumulat
e min (min 3 1) [2] = accumulate min
 1 [2] = accumulate min (min 1 2) [
] = accumulate min 1 [ ] = # 16.4.
```

1

高階関数はリストに限定されるわけではありません。木を走査し、各ノードで関数を適用する関数を書くことができます。さらに、関数は型変数に定義することができるので、新しいデータ型を定義してもそのまま使うことができます。

16.5 遅延評価と熱心評価

命令型言語では、関数の呼び出し時に実際のパラメータが評価されることを 常に想定している： before

```
n = min(j+k, (i+4)/m);
```
C

これを専門用語でイーガー評価という。しかし、イーガー評価にはif文（セクション6.2）で遭遇した問題があり、評価を短絡させるための特別な構文を定義しなければならなかった：

```
if (N > 0) and then ((Sum / N) > M) then ...
```
Ada

条件式はどのように評価されるべきか？

```
if c then e1 else e2
```

関数型プログラミング言語で定義される？eager評価では、c、e1、e2を評価し、その後に条件演算を実行することになる。もちろん、これは受け入れられません。次の式は、空のリストの先頭を取ることはエラーとなるため、イーガー評価を使用すると失敗します：

```
もし list = [ ] then [ ] else hd list
```

この問題を解決するために，MLにはif関数の評価に関する特別なルールがあ ります.

例えば、ifを普通の関数として定義することができます：

```
fun    if true x y = x
|      if false x y = y
```

When if is applied, the function is simply applied to its first argument, producing:

```
(if list=[]  []  hd list)  [] =
if []=[]  []  hd [] =
if true  []  hd [] =
[]
```

[4]For this reason lazy evaluation is also known as *call-by-need*.

そして、hd [ ]を評価しようとはしない。

遅延評価は、命令型言語のcall-by-nameパラメータ・パッシング・メカニズムに似ています。このメカニズムは命令型言語では問題があります。なぜなら、副作用の可能性があるため、再利用のために評価を計算して保存することによる最適化が不可能だからです。副作用のない関数型プログラミングでは問題はなく、遅延評価を使う言語（Miranda[5]など）が実装されている。遅延評価はイーガー評価よりも効率は落ちますが、大きな利点があります。

遅延評価の主な魅力は、インクリメンタルな評価が可能であることで、効率的な アルゴリズムのプログラミングに利用できる。例えば、上で定義した型の整数値の木を考えてみよう。2つの木を比較して、ノードの順序の下で同じ値の集合を持つかどうかを確認するアルゴリズムをプログラムしたいと思うかもしれない。これは次のように書ける：

```
fun    equal_nodes t1 t2 = compare_lists (tree_to_list t1) (tree_to_list t2)
```

treeからlistへの関数は、木を走査し、ノードの値のリストを作成します。リストの比較では、2つのリストが等しいかどうかをチェックします。 Eager評価では、たとえ探索の最初のノードが等しくなくても、比較が行われる前に両方のツリーが完全にリストに変換されます！遅延評価では、計算を続けるために必要な範囲で関数を評価するだけでよい。

リストを比較する関数とツリーをリストに変換する関数は以下のように定義される:[6].

```
fun    compare_lists [] [] = true
|      compare_lists head :: tail1 head :: tail2 = compare_lists tail1 tail2
|      compare_lists list1 list2 = false

fun    tree_to_list Empty = []
|      tree_to_list T(left, value, right) =
          value :: append (tree_to_list left) (tree_to_list right)
```

遅延評価の例は以下のようになる（ここでは関数をcmpとttlと省略し、省略記号は非常に大きな部分木を示す）：

```
cmp   ttl T( T(Empty,4,Empty), 5, . . ) ttl T( T(
      Empty,6,Empty), 5, . . ) = 5 :: append (ttl
cmp    T(Empty,4,Empty)) (ttl . . . ) 5 :: append
       (ttl T(Empty,6,Empty)) (ttl . . . ) append =
cmp    (ttl T(Empty,4,Empty)) (ttl . . . ) append
       (ttl T(Empty,6,Empty)) (ttl . . . ) =
       . . .
cmp   4 :: append [] (ttl . . . )
```

MirandaはResearch Software Ltd.の商標です.

[6]This traversal is called *preorder* because the root of a subtree is listed before the nodes in the subtree.

```
        6 :: append [] (ttl ...) =
false
```

必要な引数のみを評価することで、右側のサブツリーの不要な走査は完全に回避される。MLのようなイーガー評価を使う言語で同じ効果を得るには、プログラミング上のトリック が必要です。

遅延評価のもう1つの利点は、対話型プログラミングやシステム・プログラミングに適していることである。例えばターミナルからの入力は、単に無限の値のリストとみなされます。もちろん、遅延評価器はリスト全体を評価することはありません。その代わり、値が必要なときはいつでも、ユーザーに値を入力するように促した後、リストの先頭を削除します。

16.6 Exceptions

MLで式を評価すると例外が発生することがあります．定義済みの例外は，主に0による除算や空リストの先頭を取るような 定義済みの型を使って計算する際に発生する例外です．プログラマは，例外を宣言することもできます．

```
例外: intのBadParameter ;
```

この場合、この値は関数から返される値としてのみ使用されます。 # その後、例外を発生させて処理することができます：

```
fun only_positive n =
    if n <= 0 then raise BadParameter n else .
    ...

val i = ...;
val j = only_positive i
    handle
        BadParameter 0 => 1;
        BadParameter n => abs n;
```

この関数は、引数が正でない場合にのみ例外BadParameterを発生させます。この関数が呼び出されると，例外ハンドラが呼び出し元の式に付加され， 例外が発生した場合に返される値が指定されます．この値は、例外が発生した時点で、さらに計算を行うために使用することができます。

16.7 Environments

MLプログラムでは，関数の定義や式の評価に加え，以下のような宣言も # 含むことができます．

```
val i = 20
val s = "Hello world"
```

このように，MLはストアを持ちますが，命令型言語と異なり，このストアは 更新可能ではありません．

```
val i = 35
```

関数型プログラミングの環境では、バインディングを変更することはでき ないということです。MLでの宣言は，オブジェクトが生成されるだけで，変更することはでき # ないという点で，C言語のconst宣言と似ています． # しかし，MLでの再宣言は，前の宣言を隠しますが，C言語では同じスコープで # オブジェクトを再宣言することは違法です．

ブロックの構造化は，定義や式の中で宣言を局所化することで可能です．次の例では、判別式のローカル宣言を使用して2次方程式の根を計算しています：

```
val a = 1.0 and b = 2.0 and c = 1.0
let
    D = b*b - 4.0*a*c
in
    ( (-b+D)/2.0*a, (-b-D)/2.0*a )
end
```

各宣言は値を名前に束縛する．いつでも有効なすべてのバインディングの集合を環境と呼び、ある式は環境のコンテキストで評価されると言う。実際に環境については、命令型言語におけるスコープと可視性の文脈で詳しく説明した。 # 関数型プログラミングの環境では、バインディングを変更することはできないという違いがある。

環境内に抽象データ型を含めるのは簡単な拡張です。これは、型宣言に一連の関数を追加することで実現で by きる：

```
abstype int tree =
        Empty
|       T of (int tree × int × int tree)
with
    fun sumtree t = ...
    fun equal_nodes t1 t2 = ...
end
```

この宣言の意味は，リストされた関数のみが抽象型のコンストラクタにアク セスできるというものです．さらに，抽象型は型変数でパラメータ化することができます．

```
abstype 't tree = ...
```

これはAdaで一般的な抽象データ型を作るのと似ている。

MLには，モジュールを定義したり操作したりするための非常に柔軟なシス テムがあります．その基本的な概念は，C++のクラスや抽象的なデータ型を定義する Adaのパッケージのように，宣言（型や関数）からなる環境をカプセル化する 構造体です．しかし，MLでは，構造体はそれ自身がシグネチャと呼ばれる型を持 つオブジェクトです．構造体は，ある構造体を別の構造体にマッピングする関数である ファンクタを用いて操作することができます．これは，パッケージやクラスのテンプレートを具象型にマッピングする ジェネリックの概念を一般化したものです．ファンクターは、構造体の情報を隠したり共有したりするために使うことができる。これらの概念の詳細は本書の範囲を超えているので、興味のある読者はMLの教科書を参照されたい。

関数型プログラミング言語は、複雑なデータ構造やアルゴリズムを扱うアプリケーションのために、簡潔で信頼性の高いプログラムを書くために使うことができる。関数型プログラムの効率が許容できない場合でも、プロトタイプとして、あるいは最終的なプログラムの作業仕様として使用することができる。

16.8 Exercises

1. curried 関数 min c の型は?

```
fun min_c x y = if x < y then x else y
```

2. sumtreeとmintreeの型を推論する。

3. 一般的な挿入要素の定義を言葉で書きなさい。

4. リストを追加する関数を書き、同じ関数がaccumulateを使って定義できることを示せ # 3.

5. 木のリストを受け取り、各木の最小値のリストを返す関数を書きなさい。

6. Infer the types of compare_lists and tree_to_list.

7. What does the following program do? What is the type of the function?

```
fun    filter f [] = []
|      filter f h::t = h::(filter f t),        if f h = true
|      filter f h::t = filter f t,             otherwise
```

8. ある数列(x_1 , ... , x_n)の標準偏差は、各数値の2乗の平均の平方根から平均の2乗を引いたものとして定義される。数のリストの標準偏差を計算するMLプログラムを書きなさい。ヒント：mapとaccumulateを使ってください.

9. 例えば、1 + 2 + 4 + 7 + 14 = 28. プログラムの概要は以下の通り：

```
fun isperfect n =
    let fun addfactors ...
in addfactors(n div 2) = n end;
```

10. MLの例外とAda, C ++, Eiffelの例外を比較する。