計算機プログラミング'93　＃9

久野　靖*

1993.11.10

前置き — 書式付出力

Lisp でもプログラムが実用に近付いてくると色々な出力がしたくなるが、それを全部 print、princ、terpri などでやるのは結構大変である。そこで、C の printf のようなものが CommonLisp にも用意されている。それは format という関数である。

```
(format t "書式文字列" 式 …)
```

これは、printf と同様基本的に「書式文字列」を打ち出すが、その中に特別な形のものがあると次のような動作をする。

```
~A  :  対応する「式」を princ のようにして打ち出す。
~%  :  改行する。
```

例えばこれを使って「引数として渡された S 式を表示し、これは正しいかどうか問い合わせる」関数を示しておく。これは後で利用する。

```
(defun query (fact)
   (format t "Fact ``~A'' is correct? " fact)
   (cond ((member (read) '(t y yes))
          (format t " your answer was YES.~%") t)
         (t
          (format t " your answer was NO.~%") nil)))
```

1　推論

1.1　デモ — 動物あてシステム

まずは今回やるプログラムがどんなものかを見て頂く。最初に、以下の動物のうち 1 つを頭の中で思い浮かべて頂きたい。

- 虎
- ペンギン
- ダチョウ
- シマウマ
- キリン
- ペリカン

KCL にファイルをロードし、関数 diagnose を引数無しで起動する。すると次々に質問が出てくるので、思い浮かべた動物と対照して yes/no で答える。するとどの動物か的確に (?) 当ててくれる。どうですか?

*筑波大学大学院経営システム科学専攻

1.2 if-then 規則

さて、実はこのプログラムが利用している知識というのは次のような形で表現されている。

```
(setq *rules*
   '((kegawa-rule (if (animal-x has fur))
                  (then (aimal-x is honyuu-rui)))
     (milk-rule (if (animal-x provides milk))
                (then (animal-x is honyuu-rui)))
     (hane-rule (if (animal-x has feathers))
                (then (animal-x is bird)))
     (fly-egg-rule (if (animal-x can fly) (animal-x lay eggs))
                   (then (animail-x is bird)))
     (meat-rule (if (animal-x eat meats))
                (then (animal-x is kemono)))
     (teeth-rule (if (animal-x has sharp teeth) (animal-x has sharp nail))
                 (then (animal-x is kemono)))
     (hidume-rule (if (animal-x is honyuurui) (animal-x has hidume))
                  (then (animal-x is yuutei-rui)))
     (zebra-rule (if (animal-x is yuutei-rui) (animal-x has stripes))
                 (then (animal-x is zebra)))
     (tiger-rule (if (animal-x is kemono) (animal-x has stripes))
                 (then (animal-x is tiger)))
     (giraff-rule (if (animal-x is yuutei-rui) (animal-x has long necks))
                  (then (animal-x is giraff)))
     (ostrich-rule (if (animal-x is bird) (animal-x has long necks)
                       (animal-x cannot swim))
                   (then (animal-x is ostrich)))
     (penguin-rule (if (animal-x is bird) (animal-x has no long necks)
                       (animal-x can swim))
                   (then (animal-x is penguin)))
     (pelican-rule (if (animal-x is bird) (animal-x has long necks)
                       (animal-x can swim))
                   (then (animal-x is pelican)))))
```

つまり、「知識」はいくつかの「推論規則」で表されている。その規則をまとめてリストにしたものを、変数`*rules*`に入れておくことにする。各「推論規則」は

```
(規則名 (if 事項 ... 事項) (then 事項 ... 事項))
```

の形をしている。その意味は、「if の後にある事項が全部成立していれば、その時は then の後にある事項も全部成り立つ」ということである。

1.3 事項の記憶と検索

ところで、先の if-then 規則に現れる「事項」というのは、要するに「互いに区別できる、何らかの事柄」であって、それに「どういう意味があるか」は以下のプログラムでは立ち入らない。解釈は人間がするもんである。計算機の方は「現在どういう事項が成り立つと分かっているか」のみを扱う。で、その「成り立つと分かっているもの」を別の変数`*facts*`に入れておくことにしよう。

```
(setq *facts*
   '((animal-x has sharp teeth)
     (animal-x has stripes)
     (animal-x has fur)
     (animal-x eat meats)))
```

ではそろそろ関数定義に行く。まず「こういう事項は成り立つと分かっているかどうか」を調べる関数と、「こういう事項が成り立つことを新たに登録する」関数が必要である。

練習 26 1 つの事項を引数として与えると、それが既に知られていることなら t、知られていないことなら nil を返す関数 recall を作れ。

練習 27 1 つの事項を引数として与えると、それが既に知られていることなら nil を返し、知られていないことなら*facts* に追加して t を返す関数 remember を作れ。

なお、上掲の*rules*と*facts*は/ua/kuno/work/suiron.lsp に入れてあります。

1.4 if-then 規則の解釈

さて、以上の準備のもとに、規則の解釈を行なう関数を定義する。まず、if 部の全部の事項が成立しているかどうか調べる関数を考える。

練習 28 1 つの規則を引数として与えると、その if 部のすべての事項が recall できれば t、1 つでも recall できなければ nil を返す関数 testif を作れ。

練習 29 1 つの規則を引数として与えると、その then 部のすべての事項を順に remember させ、もし 1 つでも新しく判った事項が含まれていれば t、そうでなければ nil を返す関数 usethen を作れ。

さて、ある規則を「処理する」ということは、if 部を検査して、すべて成立なら then 部を処理するということだ。

```
(defun tryrule (rule)
  (if (testif rule) (usethen rule)))
```

次に、何か 1 つ新しい事項が成立するまで処理する関数を示す。これも新しい事項が 1 つも成立させららなければ nil を返す。

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

これを、これ以上新しい事項が成立しなくなるまで繰り返すと、与えられた規則群と与えられた事項のもとで成立するすべての事項を計算する (つまり推論する) ことができる。

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

練習 30 以上のすべての関数を揃えて、関数 deduce から実行させてみよ。また、最初に判っている事項 (*facts*の初期値) を変更して推論を実行させてみよ。

なお、ここまでのプログラムだとまだ最初のデモとは違う。どう違うか考えてみること。

1.5 前向き推論と後向き推論

ところで、上でやった推論は「... であり、... である。故に... である」、つまり演繹をおこなう。言い替えれば、既に知っている事柄をもとに、それから直接に導ける事柄をちょっとずつ増やしていく。こういうのを、「前向き推論」と呼ぶ。

しかし、「この動物について知っていることがらを全部言え」というのはあんまり使いやすくない。そこで、もう 1 つのやり方である帰納法— 「後向き推論」もやってみよう。これは前とは違って、次のように進む。

- まず、「この動物は... ではないだろうか」という仮説を立てる。
- その仮説が成立するかどうか、証拠集めをする。
- その証拠そのものの是非がまだ未知なら、その証拠を再び仮説だと思って同様に証拠集めをする。

そこで、証拠集めの一環として人間に「これは正しいか間違いか?」と問い合わせるようにすると、結構それらしいシステムができる。

1.6 診断型エキスパートシステム

ではさっそく、後向き推論のプログラムを見ていただく。まず、仮説としてすべての考えられる動物のリストが必要である。

```
(setq *hypotheses*
   '((animal-x is tiger)
     (animal-x is zebra)
     (animal-x is giraff)
     (animal-x is ostrich)
     (animal-x is penguin)
     (animal-x is pelican)))
```

今度はメインプログラムの関数から見ていこう。このプログラムは知られている事項は今度はからっぽにしてから始める。そしてまた、同じことを何回も聞かれるのは人間にとって嬉しくないので、1 回聞かれたことは *asked*という変数に保存しておくが、これも最初はからっぽである。そして、1 つずつ仮説を取り上げてはそれを verify しようと試みる。

```
(defun diagnose ()
   (setq *facts* nil)
   (setq *asked* nil)
   (dolist (h *hypotheses*)
           (cond
            ((verify h)
             (format t "Hypothesis ~A is True.~%" h)
             (return-from diagnose t))))
   (format t "No hypotheses can be confirmed.~%")
   nil)
```

なお、remember/recall は前と同じである。次に verify を示そう。まずその事項が既に知られていれば recall できれば)OK。そうでなければこの事項を成立させるのに役立つ規則のリストを求め (これはすぐ後で出てくる)、それを前向き、後向きの両方で適用してみる。成功すれば OK。どちらでも成功しなければ、人間に聞いたかどうか調べる。聞いたのならもう 1 回聞くのは無駄で、この verify は失敗。聞いてい変えれば、聞いてみて「はい」だったらそれを remember する。「いいえ」だったら失敗するが、ただし mark-asked を呼んで聞いたことを覚えておく。

```
  (defun verify (fact &aux relv)
     (if (recall fact) (return-from verify t))
     (cond
      ((setq relv (inthen fact))
       (dolist (x relv) (if (tryrule x) (return-from verify t)))
       (dolist (x relv) (if (tryrule+ x) (return-from verify t)))
       nil)
      ((asked fact) nil)
      ((query fact) (remember fact) t)
      (t (mark-asked fact) nil)))
```

なお、tryrule+は tryrule の testif の代わりに testif+を使うだけである。

```
(defun tryrule+ (rule)
  (and (testif+ rule) (usethen rule)))
```

では残っている関数は練習問題にしよう。

練習 31 1つの規則を引数として与えると、その if 部のすべての事項を順に verify によって調べ、すべてが OK であればまれていれば t、そうでなければ nil を返す関数 testif+を作れ。

練習 32 1つの事項と1つの規則を引数として与えると、その事項が規則の if 部に現れるなら t、そうでなければ nil を返す関数 thenp を作れ。

練習 33 1つの事項を引数として与えると、その事項が if 部に現れるようなすべての規則をリストとして返す関数 inthen を作れ。

練習 34 1つの事項を引数として与えると、その事項を変数*asked*に入っているリストに追加する関数 mark-asked を作れ。

練習 35 1つの事項を引数として与えると、その事項が変数*asked*に入っているリストに含まれているなら t、そうでなければ nil を返す関数 asked を作れ。

練習 36 これですべての関数がそろったので、これらを集めて diagnose から呼び出して動かせ。

2 小課題その6 — 推論データの構成

今回はボーナス問題として、Lisp のプログラムは全然直さなくても可能なのを出しておきましょう。A、B のうち好きな方をやってください。

6-A. 推論プログラムについて、プログラムは変更しないでいいから規則と事項を変更してみよ。例えば

1. 材料と調理法をもとに、夕ご飯のメインディッシュを推論する。
2. 逆に、味や見え方などから、料理の名前を推論する。
3. 経由駅や乗り換えの情報をもとに、大塚からどこかへ移動する時の移動先を推論する。
4. 自動車のエンジンがかからないとき、その原因を推論する。

その他面白そうなのをお待ちしてます。推論は前向きでも後向きでも好きな方で構いません。両方やって比較を考察すればなおよい。

6-B. さらに、推論プログラムも変更して次のような機能を入れてみよ。(これはけっこう好きな人向けです。)

1. tryrule/tryrule+を修正して、使用された規則のリストを変数*ruleused*の値として保持するようにせよ。
2. *ruleused*の値を利用して、「いかにして... を推論したか」を説明してくれる関数 how を定義せよ。もしその事項が推論されたのでなく与えられたのなら、how はそう答えるべきである。
3. なぜ「... という事実が必要になったか」という問いに答える関数 why を定義せよ。why に事項を与えると、why はその事項に依存する事項をプリントする。もしその事項が仮説ならば why はそのように答えるべきである。

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。)

〆切は一応 11 月末日までとしますが、遅れても受理はします。なお、成績報告期限の関係があるので、あまり遅くならないでください。その辺は別途掲示します。小課題の提出は義務ではありません。

資料で練習問題になっている関数の回答例

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

  (defun recall (fact)
     (dolist (old *facts*)
             (if (equal old fact) (return-from recall t)))
     nil)

  (defun testif (rule)
     (dolist (fact (cdadr rule))
             (if (not (recall fact)) (return-from testif nil)))
     t)

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

(defun testif+ (rule)
   (dolist (fact (cdadr rule))
           (if (not (verify fact)) (return-from testif+ nil)))
   t)

  (defun inthen (fact &aux result)
     (setq result nil)
     (dolist (r *rules*)
             (if (thenp fact r) (push r result)))
     result)

  (defun thenp (fact rule)
     (dolist (x (cdaddr rule))
             (if (equal x fact) (return-from thenp t)))
     nil)

(defun asked (fact)
   (dolist (x *asked*)
           (if (equal x fact) (return-from asked t)))
   nil)

(defun mark-asked (fact)
   (push fact *asked*))
```