計算機プログラミング'93　＃10

久野　靖*

1993.11.15

1　パターンマッチング

1.1　パターンとパターン変数

前回の推論システムでは、

```
(penguin-rule (if (animal-x is bird) (animal-x can swim))
              (then (animal-x is penguin)))
```

という風に、特定の個体 (animal-x) についての規則だけしか持っていなかった。しかし、もし動物が 3 匹いたとして、同様の規則を animal-y 用にも animal-z 用にも準備するのではあまりにも大変なことはお分かりだろう。

そこで、今回はパターン機能を入れることでこれらの規則をまとめて 1 個で済ますことを可能にしてみる。つまり、上のような個別の規則の代わりに

```
(penguin-rule (if (?x is bird) (?x can swim))
              (then (?x is penguin)))
```

のように書けるようにするわけである。そして、?x のところに animal-x、animal-y、…のいずれかをあてはめることにより、多数の個体について同じ規則を活用することができる。この (?x is bird) のようなものは、「特定の規則というよりこのような形をしたもの全般を表す」ので「パターン」と呼ばれる。そして、?x の部分は具体的に「さまざまなものが当てはまることを許す部分」を意味するのに使われるので「パターン変数」と呼ばれる。

これを使えば、例えば「子供の子供は孫」という規則を次のように書くことができる。

```
(mago-rule (if (?x is a child-of ?y) (?y is a child-of ?z))
           (then (?x is a grandchild-of ?z)))
```

パターンという考え方がとても協力なことがお分かりいただけると思う。

1.2　パターン変数の判定

実は Lisp の世界では?x のようなものも普通の記号 (symbol) である。ただ、以下では「先頭が?で始まる記号はパターン変数として使う」という約束を設けるというだけである。

記号の先頭が?であることを調べる関数は次のようになる。

```
(defun varp (x)
   (and (symbolp x) (char= (char (symbol-name x) 0) #\?)))
```

*筑波大学大学院経営システム科学専攻

これを説明すると、「x が記号であり、そしてその記号の名前文字列 (symbol-name) の、先頭の文字 ((char 文字列 0)) が、文字としての?(#\?) に等しい」ことを調べているわけである。これを覚える必要はないのでして、適宜打ち込んで使って頂く。

1.3 パターンマッチを行なう関数

以下では一般のパターンマッチを行なうための関数群を書くことを練習問題としてやって頂く。

練習 37 2 つの引数 pat、exp を取り、exp が pat にマッチするかどうかを判定する関数 match1 を書け。ただし pat はアトム (記号か数) であるものとする。

練習 38 2 つの引数 pat、exp を取り、exp が pat にマッチするかどうかを判定する関数 match を書け。

1.4 連想リストによるマッチの記録

ここまでだと、マッチするかどうかは分かるけれど、具体的にパターン変数が何にマッチしたのかが分からない。これではつまらないので、マッチに成功した時はただの t の代わりに変数にマッチしたものを連想リストの形で返すことにする。例えば次のようになるはずである。

```
>(match '(?x is a ?y) '(this is a pen) '((nil)))
((?X . THIS) (?Y . PEN) (NIL))
```

なお、3 番目の引数は「これまでにマッチした部分に対する連想リスト」を意味している。なぜこれが必要かはすぐに分かる。まずこれを使って、「これまでに連想リストに登録されている変数の場合は登録された値と等しいかどうかを調べ、そうでない場合には連想リストに新しい値を登録して返す」という関数 matchv を示す。

```
(defun matchv (var exp alist &aux a)
  (setq a (assoc var alist))
  (cond ((null a) (cons (cons var exp) alist))
        ((equal (cdr a) exp) alist)
        (t nil)))
```

練習 39 match1 を、3 番目の引数として連想リストを渡し、なおかつ変数に対する処理は matchv を呼ぶように変更せよ。例えば次のように動作するようになるはずである。

```
>(match1 '?x 'abc '((?x . abc) (nil)))
((?X . ABC) (NIL))
>(match1 '?x 'def '((?x . abc) (nil)))
NIL
>(match1 '?x 'def '((nil)))
((?X . DEF) (NIL))
>(match1 'def 'def '((nil)))
((NIL))
```

練習 40 match を、3 番目の引数として連想リストを渡すように変更せよ。

1.5 パターンの展開を行なう関数

ここまではパターンマッチを行なう一方だったが、パターン変数には別の機能、すなわちパターンと連想リストからパターンを展開したリストを作り出す機能も持たせられる。例えば次のような具合である。

```
>(expand '(?x is a ?y) '((?x . this) (?y . pen)))
(THIS IS A PEN)
```

これを行なうための関数群も練習問題にしておく。

練習 41 変数 var と連想リスト alist を受け取り、変数に対応する連想リスト中の値を返す関数 expandv を作れ。例えば (expandv '?x '((?x . baka))) → BAKA) となる。

練習 42 アトム pat と連想リスト alist を受け取り、それが変数だったら連想リスト中の値で置き換えるが、変数でなければ元のまま返す関数 expand1 を作れ。例えば (expand1 '?x '((?x . baka))) → BAKA、(expand1 'abc '((?x . baka))) → ABC となる。

練習 43 一般のパターン pat と連想リスト alist を受け取り、パターン中の変数を連想リスト中の値で置き換えた S 式を返す関数 expand を作れ。

2 推論へのパターンの導入

2.1 パターンによる問い合わせ

ここまでで十分パターンになじんだと思うので、前回やったような事実に対してパターンによる検索を行なってみよう。事実としては、今回はまた親子関係を使ってみる。

```
(setq *facts*
   '((onna mary)
     (fuufu bob mary)
     (jane kodomo-of mary)
     (jack kodomo-of bob)
     (otoko jack)
     (suzanne kodomo-of jack)
     (alice kodomo-of jack)))
```

なお、このデータは/ua/kuno/work/pat.lsp に (後で出てくる推論規則と一緒に) 入れてある。

このような事実の中から、パターンを使って問い合わせを行なえるようにしたら役に立つと思わないか? 例えば次のように。

```
>(recall+ '(?x kodomo-of jack) '((nil)))
(((?X . SUZANNE) (NIL)) ((?X . ALICE) (NIL)))
```

一般に jack の子供は複数いるかも知れないから、結果は連想リストのリストになるべきである。

練習 44 上に述べたような関数 recall+を書け。

さて、実はやりたいことは 1 つの事実だけというより、複数の事実にまたがった問い合わせを書くことである。これは、次のようにして連想リストのリストを保持しながら順次 recall+を呼んでいくことで行なえる。

```
(defun recall-list (patlist &aux alset r)
  (setq alset '(((nil))))
  (dolist (pat patlist)
          (setq alset (mapcan #'(lambda (a) (recall+ pat a)) alset))
          (if (null alset) (return-from recall-list nil)))
  alset)

>(recall-list '((?x kodomo-of ?y) (otoko ?x)))
(((?Y . BOB) (?X . JACK) (NIL)) ((?Y . JANE) (?X . TOM) (NIL)))
```

2.2 前向き推論への適用

実は、上の recall-list はほとんど、前回やった推論規則の then 部をパターンつきで処理する機能に相当している。そこで前回の前向き推論の関数群のうち testif、usethen、tryrule を次のように直す。

```
(defun testif (rule &aux alset)
   (recall-list (cdadr rule)))

(defun usethen (rule alset &aux result efact)
   (setq result nil)
   (dolist (fact (cdaddr rule))
           (dolist (alist alset)
                   (setq efact (expand fact alist))
                   (cond ((remember efact)
                          (format t "Rule ~A Deduces ~A~%" (car rule) efact)
                          (setq result t)))))
   result)

(defun tryrule (rule &aux alset)
   (if (setq alset (testif rule)) (usethen rule alset)))
```

あとの関数は前回と同じである。これで、例えば

```
(setq *rules*
   '((fuufu-rule-1 (if (fuufu ?x ?y))
                   (then (fuufu ?y ?x)))
     (fuufu-rule-2 (if (fuufu ?x ?y) (otoko ?x))
                   (then (onna ?y)))
     (fuufu-rule-3 (if (fuufu ?x ?y) (onna ?x))
                   (then (otoko ?y)))
     (kyoudai-rule-1 (if (?x kodomo-of ?y) (?z kodomo-of ?y))
                     (then (kyoudai ?x ?z)))
     (kyoudai-rule-2 (if (kyoudai ?x ?y))
                     (then (kyoudai ?y ?x)))))
```

のような規則を与えて推論させると、次のような事実が推論できる。

```
>(deduce)
Rule FUUFU-RULE-1 Deduces (FUUFU MARY BOB)
Rule FUUFU-RULE-2 Deduces (ONNA MARY)
Rule KYOUDAI-RULE-1 Deduces (KYOUDAI ALICE ALICE)
```

```
Rule KYOUDAI-RULE-1 Deduces (KYOUDAI ALICE SUZANNE)
Rule KYOUDAI-RULE-1 Deduces (KYOUDAI SUZANNE ALICE)
Rule KYOUDAI-RULE-1 Deduces (KYOUDAI SUZANNE SUZANNE)
Rule KYOUDAI-RULE-1 Deduces (KYOUDAI JACK JACK)
Rule KYOUDAI-RULE-1 Deduces (KYOUDAI JANE JANE)
T
```

というわけで、前回よりかなり強力なシステムになったことはお分かり頂けると思います。

> どうも、長らくおつき合い頂いてご苦労さまでした。私の方から見ると、プログラミングになじみのある方は Lisp 固有の考え方や記法になじむのに苦しみ、プログラミングになじみのない方はその点はいいのだけれどプログラミングそのものに慣れていないので苦しんだという感じがします。ともあれ、最後にアンケートをつけておきますので、記入して今週中くらいに私のメールボックスに出して下さい。(出席点をあげます。)

小課題その 7 — パターンつき前向き推論の応用

7-A. 上の推論システムについて、もっと多くの規則を追加してみよ。例えば「祖父母」「孫」「いとこ」「父親」「母親」「兄弟のうちの年上、年下」「兄」「姉」「妹」「弟」など (全部でなくてもよい!) を推論する規則を入れ、これらが正しく動くことを例題を用意して確かめる。

7-B. 前回の問題とおなじく、もっと別の推論データと規則を使ってパターンつき推論システムを構成してみよ。

7-C. 上のシステムでは「自分と自分は兄弟」になってしまう。こういうのを排除するには推論システムにどういう機能があればよいか考え、実際に組み込んでみよ。

結果をレポートとして提出する場合には、必ず A4 版の紙を使用し、次のような順で構成し、全体を綴じて提出すること。

- 表紙。課題名 (1993 年度計算機プログラミング課題# 6)、学籍番号、氏名、提出日付) のみを記すこと。
- 方針。どのような考え方で課題を解こうと思ったか記す。
- 回答。作成したプログラムとその解説、実行例など。
- 考察。課題をやった結果どんなことがわかったか、何が問題か、など。(感想も入れてほしいが、感想ばかりでは内容として不足。)

結局、成績報告の都合から〆切は一応 12 月 6 日厳守ということにさせて頂きます。小課題の提出は義務ではありません。

今回の関数群のリスト

```
(defun varp (x)
  (and (symbolp x) (char= (char (symbol-name x) 0) #\?)))

;(defun match1 (pat exp)
;  (cond ((varp pat) t)
;        (t (equal pat exp))))
```

```
;(defun match (pat exp)
;  (cond ((atom pat) (match1 pat exp))
;        ((atom exp) nil)
;        (t (and (match (car pat) (car exp))
;                (match (cdr pat) (cdr exp))))))

(defun matchv (var exp alist &aux a)
  (setq a (assoc var alist))
  (cond ((null a) (cons (cons var exp) alist))
        ((equal (cdr a) exp) alist)
        (t nil)))

(defun match1 (pat exp alist)
  (cond ((varp pat) (matchv pat exp alist))
        ((equal pat exp) alist)
        (t nil)))

(defun match (pat exp alist &aux a)
  (cond ((atom pat) (match1 pat exp alist))
        ((atom exp) nil)
        (t (setq a (match (car pat) (car exp) alist))
           (if a (match (cdr pat) (cdr exp) a) nil))))

(defun expandv (var alist &aux a)
  (setq a (assoc var alist))
  (cond ((null a) nil)
        (t (cdr a))))

(defun expand1 (pat alist)
  (cond ((varp pat) (expandv pat alist))
        (t pat)))

(defun expand (pat alist &aux a)
  (cond ((atom pat) (expand1 pat alist))
        (t (cons (expand (car pat) alist)
                 (expand (cdr pat) alist)))))

(defun remember (new)
   (dolist (old *facts*)
           (if (equal old new) (return-from remember nil)))
   (push new *facts*)
   t)

(defun recall+ (pat alist &aux a r)
  (dolist (f *facts*)
          (if (setq a (match pat f alist)) (push a r)))
  r)

(defun recall-list (patlist &aux alset r)
  (setq alset '(((nil))))
  (dolist (pat patlist)
          (setq alset (mapcan #'(lambda (a) (recall+ pat a)) alset))
          (if (null alset) (return-from recall-list nil)))
  alset)

(defun testif (rule &aux alset)
   (recall-list (cdadr rule)))

(defun usethen (rule alset &aux result efact)
```

```
   (setq result nil)
   (dolist (fact (cdaddr rule))
           (dolist (alist alset)
                   (setq efact (expand fact alist))
                   (cond ((remember efact)
                          (format t "Rule ~A Deduces ~A~%" (car rule) efact)
                          (setq result t)))))
   result)

(defun tryrule (rule &aux alset)
   (if (setq alset (testif rule)) (usethen rule alset)))

(defun stepforward ()
   (dolist (rule *rules*)
           (if (tryrule rule) (return-from stepforward t)))
   nil)

(defun deduce (&aux progress)
   (setq progress nil)
   (loop (if (stepforward)
             (setq progress t)
             (return-from deduce progress))))
```

計算機プログラミング'93 終了時アンケート

学籍番号:_____ 御芳名:_____

全体的な評価について選択してください

質問	はい				いいえ
科目の内容は自分のレベルにおおむね合ってましたか?	a	b	c	d	e
講義は興味深かったですか?	a	b	c	d	e
演習は身につきましたか?	a	b	c	d	e
小課題は難しすぎたですか?	a	b	c	d	e
負担は大きかったですか?	a	b	c	d	e
全体として満足しましたか?	a	b	c	d	e

以下の内容についてコメントをお願いします

講義・演習の進め方について:

Lisp 言語という内容について:

小課題や演習の内容について:

その他、怨みつらみでも何でもコメントをどうぞ: