13　パターン

Mathematica の特徴的でありしかも強力な機能の一つに、パターンおよびパターンマッチがある．これを効果的に用いると、プログラミングが非常に簡単かつ容易になる。ただ、高度なパターンマッチを行うために、実行時間が長くなる場合があるので注意すること．

パターン

パターンとは，式の一部あるいは全部を曖昧化したものである．たとえば，「何かの正弦関数と何かの余弦関数の和」という式は， `Sin[_]+Cos[_]` というパターンで表される．そして，パターンの曖昧化された部分を具体的なもので置き換えることで得られる式は，そのパターンにマッチすると言う．たとえば，`Sin[x]+Cos[y+z]` は `Sin[_]+Cos[_]` にマッチすると言う．また，パターンにマッチさせること，あるいはパターンにマッチする部分を見つけることをパターンマッチングという．

パターンの簡単な使用例として， `Cases` をあげておく．

- `Cases[{e1, e2, ... }, pattern]` は，パターン `pattern` にマッチする `ei` をリストにして返す．

次は，リストの中から．「何かの正弦関数」というパターン `Sin[_]` にマッチするものをリストアップしている．

```
In[9]:= Cases[{x, 1, 2/3, Sin[x], Sin[y+z], a+b, a+b+c}, Sin[_]]
Out[9]= {Sin[x], Sin[y+z]}
```

ブランク _

パターンの基本となるものは，ブランクと呼ばれるもので，それはアンダースコア `_` で表される．ブランクは何にでもマッチするパターンである．次の例では，全てのものにマッチしている．

```
In[13]:= Cases[{x, 1, 2/3, Sin[x], Sin[y+z], a+b, a+b+c}, _]
Out[13]= {x, 1, 2/3, Sin[x], Sin[y+z], a+b, a+b+c}
```

ブランクをいくつか含んだ式も，パターンになる．次の例では，`b` と何かとの和にマッチするパターン `b+_` を用いている．

```
In[12]:= Cases[{x, 1, 2/3, Sin[x], Sin[y+z], a+b, a+b+c}, b+_]
Out[12]= {a+b, a+b+c}
```

`a+b` と `b+a` とは等しいということを *Mathematica* が理解しているため， `a+b` が `b+_` にマッチしている．また， `a+b+c` は `b+(a+c)` に等しいことも理解しているため， `a+b+c` が `b+_` にマッチしている．

ブランクは，要素の部分だけでなく，関数の部分にも用いることができる．次の例は，x を引数としている関数にマッチする _[x] を用いて， Sin[x] と Cos[x] とにマッチさせている．

```
In[28]:= Cases[{x, 1, 2/3, a/b, Sin[x], Cos[x], a+b, a+b+c}, _[x]]
Out[28]= {Sin[x], Cos[x]}
```

一つのパターンの中にはブランクがいくつあってもよい．次の例は f[_, _] というパターンを用いて，引数がちょうど二つであり関数名が f であるような式にマッチさせている．この場合，一つ目のブランクと二つ目のブランクとは独立であるので，二つのブランクに当てはまるものが等しくても等しくなくてもよい。

```
In[34]:= Cases[{f[a], f[a, b], f[b, b], f[a, b, c]}, f[_, _]]
Out[34]= {f[a, b], f[b, b]}
```

> 練習問題： Sin 関数の式だけにマッチするパターンを作り，Cases を用いて {Sin[x], Cos[x], Sin[2x], Tan[y]} の中から {Sin[x], Sin[2x]} を抽出せよ．

パターンを使うときの注意

ブランクも何かのものなので，それが式の中にあると通常のもののように計算され自動的に簡約化される．すなわち _+_ は 2倍の _ であり，_-_ は 0 となる．

```
In[86]:= _ + _
Out[86]= 2 _
```

```
In[87]:= _ / _
Out[87]= 1
```

しかし，これは思わぬ間違いを犯す原因となる．たとえば，何か二つのものの和にマッチさせようと _+_ というパターンを用いると，これは 2 _ すなわち何かの2倍，というものにマッチするパターンとなる．

```
In[88]:= Cases[{a + b, 2 a}, _ + _]
Out[88]= {2 a}
```

_-_ , _*_ , _/_ というパターンも同様である．

```
In[91]:= Cases[{a - b, 0}, _ - _]
Out[91]= {0}
```

```
In[93]:= Cases[{a*b, a^2}, _*_]
Out[93]= {a^2}
```

```
In[94]:= Cases[{a/b, 1}, _/_]
Out[94]= {1}
```

_^_ はこれ以上簡約化されないので，これは目的通りに作用する．

```
In[95]:= Cases[{a^b, c}, _^_]
Out[95]= {a^b}
```

以上のことは， *Mathematica* の仕様ではあるが，この仕様は間違っていると思われる．

ダブルブランク __

ブランク _ を二つ続けて並べたもの __ を，ダブルブランクといい，1 個以上何個でも並んだ列にマッチする．次の例のパターン f[__] は 1 個以上の引数を持った関数 f にマッチする．

```
In[37]:= Cases[{f[], f[a], f[a, b], f[b, b], f[a, b, c]}, f[__]]
Out[37]= {f[a], f[a, b], f[b, b], f[a, b, c]}
```

次の例のパターン f[a, __] は，関数 f の中の引数の先頭が a であり，その後に 1 個以上の引数が続くようなものにマッチする．「a の後に 1 個以上」ということなので， f[a] にはマッチしない．

```
In[38]:= Cases[{f[], f[a], f[a, b], f[b, b], f[a, b, c]}, f[a, __]]
Out[38]= {f[a, b], f[a, b, c]}
```

練習問題： 最初の要素が a で，最後の要素が b である，長さが 3 以上のリストにマッチするパターンを作り，Cases を用いて {{a, b, c}, {a, c, b}, {a, d, e, b}, {a, b}} から {{a, c, b}, {a, d, e, b}} を抽出せよ．

トリプルブランク ___

ブランク _ を三つ続けて並べたもの ___ を，トリプルブランクといい，0 個以上何個でも並んだ列にマッチする．次の例だと，関数 f の引数があるなしに関わらず，すべてのものにマッチする．

```
In[39]:= Cases[{f[], f[a], f[a, b], f[b, b], f[a, b, c]}, f[___]]
Out[39]= {f[], f[a], f[a, b], f[b, b], f[a, b, c]}
```

次の例のパターン f[___, a, ___] は，関数 f の引数のいずれかが a であるようなものにマッチする．

```
In[41]:= Cases[{f[], f[a], f[a, b], f[b, b], f[a, b, c]}, f[___, a, ___]]
Out[41]= {f[a], f[a, b], f[a, b, c]}
```

次の例のパターン f[___, b, ___, b, ___] は，関数 f の引数の中に b が 2 個以上あるようなものにマッチする．

```
In[48]:= Cases[{f[], f[a], f[a, b], f[b, b], f[a, b, c]}, f[___, b, ___, b, ___]]
Out[48]= {f[b, b]}
```

練習問題： 二番目の要素が c で，最後の要素が b であるリストにマッチするパターンを作り，Cases を用いて {{a, b, c}, {a, c, b}, {a, d, e, b}, {a, b}} から {{a, c, b}} を抽出せよ．

パターンに参照名をつける

パターンにマッチしたものに対して，パターンに参照名をつけることにより，マッチしたものをその参照名で参照することができる．これは，関数の定義や変換規則を用いて処理を施す場合などに用いられる．

参照名のつけ方

- ブランクの直前にシンボルをつけて x_ または x__ または x___ のようにすることにより，そのブランクまたはダブルブランクまたはトリプルブランクにマッチしたものをそのシンボル x で参照することができる．
- パターンの直前にシンボル名とコロンをつけて x:pattern のようにすることにより，そのパターンにマッチしたもの全体をそのシンボル x で参照することができる．

関数定義

f[x_]:= x^2 などの関数の定義の仕方は以前に述べたが，この定義式の左辺にある x_ がブランク _ に参照名 x がついたものである．定義式の右辺において，このブランクにマッチしたもの（すなわち関数 f の引数）を，この名前 x で参照している．

これが通常の関数定義の仕方であるが，もっと具体的なパターンを用いて，効果的な関数定義をおこなうことができる．次の例は，長さが2のリストを引数にとる関数 f を定義している．

```
In[47]:= Clear[f]

In[14]:= f[{x_, y_}] := Cos[x] + Sin[y] + Tan[x + y]

In[15]:= f[{a, b}]
Out[15]= Cos[a] + Sin[b] + Tan[a + b]
```

この関数は，引数が長さ2のリストであるときのみ定義されているので，それ以外の引数の場合には，関数は計算されない．

```
In[16]:= f[{a}]
Out[16]= f[{a}]
```

これと同じ計算をする関数 g を通常のように g[x_]:= で定義すると次のようになる．

```
In[48]:= Clear[g]

In[17]:= g[x_] := Cos[x[[1]]] + Sin[x[[2]]] + Tan[x[[1]] + x[[2]]]

In[18]:= g[{a, b}]
Out[18]= Cos[a] + Sin[b] + Tan[a + b]
```

この定義は，さきほどの f の定義よりも，見辛く分かりにくいものになっている．しかも，このように定義された関数は，引数が長さ2のリストでないときにも，無理やりにこの定義式通りに計算しようとするので，エラーを引き起こす．

```
In[19]:= g[{a}]
         Part::partw : {a}のパート2は存在しません.
         Part::partw : {a}のパート2は存在しません.
Out[19]= Cos[a] + Sin[{a}〚2〛] + Tan[a + {a}〚2〛]
```

x:pattern の形式も関数定義に用いることができる．

```
In[49]:= Clear[h]

In[44]:= h[(x : Cos[_]) + (y : Sin[_])] := x y

In[46]:= h[Cos[a] + Sin[b]]
Out[46]= Cos[a] Sin[b]
```

練習問題： {x, {y, z}} の形の引数をとり， x+(y+z)/x の値を返す関数 f を定義せよ.

可変長引数関数の定義

引数の個数が定まっていないような関数，すなわち可変長引数関数を定義するときには，ダブルブランク __ あるいはトリプルブランク ___ を用いる.

次の定義にはダブルブランクが用いられている．これにマッチするのは，関数 f が1個以上の引数をとっているときであり，そのとき f は引数の列を要素にもつようなリストを返す.

```
In[50]:= Clear[f]

In[51]:= f[x__] := {x}

In[52]:= f[a, b, c]
Out[52]= {a, b, c}
```

引数がないときには，定義は適用されず，そのままの値が返る.

```
In[53]:= f[]
Out[53]= f[]
```

次の定義にはトリプルブランクが用いられている．これにマッチするのは，関数 f が任意個数の弾きすをとっているときであり，そのとき f は引数の列を要素にもつようなリストを返す.

```
In[54]:= Clear[g]

In[55]:= g[x___] := {x}

In[57]:= g[a, b, c]
Out[57]= {a, b, c}
```

引数がないときにもこの定義が適用され，空リストが帰る.

```
In[56]:= g[]
Out[56]= {}
```

次の関数 h は，引数の個数を返すだけの関数である．引数がないときには 0 を返すために，引数がないときにもマッチするようにトリプルブランク ___ を用いている.

```
In[96]:= Clear[h]

In[97]:= h[x___] := Length[{x}]

In[98]:= h[a, b, c, d]
Out[98]= 4
```

```
In[110]:= h[]
Out[110]= 0
```

次の関数 mean は引数の平均値を返す．引数がないときには，平均値は求められないので，引数がないときにはマッチしないようにダブルブランク __ を用いている．

```
In[111]:= Clear[mean]

In[112]:= mean[x__] := Plus[x] / Length[{x}]

In[114]:= mean[4, 6, 5, 7, 2]
Out[114]= 24/5

In[115]:= mean[]
Out[115]= mean[]
```

練習問題：引数の相乗平均を返す関数 geomean を作れ．

次の定義には，二つのブランクと一つのトリプルブランクが用いられている．これにマッチするのは，関数 h の引数が長さ2以上のリストであるときであり，そのとき f はリストの最初の二つの要素をリストで括ったものを返す．

```
In[74]:= Clear[h]

In[75]:= h[{x_, y_, z___}] := {{x, y}, z}

In[76]:= h[{a, b, c, d, e}]
Out[76]= {{a, b}, c, d, e}
```

次の定義には，三つのトリプルブランクとその間に挟まった具体的な値 0 がある．これにマッチするのは，関数 f に引数が要素に 0 を 2個以上含んだリストであるときであり，そのとき f はリスト中の最初の 0 から 2 番目の 0 までの間を削除したリストを返す．

```
In[77]:= Clear[f]

In[82]:= f[{x___, 0, y___, 0, z___}] := {x, 0, 0, z}

In[83]:= f[{1, 2, 0, 3, 4, 5, 0, 7, 8, 0, 9}]
Out[83]= {1, 2, 0, 0, 7, 8, 0, 9}
```

ここで注意して欲しいことは，このパターンがマッチしているのは，最初の 0 と 2 番目の 0 とであり，3 番目の 0 にはマッチしていないことである．このように，パターンマッチングは，式の先頭（左側）から見て行き，初めてマッチした部分で止まる．

変換規則

- 変換規則 pattern -> rhs は，式 expr 中の部分で，パターン pattern にマッチするものを rhs に置き換えるという規則を表す．

たとえば， f[t_] -> g[t+1] という変換規則は， f[何か] というものを， g[何か+1] で置き換えるという規則を表す。

```
In[17]:= Cos[f[x]] + f[x]^2 + f[x + 1] /. f[t_] → g[t + 1]
Out[17]= Cos[g[1 + x]] + g[1 + x]^2 + g[2 + x]
```