第P章 Prolog 超簡易入門

Prologは、 論理型言語 と呼ばれるプログラミング言語の仲間のうち、もっとも代表的なものである。論理型言語では、「～ならば～」という論理式の集まりをプログラムとみなし、論理式の証明をプログラムの実行とみなす。

Prolog などの論理型言語に特徴的な概念としては、 ユニフィケーション （単一化）、 バックトラッキング （後戻り）、 論理変数 などが挙げられる。

P.1 Prolog でのプログラミング

Prolog では「～ならば～」という論理式の集まりをプログラムとみなすが、この「～」の部分を構成するのが、次のような形式である。

```
述語(項1,…, 項n)
```

このかたちを 素論理式 (アトム (Atom)、 複合項などと呼ぶこともある。)と呼ぶ。「項」は数・文字列などの定数や変数、あるいはリストなどのデータ構造などである。「述語」は直観的には項 $_1$,..., 項$_n$ の間の関係を表す。例えば、`father(adam, cain)` という素論理式は「Adam は Cain の父である」という関係を表す。

Prolog のプログラムは、 ホーン節 (Horn clause)と呼ばれる形式の集まりである。ホーン節とは、

```
素論理式0 :- 素論理式1, …, 素論理式n.
```

という形である。（最後に必ずピリオド（.）が必要である。）「:-」は左向きの矢印（←）と考えれば良い。つまり、これは素論理式$_1$, ..., 素論理式$_n$ がすべて成り立つならば、素論理式$_0$ も成り立つという規則を表している。

ホーン節の「:-」の左側には素論理式は一つのみであり、これを ヘッド (head)と呼ぶ、「:-」の右側の素論理式の並びは ボディー (body) と呼ぶ。

例えば、

```
1 grandchild(X, Z) :- child(X, Y), child(Y, Z).
```

は、`X` と `Y`, `Y` と `Z` の間に `child` という関係がある（`X` が `Y` の子であり、 `Y` が `Z` の子である）ならば、`X` と `Z` の間に `grandchild` という関係がある（`X` は `Z` の孫である）という規則を述べている。ここで `X`, `Y`, `Z` は変数である。Prolog の変数は`X`, `Xs`, `Y` のように、アルファベットの 大文字 （あるいはアンダースコア「_」）からはじまる識別子（名前）を使用する。一方、小文字からはじまる識別子は、述語や構成子などの定数に利用される (感覚的には Haskell と全く逆なので注意する。)。変数はどのような項に具体化しても良いので、変数を使用した規則は、実際には無数の規則を表していることになる。

ホーン節のうちボディーがないものを 事実 (fact) と言う。このときは、「:-」も省略する。（文末のピリオドは必要である。） 例えば、

```
child(hidetada, ieyasu).
```

は hidetada（秀忠）が ieyasu（家康）の子である、という事実を表明している。ここで hidetada, ieyasu は小文字からはじまるので定数である。

Prolog のプログラムは、このようなホーン節（事実を含む）を集めたものである。例えば徳川家の家系図を表すプログラムの一部は次のようになる。

```
grandchild(X, Z) :- child(X, Y), child(Y, Z).

child(hideyasu, ieyasu).
child(hidetada, ieyasu).
child(yoshinao, ieyasu).
child(yorinobu, ieyasu).
child(yorifusa, ieyasu).

child(iemitsu, hidetada).
child(tadanaga, hidetada).
child(masayuki, hidetada).

child(ietsuna, iemitsu).
child(tsunayoshi, iemitsu).
```

このようなプログラムをファイルに作成し、処理系にロード (Prolog では伝統的に consult という。)する。例えば Windows 上で動作する SWI-Prolog という処理系ではメニューの「File」—「Consult ...」 でプログラムファイルをロードすることができる。プログラムは、このようなホーン節の集まりに対して質問を発することにより起動される。Prolog の処理系は「?- 」というプロンプトを出力するので、このあとに質問を入力する。（質問も最後に必ずピリオドが必要である。）

```
?- grandchild(hidetada, ieyasu).

No
?- grandchild(iemitsu, ieyasu).

Yes
```

1 番目の質問は、Hidetada（秀忠）は Ieyasu（家康）の孫か? という質問である。これはプログラム中の規則から導出できないので、処理系は No と答えている。2 番目の質問は、Iemitsu（家光）は家康の孫か? という質問である。これは、child(hidetada, ieyasu). と child(iemitsu, hidetada). という二つの事実と

```
grandchild(X, Z) :- child(X, Y), child(Y, Z).
```

というホーン節から導出できるので Yes と答えている。

さらに Prolog では質問の中に変数を含めることができる。すると Prolog の処理系は、その質問を成り立たせる 変数への代入 を出力する。例えば、

```
?- grandchild(X, ieyasu).
```

という質問に対しては、

```
X = iemitsu
```

という解を出力する。ここで、リターンキーを押すとこれで終ってしまうが、「;」を入力すると、さらに別解を表示させることができる。

```
X = iemitsu ;

X = tadanaga ;

X = masayuki ;

No
```

質問には一般に複数の素論理式を並べることができる。

```
?-  素論理式1, …, 素論理式n.
```

このような形を ゴール節 と呼ぶ。直観的には、並べた素論理式がすべて成り立つか?（成り立たせるような変数への代入が存在するか?） という質問を表している。

P.2 単一化（ユニフィケーション）と後戻り

この節では、Prolog のプログラムの実行方法を解説する。解説を簡単にするために、まず、最初にいくつかの言葉を定義しておく。

2 つ以上の（通常変数を含む）素論理式があり、変数に適切な代入をして、これらの素論理式をまったく同一のものにすることができるとき、これらの素論理式は 単一化可能 (unifiable) であるという。また、その時の代入を 単一化代入 (unifier)という。例えば、次の 2 つの素論理式

```
p(a, Y, Z) と p(X, b, Z)
```

は `X = a, Y = b` を単一化代入として単一化可能である。単一化可能なときは、通常必要なのは最汎（最も一般的な）単一化代入 (most general unifier, MGU) である。例えば、上の例では、`X = a, Y = b, Z = c` という代入も単一化代入であるが、前者の方がより一般的である。実際、この例の場合は `X = a, Y = b` が MGU になる。

プログラム中のホーン節

```
P :- Q1,  Q2, …, Qn,
```

のヘッド `P` が素論理式 (`G`) と単一化可能なとき、このホーン節は `G` に 適用できる という。

Prolog のプログラムの実行方法は、次のようにまとめられる。

1. ゴール節中の もっとも左の 素論理式に対し、適用できるホーン節を選ぶ。適用できるホーン節が複数あるときは、プログラム中に 先に書かれている ホーン節から試みる (Prolog を定理証明系として見たときには、ここで、「最も左」、「先に書かれている」という選択をするために、証明系としての力が弱くなってしまう。つまり、このような制限をしなければ証明できるはずの論理式が証明できなくなってしまう。しかし、プログラミング言語として見たときは、効率を確保するために必要な制限である。)。

2. 選んだ適用可能なホーン節に対して、その MGU をゴール節の残りの素論理式に適用し、さらにゴール節中の最左素論理式を、上で選んだホーン節のボディに MGU を適用したものと置き換える。(ホーン節のボディーがないときは、最左素論理式を消す。)

3. もし適用できるホーン節がなければ、後戻り(バックトラック)して、ひとつの前のゴール節中の素論理式がある状態からやり直す。そして、その素論理式に適用できるホーン節のうち次の候補を選ぶ。

4. ゴール節に素論理式がなくなれば、プログラムの実行を終了し、その時得られた変数への代入を表示する。素論理式がまだあれば、1. に戻る。

具体的に、

```
?- grandchild(X, ieyasu).
```

というゴール節での動作を説明することにする。まずこのゴール節の最左(1 つしかないが)素論理式は `grandchild(X, ieyasu)` である。これに適用できるホーン節は、今のプログラムでは、

```
grandchild(X, Z) :- child(X, Y), child(Y, Z).
```

しかなく、最汎単一化代入は `Z = ieyasu` である。するとゴール節は

```
                                                … ①
```

に変換される。

次に、この最左の `child(X, Y)` に適用できる最初のホーン節は、

```
child(hideyasu, ieyasu).
```

であり、MGU は `X = hideyasu, Y = ieyasu` である。すると、ゴール節は、

```

```

に変換される。しかし、このゴール節に適用できるホーン節はない。

そこで、上の ① のところまでバックトラックし、次の候補である

```
child(hidetada, ieyasu).
```

の適用を試みる。しばらくのあいだ同様にバックトラックが続き、最終的に①のゴール節に対し、

```
child(iemitsu, hidetada).
```

というホーン節を選ぶ。そのとき MGU は `X = iemitsu, Y = hidetada` となる。すると、ゴール節は、

```
1
```

に書き換えられる。これはすぐに適用できるホーン節が見つかり、ゴール節が空になって、結果として代入:

```
1 X = iemitsu
```

が出力される。ここで「;」が入力されると、またバックトラックが起こる。

ホーン節の適用は命令型言語・関数型言語の関数呼出しのようなものだと考えることもできるが、単一化により呼び出される側（ホーン節のヘッド）から呼び出す側（ゴール節）に情報が流れることがある、というところが論理型言語に特徴的なところである。例えば、

```
1 ?- child(X, ieyasu).
```

という呼出しに対しては、`X = hideyasu`（**あるいは** `hidetada, yoshinao` …）という情報が、呼び出された側から、呼び出した側に伝えられる。

Prolog の変数は命令型言語の変数と異なり、いったん単一化により値が代入されれば、以降は値を変更されることはない。ただし、（純）関数型言語の変数とも異なり、最初はいったん不定 (unknown) な状態を取ることができる。このような振舞いをする変数は一般に 論理変数 (logical variable) と呼ばれる。

P.3 Prologでのリスト処理

Prolog でのリストの記法は要素を「,」で区切り、角括弧（`[` と `]`）で囲んで、`[1, 2, 3]` のように書く。

また、先頭の要素が `X` で、先頭を除く残りの要素からなるリストが `Xs` であるようなリストは `[X|Xs]` と表記される。これは Haskell の `x:xs` という書き方に相当する。また、`[1,2,3]` は `[1|[2|[3|[]]]]` の略記法である。

Prolog でのリストを連接 (append) するプログラムは次のように記述できる。

```
1 myappend([], Y, Y).
2 myappend([H|X], Y, [H|Z]) :- myappend(X, Y, Z).
```

これは、1 番目の引数と 2 番目の引数を連接した結果が 3 番目の引数になる、という関係を表している。

例えば, `[1,2]` と `[3,4]` の連接は次のように求められる。

```
?- myappend([1,2], [3,4], Z).

Z = [1,2,3,4] ;

No
```

問 P.3.1 実際に、上のような結果が出ることを、上に示した Prolog の実行方法で1ステップずつ確認せよ。

?- myappend([1,2], [3,4], Z_0).

これに myappend([H_1|X_1], Y_1, [H_1|Z_1]) :- myappend(X_1, Y_1, Z_1). が

H_1=1, X_1=[2], Y_1=[3,4], Z_0=[1|Z_1] で適用できて

?- myappend([2], [3,4], Z_1).

これに myappend([H_2|X_2], Y_2, [H_2|Z_2]) :- myappend(X_2, Y_2, Z_2). が

H_2=2, X_2=[], Y_2=[3,4], Z_1=[2|Z_2] で適用できて

?- myappend([], [3,4], Z_2).

これに myappend([], Y_3, Y_3). が

Y_3=[3,4], Z_2=[3,4] で適用できてゴール節が消える

まとめると Z_0 = [1|[2|[3,4]]] (= *[1,2,3,4]*)

上の問を解いてみるとわかるが、変数 Z_0 は後ろの要素が不定（変数）のまま、前の要素から順に定まっていく。

後ろの要素がもっと後で定まるようなプログラムを書くことも可能である。例えば、

```
myconcat([], []).                                  % ①
myconcat([Xs|Xss], Ys)
   :- myappend(Xs, Zs, Ys), myconcat(Xss, Zs). % ②
```

これは、リストのリストを平坦なリストに変換するプログラムである。

```
?- myconcat([[1, 2], [3, 4, 5], [6, 7]], Ys).

Ys = [1, 2, 3, 4, 5, 6, 7] ;

No
```

問 P.3.2 実際に、上のような結果が出ることを、上に示した Prolog の実行方法で1ステップずつ確認せよ。

?- myconcat([[1, 2], [3, 4, 5], [6, 7]], Xs).

これに ② が Xs_1=[1,2], Xss_1=[[3,4,5],[6,7]], Ys_1=Xs で適用できて

?- myappend([1,2],Zs_1,Xs), myconcat([[3,4,5],[6,7]],Zs_1).

?- myappend([2],Zs$_1$,Z$_1$), myconcat([[3,4,5],[6,7]],Zs$_1$).

ただし Xs=[1|Z$_1$]

?- myappend([],Zs$_1$,Z$_2$), myconcat([[3,4,5],[6,7]],Zs$_1$).

ただし Z$_1$=[2|Z$_2$]

?- myconcat([[3,4,5],[6,7]],Zs$_1$). ただし Z$_2$=Zs$_1$

まとめると Xs=[1|[2|Zs$_1$]]

?- myconcat([[6,7]],Zs$_2$). ただし Zs$_1$=[3|[4|[5|Zs$_2$]]]

⋮

?- myconcat([],Zs$_3$). ただし Zs$_2$=[6|[7|Zs$_3$]]

これに対して ① が Zs$_3$=[] で適用できゴール節が消える

まとめると Xs=[1,2,3,4,5,6,7]

この `myappend` の第 2 引数の `Zs` はあとの `myconcat` の呼出しで値が定まる。また `myappend` の第 3 引数の `Ys` はしばらくの間、最後が論理変数で終る形で扱われることになる。

このように尾 (tail) の部分が論理変数になっているリストを 開リスト (open list) という。開リストは Prolog でリストを扱う時に良く使われるイディオムである。

さらに Prolog のおもしろいところは `myappend` の逆向きの計算もできるということである。

```
?- myappend(X, Y, [1, 2, 3]).

X = []
Y = [1, 2, 3] ;

X = [1]
Y = [2, 3] ;

X = [1, 2]
Y = [3] ;

X = [1, 2, 3]
Y = [] ;

No
```

この例では連接して `[1, 2, 3]` になる 2 つのリストの、すべての可能性を求めていることになる。ユーザーが「;」を入力するたびにバックトラックが起こり別解を表示する。

問 P.3.3 逆向きの `myappend` の計算ができることを、前節で示した Prolog の実行方法で実際に 1 ステップずつ確認せよ。