プログラミング言語論2014 # 5 —
AIプログラミングとLisp

久野　靖 *

2014.5.15

今回は「ピーター・ノーヴィグ著, 杉本宣男訳, 実用 Common Lisp — AI プログラミングのケーススタディ —, 翔泳社, 2010 [1]」をネタ本に、AI プログラミングの初歩的なものとその考え方を紹介します。ただしその前に、まず「今日的な」Lisp の書き方の話を少し紹介します。それが無いと、出て来るコードが読めないと思いますので。

1　今日的なLispの書き方

1.1　eq、eql、equal

前回のスタイルでは、Lisp をできるだけ根源的な機能 (関数や記号) のみで理解する、という形だったので、必ずしも今日的でないところがあります。今回はそのような側面を補足してから本題に入ることにしましょう。まず、「等しい」ことを表す述語として前回は eq を取り上げました。

- eq — 2 つの引数がともに同じ記号のとき真。

実は、数値が等しい、文字列が等しい、などを調べるのは eq ではできません。次のものを使います。

- eql — eq に加え、2 つの数値が等しいこと、2 つの文字が等しいことも調べられる。
- string= — 2 つの文字列が等しいことを調べる。

そして最後に、アトムを含んだリスト構造について等しいことを調べるものは equal です (これは古くからあります)。

- equal — 2 つの値 (リスト等) が等しいことを調べる。

1.2　マクロ

前回は関数のみ使いましたが、今日の Lisp では**マクロ**と呼ばれる機能が大きな役割を果たしています。

たとえば、if が無かったものとして、cond から if を構築することを考えます。次の関数はどうでしょう?

```
(defun myif (condition then else)
  (cond (condition then)
        (t else)))
```

一見、うまく行きそうに見えます。

*経営システム科学専攻

[1] この本の原著は 1992 年で決して新しくはありませんが、AI プログラミングの様々な事例について一通り書かれているという点でとてもためになります。ただし 896p とぶ厚い本なので、紹介できるのはほんのさわりだけ。

```
(setq x 10)
=> 10
(myif (> x 5) 'large 'small)
=> LARGE
```

しかし次のはどうでしょうか。

```
(myif (> x 5) (print 'large) (print 'small))
LARGE
SMALL
=> LARGE
```

つまり、関数に渡すものは渡す前に評価 (実行) されてしまうので、myif に渡すものは両方とも実行されてしまうのです。これでは if の役を果たしません。

そこでマクロの出番です。マクロとは defmacro によって定義される関数「のようなもの」であり、直接計算する代わりに「その計算を行うような S 式」を返します。そして、返された S 式が評価 (実行) されます。

```
(defmacro myif (condition then else)
  (list 'cond (list condition then) (list 't else)))
```

マクロがどのように展開されるかは、macroexpand-1 という関数を使えば分かります。

```
(macroexpand-1 '(myif (> x 5) (print 'large) (print 'small)))
=> (COND ((> X 5) (PRINT 'LARGE)) (T (PRINT 'SMALL)))
```

cond 式が組み立てられていますね。ですから、これを実行すれば予定した動作のものが実現できます。

```
(myif (> x 5) (print 'large) (print 'small))
LARGE
=>LARGE
```

マクロの価値は、このように評価しない引数が取れるということと、実行前にまず展開するので、コンパイラを使用する場合にはコンパイル時に展開され、実行時の効率が良い (関数呼出を伴わない) ということです。

1.3 バッククォート記法

しかし、先の myif の定義本体は分かりづらいですね。「だいたいこういう形」「ただし一部には値を埋め込みたい」という場合には**バッククォート記法**というのが使えます。

```
`(a b c ,x d e)
=> (A B C 10 D E) ;; x のところは評価
`(a b c ,(* 2 x) d e)
=> (A B C 20 D E) ;; 式でもよい
```

バッククォートはクォートと似ているけれど、実際の中身は先の myif の本体のようにリストを使って実現されています。で、その主な用途はマクロ本体を書きやすくすることなわけです。

```
(defmacro myif (condition then else)
  `(cond (,condition ,then)
      (t ,else)))
```

なお、蛇足ながら、カンマ記法の親戚で「リストをその位置に貼り込む」,@記法というのもあります。次の例を見ると違いが分かります。

```
(setq l '(1 2 3)
=> (1 2 3)
`(a b c ,@l d e)
=> (A B C 1 2 3 D E)
`(a b c ,l d e)
=> (A B C (1 2 3) D E)
```

また、これと相性のよい defmacro の機能で「ある箇所から残りを 1 つのリストで受け取る」というのがあります。これを使うと、cond などの制御構造と同様、複数の処理を次々に書くようなマクロが作れます。

```
(defmacro mywhen (condition &rest body)
  `(cond (,condition ,@body)))
=> MYWHEN
(mywhen (> x 5) (print 1) (print 2))
1
2
=> 2
```

演習 1 ここまでに出て来たマクロの例を自分でもやってみなさい。動いたら、こんどは以下のような制御構造をマクロで作ってみなさい。

a. 指定した条件が成り立たない場合に一連の動作を実行するマクロ myunless(mywhen の条件が反転したもの)。

b. 指定した変数を下限から上限まで 1 ずつ変化させる制御構造 myfor。例: `(myfor i 1 5 (print i)) => 1 2 3 4 5`。

   ヒント: CommonLisp の loop 機能を使うと次のように書くことができるので、これに展開するマクロにするとよい。

   ```
   (loop for i from 1 upto 5 do (print i))
   ```

c. そのほか、自分で面白いと思う制御構造。

1.4 リスト操作とベクター

CommonLisp は豊富な関数群を予め提供しているので、これらに基づいて「分かりやすい」表現を使う、ということが前提になっています。そのため、これまでの左のものを右のような書き方に改めます。

```
× (car '(1 2 3)) =>  1       ○ (first '(1 2 3)) => 1
× (cdr '(1 2 3)) =>  (2 3)   ○ (rest '(1 2 3)) => (2 3)
× (cadr '(1 2 3)) => 2       ○ (second '(1 2 3)) => 2
× caddr, cadddr, ...         ○ (elt '(1 2 3) 2) => 3  ;; 先頭は 0
```

ただし、ドット対をドット対として扱う場合には、car と cdr が一番自然な表記方法だからそれで構いません。

また、リストは先頭から順にたどらなければならないのに対し、他の言語における配列のようなオブジェクトとしてベクターがあり、そのアクセスは上と同じ elt を使えます。

```
(vector 1 2 3 4)  ;; #(1 2 3 4) でも同じ
```

上記は直接初期値を指定する方法でしたが、大きさだけ指定することもできます。

```
(make-array 100)
(make-array '(10 10))  ;; 2次元配列
```

実はベクターは1次元配列のことであり、2次元以上の場合のアクセスは elt ではだめで「`(aref a 添字 添字 …)`」によります。

1.5 汎変数

これまで、変数に値を設定するのは setq を使うとしてきました。しかし Lisp では変数以外に cons 対の car/cdr 部やベクターの要素など、様々な場所に値を設定できます。これらを統一して扱うため、**汎変数**いうと言い方をします。そして、汎変数に値を設定するのには setf というマクロを使います。setf は普通の変数にも使えます。

```
(setf x '(a b))
=> (A B)
(setf (car x) '(1 2 3))
=> (1 2 3)
x
=> ((1 2 3) B)
(setf a #(1 2 3 4))
=> #(1 2 3 4)
(setf (aref a 2) 'zzz)
=> ZZZ
a
=> #(1 2 ZZZ 4)
```

1.6 構造体

構造体とは、複数の値を組にしたもので、それぞれの値は名前で指定できるようなものです。CommonLisp では構造体は defstruct で指定します。

```
(defstruct human name age height)
=> HUMAN
(setq h1 (make-human :name "Kuno" :age 20 :height 170))
=> #S(HUMAN :NAME "Kuno" :AGE 20 :HEIGHT 170)
```

なお、「:名前」というのは定数シンボルと呼ばれ、評価したときに自分自身と同じになるので、(変数として使う) 普通のシンボルと違いクォートをつけなくて済みます。主な用途は、上記のようにプログラム中のさまざまな予約語として使うことです。

さて、構造体を定義すると、自動的に「構造体名-フィールド名」というアクセス関数ができます。また、これは汎変数なので setf で値が設定できます。

```
(human-name h1)
=> "Kuno"
(setf (human-age h1) 30)
=> 30
h1
=> #S(HUMAN :NAME "Kuno" :AGE 30 :HEIGHT 170)
```

1.7 変数の使い分け

CommonLisp では通常の変数はレキシカルスコープを持つローカル変数 (**レキシカル変数**) ですが、これ以外に動的なスコープを持つ変数 (**スペシャル変数**) も存在します。そのような変数は defvar や defparameter で設定し、*...*という名前を持たせる慣例があります。

```
(defvar *debug-level* 3)
(defparameter *grammer-rules* '(....))
```

これら 2 者は機能は同じですが、defparameter の方は「プログラムの一部であり、安易には書き換えない」ものに使います。さらにもう 1 つ同じ機能のものとして、一度設定したら値は変えない (定数を設定する)、という目的には defconstant を使います。

```
(defconstant fail nil)  ;; 失敗の印
```

さて、スペシャル変数とは、グローバル変数なのでしょうか? そうではなく、**動的スコープを持つ変数**というのが正確なところです。次の例を見てください。

```
(defparameter *magic* 7)
=> *MAGIC*
*magic*
=> 7
(defun pmagic () (print *magic*))
=> PMAGIC
(pmagic)
7
=> 7
```

ここまではグローバル変数のように見えます。ここからが問題です。

```
(let ((*magic* 13)) (pmagic))
13
=> 13
```

このように、内側のスコープで同名の変数が定義された場合、pmagic の中からは「一番近いところで定義された」変数が参照されます。この一番近いというのは、プログラムの実行過程で動的に定まるので、スペシャル変数は動的なスコープを持つ、というわけです。

演習 2 グローバル変数とスペシャル変数は結局どう違うのか、自分なりのまとめ (説明文) を書いてみなさい。また、そのことをうまく使った実用的な例題を考案し、それも説明しなさい。

2 テーマ 1: 英文の生成

2.1 関数による直接生成

この節では、英文をランダムに生成する、という問題を扱います。Lisp では単語の並びはリストで自然に表せるので、このような問題は扱いやすいわけです。ただし、でたらめに単語を並べても英語の文になりませんから、文法は守る必要があります。

1 つの考え方として、文とか動詞などの名前の関数を呼ぶと、の構文単位を生成してくれる、というふうにする方法があります。

```
(defun random-elt (choices)
  "リストから要素をランダムに選ぶ"
  (elt choices (random (length choices))))

(defun one-of (set)
  "set から 1 つ要素を選び、リストとして返す"
  (list (random-elt set)))

(defun sentence () (append (noun-phrase) (verb-phrase)))
(defun noun-phrase () (append (Article) (Noun)))
(defun verb-phrase () (append (Verb) (noun-phrase)))
(defun Article () (one-of '(the a)))
(defun Noun () (one-of '(man ball woman table)))
(defun Verb () (one-of '(hit took saw liked)))
```

動かしてみましょう。

```
(sentence)
=> (THE BALL LIKED THE MAN)
(sentence)
=> (THE MAN SAW THE WOMAN)
(sentence)
=> (THE TABLE LIKED A MAN)
(sentence)
=> (A BALL HIT A TABLE)
```

演習 3 この例題を動かせ。動いたら、もう少し複雑な文法までカバーしてみよ。

2.2 エンジンとデータの分離

前節の方法の弱点は、文法のメンテナンスが面倒だということです。そこで、次のような「まとめた」形で文法を記述し (データ)、それをもとにして生成を行うコード (エンジン) は分離する、という方針を取ります。

```
(defparameter *simple-grammer*
  '((sentence -> (noun-phrase verb-phrase))
    (noun-phrase -> (Article Noun))
    (verb-phrase -> (Verb noun-phrase))
    (Article -> the a)
    (Noun -> man ball woman table)
    (Verb -> hit took saw liked))
  "簡単な英語の部分集合の文法")
```

文法記述の「->」は見やすさのために入れています。

これを用いたエンジン部分は次のようになります。

```
(defvar *grammer* *simple-grammer*
  "generate が参照する文法。*simple-grammer*以外のもも可")

(defun rule-lhs (rule)
  "文法規則の左辺"
```

```
  (first rule))

(defun rule-rhs (rule)
  "文法規則の右辺"
  (rest (rest rule)))

(defun rewrites (category)
  "このカテゴリとして書き換え可能なものを返す"
  (rule-rhs (assoc category *grammer*)))

(defun mappend (fn the-list)
  "fn をリストの各要素に適用し、結果を連結"
  (apply #'append (mapcar fn the-list)))

(defun generate (phrase)
  "phrase をランダムに生成する"
  (cond ((listp phrase)
         (mappend #'generate phrase))
        ((rewrites phrase)
         (generate (random-elt (rewrites phrase))))
        (t (list phrase))))
```

これを動かしているようすを示します。

```
(generate 'sentence)
=> (A MAN TOOK A BALL)
(generate 'sentence)
=> (A MAN LIKED THE BALL)
(generate 'sentence)
=> (A TABLE HIT A WOMAN)
```

文法がさっきと同じなら、生成されるものはこれまでと代わりません。しかし、文法を拡張するのが容易になったので、拡張しましょう。

```
(defparameter *bigger-grammer*
  '((sentence -> (noun-phrase verb-phrase))
    (noun-phrase -> (Article Adj* Noun PP*) (Name) (Pronoun))
    (verb-phrase -> (Verb noun-phrase PP*))
    (PP* -> () (PP PP*))
    (Adj* -> () (Adj Adj*))
    (PP -> (Prep noun-phrase))
    (Prep -> to in by with on)
    (Adj -> big little blue green adiabatic)
    (Article -> the a)
    (Name -> Pat Kim Lee Terry Robin)
    (Noun -> man ball woman table)
    (Verb -> hit took saw liked)
    (Pronoun -> he she it these those that)))
```

こちらを使ってみます。

```
(setq *grammer* *bigger-grammer*)
...
(generate 'sentence)
=> (TERRY TOOK THE MAN IN IT)
(generate 'sentence)
=> (A LITTLE ADIABATIC TABLE ON THESE TOOK ROBIN IN TERRY)
(generate 'sentence)
=> (THE MAN LIKED THE MAN WITH TERRY)
(generate 'sentence)
=> (A ADIABATIC WOMAN SAW A BLUE TABLE IN ROBIN)
```

だいぶ、それらしくなりましたね?

演習 4 自分でも文法をさらに拡張して試せ。また、もっと英文らしくするには、どういう機能がさらにあったらよいか考えてみよ。

3 テーマ 2: GPS

3.1 GPS とは

GPS は「General Problem Solver」つまり「汎用の、問題を解決するプログラム」いうと恐ろしい名前のプログラムで、1957 年に Alan Newell と Herbert Simon によって提唱されました。もちろん、本当に問題が何でも解決できるわけはないですが、歴史的に見て重要ではあります。

GPS の主要なメカニズムは「手段=目標分析 (mean-ends analysis)」で、これをヒューリスティックに基づいて実行して行くというのがコア部分です。問題例として、次のものを考えます。

> 私は息子を保育園へ連れて行きたい。私はその手段として自動車を持っているが、バッテリーがあがっている。新しいバッテリーが必要で、それには自動車修理店に頼んで、交換してもらう必要がある。しかし自動車修理店はそんなことは知らないから、電話などで連絡する必要がある…

先の文法のときと同様、「自分は何を持っている」「どんな手段が取れる」「その手段を取るには何が必要」「その手段の帰結として何が起こる」などのことがらをデータとして用意します。そして、それをもとに目的を達成する道筋を構成してくれるのがエンジンである GPS ということになります。

GPS の呼び出し方は次のようなものということにします。

```
(GPS '(持っているものの並び) '(目的の並び) 手段の並び)
```

3.2 GPS バージョン 1: データ構造

まずは簡単なバージョンを作ってみます。現在どの状態にあるかを変数*state*に入れますが、これは「現在持っているもののリスト」になります。そして*ops*は「利用可能な操作のリスト」なります。

```
(defvar *state* nil "現在状態。条件のリスト")

(defvar *ops* nil "利用可能な操作のリスト")

(defstruct op "操作"
  (action nil) (preconds nil) (add-list nil) (del-list nil))
```

個々の操作は op という構造体で定義します (フィールド名がかっこに入っているのは、初期値を一緒に指定する形式)。それぞれの操作には、「操作の名前」「何が必要か」「その操作によって何が達成できるか」「その操作によって何が消費されるか」の情報が付属します。ここで、どのような操作があるか分からないと読みづらいので、先の問題のための操作群を見せます。

```
(defparameter *school-ops*
  (list
   (make-op :action 'drive-son-to-school
            :preconds '(son-at-home car-works)
            :add-list '(son-at-school)
            :del-list '(son-at-home))
   (make-op :action 'shop-install-battery
            :preconds '(car-needs-battery shop-knows-problem
                                          shop-has-money)
            :add-list '(car-works))
   (make-op :action 'tell-shop-problem
            :preconds '(in-communication-with-shop)
            :add-list '(shop-knows-problem))
   (make-op :action 'telephone-shop
            :preconds '(know-phone-number)
            :add-list '(in-communication-with-shop))
   (make-op :action 'look-up-number
            :preconds '(have-phone-book)
            :add-list '(know-phone-number))
   (make-op :action 'give-shop-money
            :preconds '(have-money)
            :add-list '(shop-has-money)
            :del-list '(have-money))))
```

たとえば、息子を車で保育園につれていく操作 drive-son-to-school は、son-at-home と car-works という条件が必要であり、この操作を実行すると son-at-home は削除されて son-at-school が追加される、というふうになるわけです。

3.3 GPS バージョン 1: プログラム

では、プログラムの方を見ましょう。

```
(defun GPS (*state* goals *ops*)
  "GPS --- *ops*を利用して goals を達成"
  (if (every #'achieve goals) 'solved))
```

GPS 本体は、「すべての目標が達成されたら成功」ということで、ごく簡単ですね。

```
(defun achieve (goal)
  "ゴールは既に達成済みか適用可能な操作で達成可能なら達成"
  (or (member goal *state*)
      (some #'apply-op
            (remove-if-not
             #'(lambda (op) (appropriate-p goal op)) *ops*))))
```

ある目標を達成するには、まず*state*に含まれていればもう達成ずも、そうでなければ*ops*から適切な操作を 1 つ持って来て適用します。

```
(defun appropriate-p (goal op)
  "op は goal の add-list 中にあるなら適用可能"
  (member goal (op-add-list op)))
```

適切な操作とは、達成したい目標が add-list の中にあるような操作、ということですね。

```
(defun apply-op (op)
  "op が適用可能ならメッセージを出して*state*を更新"
  (when (every #'achieve (op-preconds op))
    (print (list 'executing (op-preconds op)))
    (setf *state* (set-difference *state* (op-del-list op)))
    (setf *state* (union *state* (op-add-list op)))
    t))
```

操作を適用するには、まず前提条件をすべて達成します (ここで失敗したら中止)。OK なら、メッセージを出し、del-list にあるものを削除し、add-list にあるものを追加します。以上です。

さっそく動かしてみます。

```
(GPS '(son-at-home have-money car-needs-battery know-phone-number)
     '(son-at-school) *school-ops*))
(EXECUTING (KNOW-PHONE-NUMBER))
(EXECUTING (IN-COMMUNICATION-WITH-SHOP))
(EXECUTING (HAVE-MONEY))
(EXECUTING (CAR-NEEDS-BATTERY SHOP-KNOWS-PROBLEM SHOP-HAS-MONEY))
(EXECUTING (SON-AT-HOME CAR-WORKS))
=>SOLVED
```

つまり、まず電話番号を確認し、修理店と連絡し、お金を払ってバッテリーを交換し、車が動くようになったので連れて行く。というふうにちゃんと動いています。

演習 5 この例題を自分でも動かしなさい。また、同じ場面でもっと別な状況設定で動かしてみなさい。たとえば、お金が無いとか、車は最初から動くとか。さらに、新たな操作を追加してみなさい。たとえば、「店の電話番号は知らないが、電話帳を手に取れば調べられる」など。

3.4 GPS バージョン 1 のいくつかの問題

たとえば、「明日遊園地に行きたいので、お金を残しておきたい」と思ったとします。

```
(GPS '(son-at-home have-money car-needs-battery know-phone-number)
     '(have-money son-at-school) *school-ops*)
(EXECUTING (KNOW-PHONE-NUMBER))
(EXECUTING (IN-COMMUNICATION-WITH-SHOP))
(EXECUTING (HAVE-MONEY))
(EXECUTING (CAR-NEEDS-BATTERY SHOP-KNOWS-PROBLEM SHOP-HAS-MONEY))
(EXECUTING (SON-AT-HOME CAR-WORKS))
=>SOLVED
```

あれあれ、店に頼んでお金を使ってしまったのに OK? これは、先にまずお金があることを OK にして、その後は顧みないためです。結局、*state*という 1 つの状態しか持たず、それを構わず書き換えてしまったのでは、うまく行かないときに別の方法でやり直すことができない、というのが根本的な問題です。また、「堂々巡り」になった場合なども考えると、過去にどんな操作を適用してきたかをちゃんと記録して対応するべきなのですね。

3.5 デバッグ出力

途中経過がもっとよく分かるように、デバッグ機能をまず入れます。

```
(defvar *dbg-ids* nil "dbg が使用する名前のリスト")

(defun dbg (id format-string &rest args)
  "(DEBUG ID) が指定ずみならデバッグ情報を出力"
  (when (member id *dbg-ids*)
    (fresh-line *debug-io*)
    (apply #'format *debug-io* format-string args)))

(defun dodebug (&rest ids)
  "指定した ids について dbg 出力を開始する"
  (setf *dbg-ids* (union ids *dbg-ids*)))

(defun undebug (&rest ids)
  "指定した ids について dbg 出力を終了する"
  (setf *dbg-ids* (if (null ids) nil
                      (set-difference *dbg-ids* ids))))

(defun dbg-indent (id indent format-string &rest args)
  "(dodebug id) が指定ずみなら字下げされたデバッグ情報を出力"
  (when (member id *dbg-ids*)
    (fresh-line *debug-io*)
    (dotimes (i indent) (princ " " *debug-io*))
    (apply #'format *debug-io* format-string args)))
```

使い方は「dodebug(名前)」することで、とその名前を指定した dbg-indent の呼び出し箇所でメッセージが表示される、という形です。

3.6 GPS バージョン 2

こんどは、構造体の形は変わらないけれど、操作を「executing ○○」という名前に統一します。

```
(defvar *ops* nil "利用可能な操作のリスト")

(defstruct op "操作"
  (action nil) (preconds nil) (add-list nil) (del-list nil))

(defun executing-p (x)
  "x は (executiong ...) という形か?"
  (starts-with x 'executing))
```

```
(defun starts-with (list x)
  "list の最初の要素は x か?"
  (and (consp list) (eql (first list) x)))

(defun convert-op (op)
  "op を executing... の形に変換する"
  (unless (some #'executing-p (op-add-list op))
    (push (list 'executing (op-action op)) (op-add-list op)))
  op)

(defun op (action &key preconds add-list del-list)
  "(executing op) の形に従う新しい op を作る"
  (convert-op (make-op :action action :preconds preconds
                       :add-list add-list :del-list del-list)))
```

この最後に出て来る関数 op で操作を生成します。内容はさっきと同じ。

```
(defparameter *school-ops*
  (list
   (op 'ask-phone-number
       :preconds '(in-commnication-with-shop)
       :add-list '(know-phone-number))
   (op 'drive-son-to-school
       :preconds '(son-at-home car-works)
       :add-list '(son-at-school)
       :del-list '(son-at-home))
   (op 'shop-install-battery
       :preconds '(car-needs-battery shop-knows-problem
                                     shop-has-money)
       :add-list '(car-works))
   (op 'tell-shop-problem
       :preconds '(in-communication-with-shop)
       :add-list '(shop-knows-problem))
   (op 'telephone-shop
       :preconds '(know-phone-number)
       :add-list '(in-communication-with-shop))
   (op 'look-up-number
       :preconds '(have-phone-book)
       :add-list '(know-phone-number))
   (op 'give-shop-money
       :preconds '(have-money)
       :add-list '(shop-has-money)
       :del-list '(have-money))))
```

では GPS 本体を示します。これまでと違い、各段階で選んだ操作のリストを返すようになっていて、それらの操作で達成すべき目標がキャンセルされてしまえば失敗ということをちゃんと見ています。

```
(defun GPS (state goals &optional (*ops* *ops*))
  "GPS --- *ops*を利用して states から初めてゴールを達成"
```

```
  (remove-if #'atom (achieve-all (cons '(start) state) goals nil)))
```

途中でメッセージを出すかわりに、achieve-all は操作のリストを返すようになったので、そこから要らないものを削除して表示用に返します。

```
(defun achieve-all (state goals goal-stack)
  "各ゴールを達成し、最後の時点でそれらが全て成立しているように"
  (let ((current-state state))
    (if (and (every #'(lambda (g)
                        (setf current-state
                              (achieve current-state g goal-stack)))
                    goals)
             (subsetp goals current-state :test #'equal))
        current-state)))
```

achieve-all は現在状態とゴール群とゴールスタック (現在目標としているゴールが先頭に積まれた並び) を受け取り、すべてのゴールについてそれを達成するともに、それによって変化した状態を current-state に覚えます。そして全部が終わったところでゴール群がすべて成立したままの状態にあるなら、最後の状態を返します。

```
(defun achieve (state goal goal-stack)
  "ゴールは既に成立しているか、適用可能な適切な op があるなら達成"
  (dbg-indent :gps (length goal-stack) "Goal: ~a:" goal)
  (cond ((member-equal goal state) state)
        ((member-equal goal goal-stack) nil)
        (t (some #'(lambda (op) (apply-op state goal op goal-stack))
                 (remove-if-not
                  #'(lambda (op) (appropriate-p goal op)) *ops*)))))
```

ゴールが (1) に既成立してるなら OK(先と同じ)、(2) 既に達成しようとしているものと同じなら堂々巡りなのであきらめ、(3) それ以外なら適切な操作を 1 つ持って来てやってみる (先と同じ)。

```
(defun member-equal (item list)
  (member item list :test #'equal))
```

member-equal は equal を使った要素検索。

```
(defun apply-op (state goal op goal-stack)
  "op が適用可能なら、新しい変換済みの state を返す"
  (dbg-indent :gps (length goal-stack) "Consider: ~a" (op-action op))
  (let ((state2 (achieve-all state (op-preconds op)
                             (cons goal goal-stack))))
    (unless (null state2)
      (dbg-indent :gps (length goal-stack) "Aciton: ~a" (op-action op))
      (append (remove-if #'(lambda (x)
                             (member-equal x (op-del-list op)))
                         state2)
              (op-add-list op)))))
```

操作を適用するには、まず前提条件をすべて達成しようとする。それが OK なら、状態から削除リストにあるものを削除したものの後に、追加リストを連結して返す。

```
(defun appropriate-p (goal op)
  "op が goal の add-list 中にあるなら、適用可能"
  (member-equal goal (op-add-list op)))
```

適用可能かどうかは、追加リスト中に指定の目標があるかどうか。

```
(defun use (oplist)
  "oplist をデフォルトの op のリストとして使用"
  (length (setf *ops* oplist))) ; 有用なものを返す
```

use は新しいもので、操作群を対話的に切り替えるのに使います。
では実行例を見てみましょう。

```
(use *school-ops*)
=> 7
(GPS '(son-at-home have-money car-needs-battery know-phone-number)
     '(have-money son-at-school))
=> NIL
(GPS '(son-at-home have-money car-needs-battery know-phone-number)
     '(son-at-school))
=> ((START) (EXECUTING TELEPHONE-SHOP) (EXECUTING TELL-SHOP-PROBLEM)
   (EXECUTING GIVE-SHOP-MONEY) (EXECUTING SHOP-INSTALL-BATTERY)
   (EXECUTING DRIVE-SON-TO-SCHOOL))
```

ただし、このバージョンでも「とりあえず選択可能な操作をやる」いうと点は変わっていないので、複数の選択肢があって片方だけうまく行くような場合は、たまたまそちらを選ぶかどうかで成功/失敗が分かれてしまいます。

演習 6 この例題を自分で動かし、操作をいくつか追加して (たとえば自分でバッテリーを交換するとか)、お金を残しておくことがうまく行くようにしてみなさい。うまくいくかどうかは GPS が操作を考慮する順番に依存することも確認しなさい。

3.7 別の問題: 猿とバナナ

GPS が「汎用」であることを示す例として、まったく別の問題を解いてみましょう。これは「猿とバナナ」という著名な問題で、次のような構造になっています。

- 猿は檻の入口にいて、床上でボールを持っていて、腹がすいている。
- 檻の中央にバナナがつるしてある。
- バナナには猿の手がとどかないが、椅子に乗れば届く。
- 椅子が猿のところにあって、それを中央まで押していける。
- 椅子を押すには、ボールを持っていたらだめ。

猿が問題を解決するには、どうしたらいいでしょう?

```
(defparameter *banana-ops*
  (list
   (op 'climb-on-chair
       :preconds '(chair-at-middle-room at-middle-room on-floor)
       :add-list '(at-bananas on-chair)
```

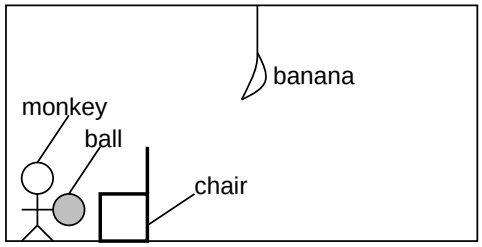

図 1: 猿とバナナ

```
        :del-list '(at-middle-room on-floor))
    (op 'push-chair-from-door-to-middle-room
        :preconds '(chair-at-door at-door)
        :add-list '(chair-at-middle-room at-middle-room)
        :del-list '(chair-at-door at-door))
    (op 'walk-from-door-to-middle-room
        :preconds '(at-door on-floor)
        :add-list '(at-middle-room)
        :del-list '(at-door))
    (op 'grasp-bananas
        :preconds '(at-bananas empty-hands)
        :add-list '(has-bananas)
        :del-list '(empty-hands))
    (op 'drop-ball
        :preconds '(has-ball)
        :add-list '(empty-hands)
        :del-list '(has-ball))
    (op 'eat-bananas
        :preconds '(has-bananas)
        :add-list '(empty-handed not-hungry)
        :del-list '(has-bananas hungry))))
```

これを GPS に解かせてみます。

```
(GPS '(has-ball on-floor at-door chair-at-door hungry)  '(not-hungry))
=> ((START) (EXECUTING PUSH-CHAIR-FROM-DOOR-TO-MIDDLE-ROOM)
   (EXECUTING CLIMB-ON-CHAIR) (EXECUTING DROP-BALL) (EXECUTING
   GRASP-BANANAS) (EXECUTING EAT-BANANAS))
```

すばらしい。ちゃんと解けているようです。

演習 7 この例題を自分で動かし、さまざまな状態で猿がバナナを取れるかどうか、本当は取れるはずなのにうまく行かないのはどういう倍かを調べてみなさい。

3.8 別の問題: 迷路

今度は図 2 のような迷路の入口から出口までの経路を GPS に探索させてみます。

迷路の場合は 2 つの地点のどちらからどちらへも行けるので、2 地点を指定してその両方向の移動操作を一緒に作るようにします。

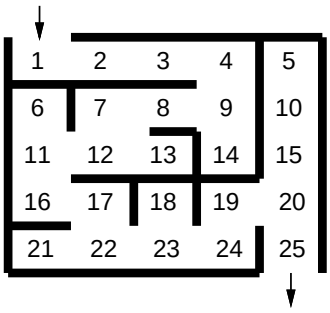

図 2: 簡単な迷路

```
(defun make-maze-ops (pair)
  "迷路の op を双方向に作る"
  (list (make-maze-op (first pair) (second pair))
        (make-maze-op (second pair) (first pair))))

(defun make-maze-op (here there)
  "2 つの場所の間で移動する op を作る"
  (op `(move from ,here to ,there)
      :preconds `((at ,here))
      :add-list `((at ,there))
      :del-list `((at ,here))))

(defun mappend (fn the-list)
  "fn をリストの各要素に適用し、結果を連結"
  (apply #'append (mapcar fn the-list)))

(defparameter *maze-ops*
  (mappend #'make-maze-ops
           '((1 2) (2 3) (3 4) (4 9) (9 14) (9 8) (8 7)
             (7 12) (12 13) (12 11) (11 6) (11 16)
             (16 17) (17 22) (22 21) (22 23) (23 18)
             (23 24) (24 19) (19 20) (20 15) (20 25)
             (15 10) (10 5))))
```

最後の defparameter で隣接点のリストをもとにすべての操作を生成しています。では、これを動かしてみましょう。

```
(use *maze-ops*)
=> 48
(gps '((at 1)) '((at 25)))
=> ((START) (EXECUTING (MOVE FROM 1 TO 2)) (EXECUTING (MOVE FROM 2 TO 3))
   (EXECUTING (MOVE FROM 3 TO 4)) (EXECUTING (MOVE FROM 4 TO 9))
   (EXECUTING (MOVE FROM 9 TO 8)) (EXECUTING (MOVE FROM 8 TO 7))
   (EXECUTING (MOVE FROM 7 TO 12)) (EXECUTING (MOVE FROM 12 TO 11))
   (EXECUTING (MOVE FROM 11 TO 16)) (EXECUTING (MOVE FROM 16 TO 17))
   (EXECUTING (MOVE FROM 17 TO 22)) (EXECUTING (MOVE FROM 22 TO 23))
   (EXECUTING (MOVE FROM 23 TO 24)) (EXECUTING (MOVE FROM 24 TO 19))
   (EXECUTING (MOVE FROM 19 TO 20)) (EXECUTING (MOVE FROM 20 TO 25)) (AT 25))
```

確かにちゃんと 1 から 25 へ到達しました。

演習 8 この例題を自分で動かしてみよ。1 から 25 以外の経路についても試せ。納得したら、別の迷路を自分で作ってそれも解かせてみよ。

3.9 別の問題: 積み木の操作

テーブルの上にいくつかの積み木があってこれを積むという問題もよく使われます。要件は次の通りです。

- 積み木は、テーブルの上にあるか、または他の積み木の上にある。
- 積み木の上が空いていれば、その上に 1 つだけ積み木が積める。高さには制限はない。

これらの前提の上で、図 3 のような変更をどうやって実現するかを考えさせるわけです。

この問題を表現するため、積み木のリストを与えたとき、すべての操作を自動生成するようにします。具体的には任意の積み木 A、B、C(互いに異なる) に対して、「A を B の上から C の上に移す」を生成し、また任意の積み木 A、B(互いに異なる) に対して「A を机の上から B に移す」「A を B の上から机の上に移す」を生成します。

```
(defun make-block-ops (blocks)
  (let ((ops nil))
    (dolist (a blocks)
      (dolist (b blocks)
        (unless (equal a b)
          (dolist (c blocks)
            (unless (or (equal c a) (equal c b))
              (push (move-op a b c) ops)))
          (push (move-op a 'table b) ops)
          (push (move-op a b 'table) ops))))
    ops))

(defun move-op (a b c)
  "A を B から C へ移す op を作る"
  (op `(move ,a from ,b to ,c)
      :preconds `((space on ,a) (space on ,c) (,a on ,b))
      :add-list (move-ons a b c)
      :del-list (move-ons a c b)))

(defun move-ons (a b c)
  (if (eq b 'table)
      `((,a on ,c))
      `((,a on ,c) (space on ,b))))
```

では、まず問題 (1) からやってみましょう。

```
(use (make-block-ops '(a b)))
=> 4
(gps '((a on table) (b on table) (space on a) (space on b) (space on table))
     '((a on b) (b on table)))
=> ((START) (B ON TABLE) (SPACE ON A) (SPACE ON TABLE)
    (EXECUTING (MOVE A FROM TABLE TO B)) (A ON B))
```

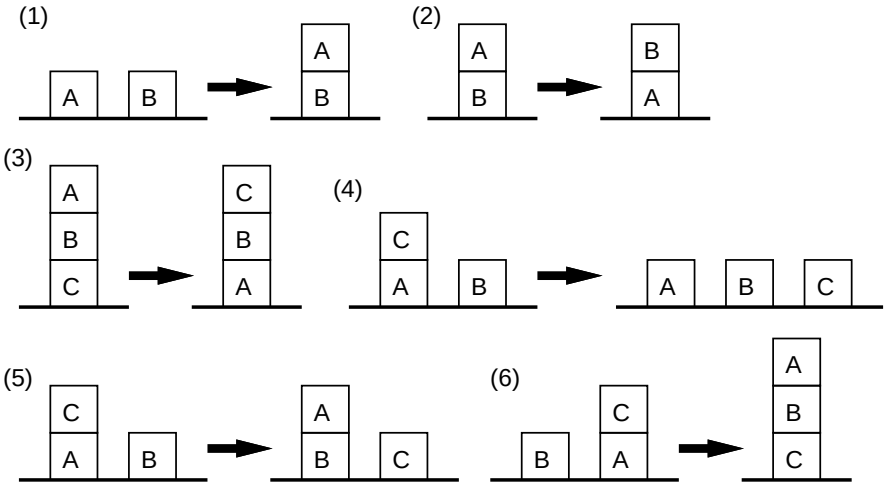

図 3: ブロック移動の問題

正しくできているようです。次は (2) すが、でもうちょっと情報が分かるように、dodebug を指定してから実行します。

```
(dodebug :gps)
=> :GPS
(gps '((a on b) (b on table) (space on a) (space on table)) '((b on a)))
=> Goal: (B ON A):
   Consider: (MOVE B FROM TABLE TO A)
    Goal: (SPACE ON B):
    Consider: (MOVE A FROM B TO TABLE)
     Goal: (SPACE ON A):
     Goal: (SPACE ON TABLE):
     Goal: (A ON B):
    Aciton: (MOVE A FROM B TO TABLE)
    Goal: (SPACE ON A):
    Goal: (B ON TABLE):
   Aciton: (MOVE B FROM TABLE TO A)
   ((START) (SPACE ON TABLE) (EXECUTING (MOVE A FROM B TO TABLE)) (A ON TABLE)
    (SPACE ON B) (EXECUTING (MOVE B FROM TABLE TO A)) (B ON A))
```

うまく行っているようです。

ちょっと面倒な問題として、今度は (3) を考えます。

```
(gps '((c on table) (b on c) (a on b) (space on a) (space on table))
 '((b on a) (c on b)))
=> ((START) (SPACE ON TABLE) (EXECUTING (MOVE A FROM B TO TABLE)) (A ON TABLE)
    (EXECUTING (MOVE B FROM C TO A)) (B ON A) (SPACE ON C)
    (EXECUTING (MOVE C FROM TABLE TO B)) (C ON B))
```

ちゃんとできています。すばらしい。しかし、目標の与え方が上では「B は A の上、C は B の上」したが、でこれを逆にすると失敗します。

```
(gps '((c on table) (b on c) (a on b) (space on a) (space on table))
 '((c on b) (b on a)))
=> NIL
```

再び dodebug を使って様子を見ます。

```
(dodebug :gps)
=> :GPS
(gps '((c on table) (b on c) (a on b) (space on a) (space on table))
 '((c on b) (b on a)))
Consider: (MOVE C FROM TABLE TO B)
 Goal: (SPACE ON C):
 Consider: (MOVE B FROM C TO TABLE)
  Goal: (SPACE ON B):
  Consider: (MOVE C FROM B TO TABLE)
   Goal: (SPACE ON C):
  Consider: (MOVE C FROM B TO A)
   Goal: (SPACE ON C):
  Consider: (MOVE A FROM B TO TABLE)
   Goal: (SPACE ON A):
   Goal: (SPACE ON TABLE):
   Goal: (A ON B):
  Aciton: (MOVE A FROM B TO TABLE)
  Goal: (SPACE ON TABLE):
  Goal: (B ON C):
 Aciton: (MOVE B FROM C TO TABLE)
 Goal: (SPACE ON B):
 Goal: (C ON TABLE):
Aciton: (MOVE C FROM TABLE TO B)
Goal: (B ON A):
Consider: (MOVE B FROM C TO A)
 Goal: (SPACE ON B):
 Consider: (MOVE C FROM B TO TABLE)
  Goal: (SPACE ON C):
  Goal: (SPACE ON TABLE):
  Goal: (C ON B):
 Aciton: (MOVE C FROM B TO TABLE)
 Goal: (SPACE ON A):
 Goal: (B ON C):
 Consider: (MOVE B FROM TABLE TO C)
  Goal: (SPACE ON B):
  Goal: (SPACE ON C):
  Goal: (B ON TABLE):
 Aciton: (MOVE B FROM TABLE TO C)
Aciton: (MOVE B FROM C TO A)
=> NIL
```

なかなか長いですが、読み解くと「C を B の上に乗せるために、B を C の上からテーブルに降ろそうするが、そのためには A を B 上からテーブルに降ろし、そして B を C の上からテーブルに降ろし、C を B の上に乗せ、次に B を A の上に乗せようとするがどうにもならない」ということですね。このような、順序の問題をちゃんと解くには GPS の能力では足りない、ということなわけです。

演習 9 図 3 の他の問題もやらせてみなさい。目標の指定方法による違いがあるかどうかも調べ、また成功したもの・失敗したものともに「経過報告」を日本語で簡単に説明しなさい。

演習 10 GPS に何か他の面白い課題を解かせてみなさい。