エキスパートシステム

人工知能研究の産業への応用としてもっとも成功したエキスパートシステム (ES) について、その機能を習得する。ES は人間の知的活動の多くの部分を占める論理的な思考の過程をモデル化することにより、エキスパートの判断を代行するシステムである。本実験では実際に ES 構築ツールを用いて、診断型、設計型の代表的な 2 つの ES を構築することで、問題を知識として体系化する手法を習得する。

第 1 日目	エキスパートシステム概要、ツールの使い方
第 2 日目	診断型エキスパートシステム
第 3 日目	設計型エキスパートシステム
第 4 日目	自由課題によるエキスパートシステム作成

目次

佐川雄二　1995/2/10

エキスパートシステム

1　目的

人工知能研究の産業への応用として最も成功したエキスパートシステムについて、実際にエキスパートシステムを構築することで、その機能を習得する。またその構築の上で大きな比重を占める問題を知識として体系化する手法を習得する。

2　解説

2.1　エキスパートシステムとは

人間の知的活動の多くは、論理的な思考に基づいている。特に「エキスパート」と呼ばれるある分野の専門家が行なっている判断は、この傾向が強い[1]。

論理的な思考の基本は、すでにわかっている**事実**と、「〜ならば…である。…ならば ― である。」という形で表すことのできる**規則**を組合わせて推論を繰り返すことである。したがってエキスパートの持っている知識をこのような形で記述できれば、エキスパートの行なう判断をコンピュータに代行させることができると考えられる。これがエキスパートシステムと呼ばれるシステムである。

例えば、以下のような推論規則と、

1. 体毛を持つ動物はほ乳類である
2. 授乳する動物はほ乳類である
3. 羽根を持つ動物は鳥である
4. 飛び、かつ卵を産む動物は鳥である
5. 肉を食べる動物は食肉獣である
6. 鋭い歯を持ち、鋭い爪を持ち、前向きの目を持つ動物は食肉獣である
7. ほ乳類で、蹄を持つ動物は有蹄動物である
8. ほ乳類で、反芻する動物は有蹄動物である
9. ほ乳類で、食肉獣で、黄褐色で、黒い斑点がある動物はチータである
10. ほ乳類で、食肉獣で、黄褐色で、黒い縞がある動物は虎である
11. 有蹄動物で、長い首を持ち、長い脚を持ち、黒い斑点のある動物はキリンである

[1]逆にものをしゃべったり、見たりする活動の方が、論理的に言葉で表現することが難しく、コンピュータに行なわせることが困難である。

12. 有蹄動物で、黒い縞のある動物はシマウマである

13. 鳥で、飛ばなくて、長い首を持ち、長い脚を持ち、白黒である動物はダチョウである

14. 鳥で、飛ばなくて、泳ぎ、白黒である動物はペンギンである

15. 鳥で、うまく飛ぶ動物はアホウドリである

16. ある種族の動物の子どもは同じ種族に属する

以下のような事実から、

1. ロビーには黒い斑点がある

2. ロビーは黄褐色である

3. ロビーは肉を食べる

4. ロビーは体毛がある

5. スージーには羽根がある

6. スージーはうまく飛ぶ

次のような事実を推論することができる。

1. ロビーはほ乳類である

2. スージーは鳥である

3. ロビーは食肉獣である

4. ロビーはチーターである

5. スージーはアホウドリである

2.2 エキスパートシステムの種類

エキスパートシステムは、それが扱う問題のタイプによっていくつかのタイプに分けることができる。

1. 解析型問題

   システムの構造をもとに、システムの性質を推定したり、システムを制御したりする問題

   (a) 解釈問題

   計測器などから収集したデータを基にその意味を決定する問題

   【例】分子の部分構造の質量を質量分析器から得て、その化学構造を決定する

   (b) 診断問題

   システムに異常が発生した時、観測されたデータを基に異常の原因を決定する問題

   【例】医療診断、機械の故障診断

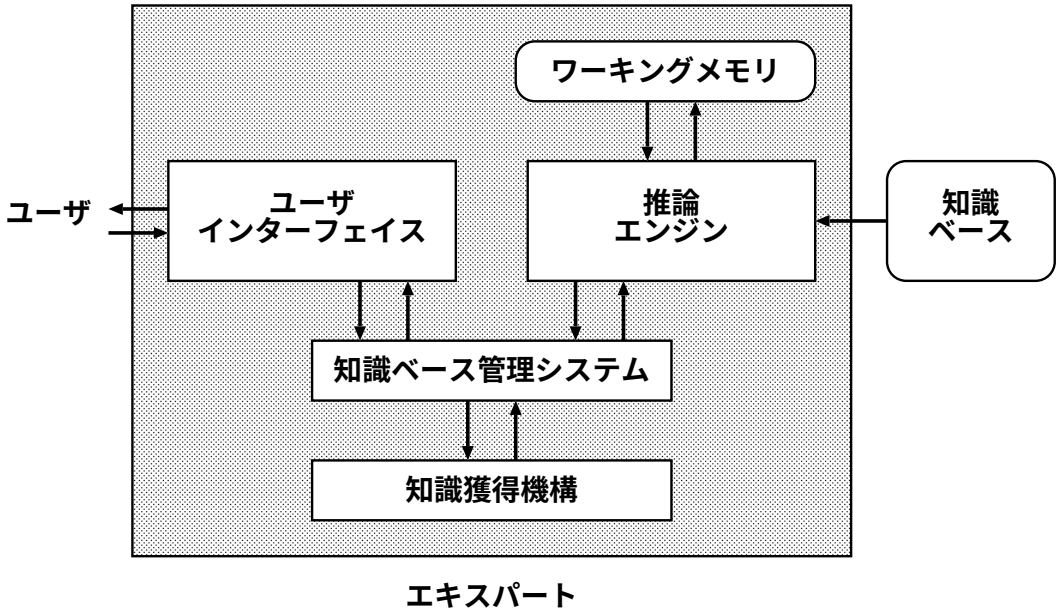

図 1: エキスパートシステムの概要

(c) 制御問題

システムが計画通りに動作するよう、システムを監視し操作する問題

【例】呼吸補助装置を使用する手術後の患者の監視

2. 合成型問題

システムの性質をもとに、システムの構造を決定する問題

(a) 計画問題

与えられた目標を達成するために行なう一連の行動を決定する問題

【例】遺伝子操作実験のための実験計画システム

(b) 設計問題

特定の要求を満たす対象物を作成するための仕様を決定する問題

【例】ディジタル回路の設計

2.3 エキスパートシステムの構築法

2.3.1 エキスパートシステム構築ツール

エキスパートシステムの概略は図 1のようになる。このように知識とそれに基づいて推論する部分が明確にわかれているのがエキスパートシステムの特徴である。したがってこのうち黒く囲った部分のみを作成しておけば、知識ベースを入れ換えるのみで、さまざまなエキスパートシステムを構築できる。この目的で作られたシステムがエキスパートシステム構築ツールである。

エキスパートシステム構築ツールには、使用する知識表現 (ルール、意味ネットワーク、フレーム) やユーザインターフェイスの枠組によってさまざまなものが作られ、市販されて

表 1: 主なエキスパートシステム構築ツール

ルール型	OPS5, S1
フレーム型	UNITS, SRL
ブラックボード型	AGE
ハイブリッドツール	KEE, ART, Knowledge Craft
組み込み型ツール	OPS83
領域特化型ツール	G2, FLEXSYS
タスク指向ツール	CSRL, DSPL
簡易ツール	大創玄, GURU, Nexpert Object

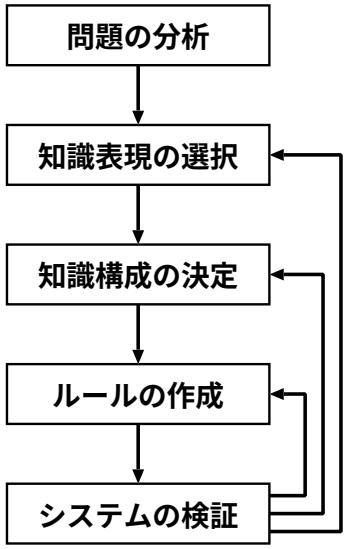

図 2: エキスパートシステムの構築手順

いる。表 1に主なエキスパートシステム構築ツールをあげる。これらの中には、ユーザが推論機構の制御に一部手を加えることのできるものもある。

2.3.2 構築手順

エキスパートシステムは図 2のような手順で構築される。

1. 問題の分析

   以下のような問題の性質や範囲を調べる。

   - エキスパートシステム向きの問題かどうかのチェック
     エキスパートシステム向きの問題とは、数理的・アルゴリズム的な解決が難しい ill-structured な問題である。
   - 問題の構造の複雑さ・変化の度合のチェック

あまりにも複雑過ぎる問題や変化の激しい問題は開発や保守のコストがかかりすぎる。

- 構築されたシステムの利用形態のチェック
  使われないものを作るのは無駄だし、保守の形態によってシステムのインターフェイスなどを考えなければならない。

2. 知識表現の選択

   問題の分析結果に基づいて、必要な概念や関係を調べる。問題の分析および知識表現の選択は、システム作成者とエキスパートが別人の場合はインタビューを通して行なわれる。その結果もっとも適切と思われるエキスパートシステム構築ツールを選択する。

3. 知識構成の決定

   選択したツールを用いて具体的な知識や経験例を表現する方法を設計する。大規模なシステムでは、知識のグループ化により開発と処理を行ないやすくする。

4. ルールの作成

   実際にルールを記述し、プロトタイプと呼ばれる実験的なシステムを構築する。エキスパートシステム構築の特徴は、始めからすべてを明確にすることなく、とりあえず動くプロトタイプを徐々に練り上げていくというプロトタイピング的なプログラミングスタイルがとれる点にある。

5. システムの検証

   プロトタイプがシステムの目的を十分達成できるかチェック。達成していなければ再設計を行なう。

2.4 TEST 使用の手引

2.4.1 概要

本実験で使用するエキスパートシステム構築ツール TEST(Tiny Expert System building Tool) は、基本的なエキスパートシステムプログラミングの習得を目的として、名古屋大学杉江研究室において作成されたものである。 TEST は Common Lisp 上で動作する。

TEST は、前向き推論機構のみをサポートするルール型のツールである。

2.4.2 基本動作

TEST にはファクトとルールの 2 種類の知識がある。

ファクトはあらかじめわかっている事実 (fact) で、ファクトファイルから読み込まれ、推論の実行に先だってワーキングメモリに入れられる。

ルールは「もし〜ならば…である」という形の規則で、〜の部分を LHS(Left Hand Side)、…の部分を RHS(Right Hand Side) と呼ぶ。ルールは、ルールファイルから読み込まれ、知識ベースに入れられる。

推論の実行は、図 3に示すように、**マッチング、競合解決、ルールの実行**という 3 つのステップからなる認識・実行サイクルを繰り返して行なわれる。

マッチングは、知識ベースの各ルールの LHS とワーキングメモリを比較して、実行可能なルールを選び出すステップである。

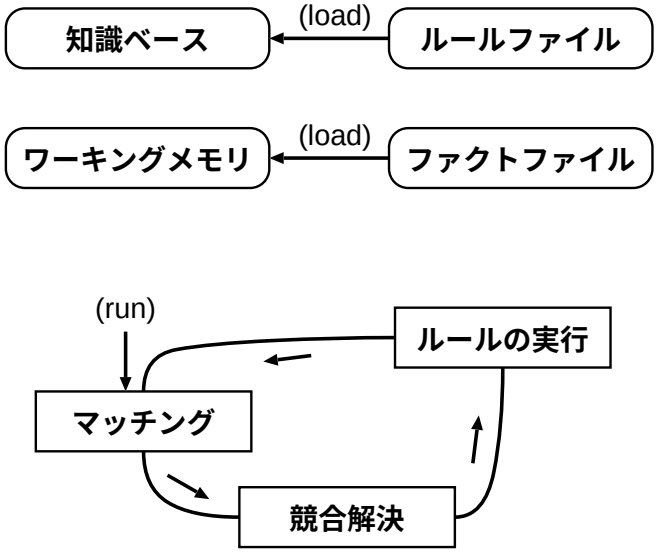

図 3: TEST 基本動作

競合解決は、実行可能なルールの中から実際に実行するルールを一つだけ選び出すステップである。ルールに優先順位をつけたりする方法もあるが、現在のバージョンでは簡単のため最初に見つかったルールを選択するようになっている。

ルールの実行は、選び出されたルールの RHS を実行するステップである。この実行によってワーキングメモリに新たなファクトが追加されることがあるので、実行のステップの後、再びマッチングのステップに戻る。

このサイクルを実行するルールがなくなるまで繰り返すのが TEST の基本動作である。

2.4.3 起動、実行および終了

起動および実行

1. Lisp を立ちあげ、システムをロードする。

2. TEST が立ち上がり、 TEST のプロンプト”>” が表示される[2]。

```
% lisp ↓
(数行のメッセージ)
>(load "test")
```

3. ルールファイルの読み込み。ルールファイルは任意のエディタで作成する。

```
> (load "ファイル名") ↓
; loading "ファイル名"
>
```

4. ファクトファイルの読み込み。ファクトファイルは任意のエディタで作成する。

[2] ↓はリターンキーの入力を表す。

```
> (load "ファイル名") ↓
; loading "ファイル名"
>
```

5. 推論の実行。正常に終了すれば TEST のプロンプトに戻る。

```
> (run) ↓
実行過程の表示
>
```

終了　TEST のプロンプトで (quit) と入力する。

```
> (quit) ↓
%
```

正常に実行できない場合　ルールやファクトの記述誤りなどで、推論が正しく行なわれなかった場合、 TEST はエラーメッセージを表示して停止する。この時プロンプトが前に数字のついたものに変わる。このプロンプトに対しても、 load、 run、 quit は有効に働く。

2.4.4　ファクトの記述

ファクトの記述には deffacts を使用する。

deffacts　(マクロ)

引数 ファクト式 1 ... ファクト式 n

機能 ファクト式 1 ... ファクト式 n をワーキングメモリにセットする。

説明 各ファクト式は括弧で囲まれたリスト表現でなければならない。リストの各要素は英数字からなるシンボルかリストでなければならない (例参照)。あるファクトをどのようなリストで表すかは、プログラマに任されている。シンボルの大文字、小文字は区別されない。ワーキングメモリの各要素は wme(working memory element) と呼ばれる。

例 3 つのファクトがセットされる。

```
(deffacts
   (I have a pen)        ; 英文風、(i have a pen) でも同じ
   (not (i am silly))    ; リストを含むリスト
   (love tom mary))      ; 命題風
```

2.4.5　ルールの記述

ルールの記述には defrule を使用する。

defrule （マクロ）

引数 ルール名 LHS 式 1 ... LHS 式 n − > RHS 式 ... RHS 式 n

機能 与えられた LHS および RSH を持つルールを与えられたルール名を持つルールとして知識ベースに格納する。

説明 各 LHS 式は AND でつながれていると解釈される。すなわち、 LHS 式 1 ... LHS 式 n のすべてがワーキングメモリとマッチした時のみ、そのルールが実行可能となる。

LHS 式がワーキングメモリとマッチするとは、ワーキングメモリの中にその LHS 式と全く同じ要素、構造からなる wme が存在することをいう[3]。ただし LHS 式には変数を含むことができる。変数は? で始まるシンボルで表す。変数はいかなるシンボルともマッチすることができる ((i love ?person) は (i love mary) とマッチする) が、同じルール内の同じ名前の変数は同じシンボルとしかマッチできない ((i love ?person),(?person is female) と (i love mary),(mary is female) はマッチするが、 (i love ?person),(?person is female) と (i love john),(mary is female) はマッチしない)。

また、 (not 式) の形をした LHS 式は、式とマッチする wme が存在しない場合か、(not 式) と全く同じ wme が存在する時にマッチする。

ルールが選択され実行されると、 RHS 式 ... RHS 式 n が順に実行される。 RHS 式としては、 assert, printout, ask-if, set 式のみが許される。各式の詳細は後述。また、 LHS のマッチングに際に行なわれた変数の代入は RHS にも適用される。たとえば、 LHS が?person=mary でマッチしたとすると、その同じルールの RHS 式 (assert (?person loves me)) は (assert (mary loves me)) として実行される。

例 以下のような定義のルールに対して、

```
(defrule my-favorite                    ; my-favorite がルール名
   (i am male)                          ; もし私が男で
   (?person is female)                  ; ?person が女性で
   (?person is pretty)                  ; ?person がかわいい
->                                      ; ならば
   (assert (i love ?person))            ; 私は?person が好き
   (printout "I love " ?person "!"))    ; そのことを表示
```

ワーキングメモリの内容が以下のようなら、

```
(i am male)
(john is male)
(mary is female)
(cathy is female)
(mary is pretty)
```

このルールは実行可能で実行されるのは、以下 2 つの式である。

```
(assert (i love mary))
(printout "I love " mary "!")
```

[3](i love you) と (love i you) はマッチしない。

assert （マクロ）

引数 ファクト式

機能 ファクト式をワーキングメモリにセットする。

説明 ファクト式は deffacts の項の説明と同じ。

例 (i love mary) がセットされる。

```
(assert (i love mary))
```

printout （マクロ）

引数 文字列またはシンボル 1 ... 文字列またはシンボル n

機能 文字列またはシンボル 1 ... 文字列またはシンボル n を順に端末に表示する。

説明 文字列は "(ダブルクォテーション) で囲む必要がある。日本語は不可。シンボルはそのままで良い。

例 I love mary! と表示される。

```
(printout "I love " mary "!")
```

ask-if （マクロ）

引数 then 式 else 式 文字列またはシンボル 1 ... 文字列またはシンボル n

機能 質問文を表す文字列またはシンボル 1 ... 文字列またはシンボル n を順に端末に表示し、ユーザの答えが y ならば then 部を、 n ならば else 部を実行する。

説明 文字列は "で囲む必要がある。日本語は不可。シンボルはそのままで良い。 else 部は省略できない。何もしない場合は nil と記述する。 then 部、 else 部には、 RHS 式として許される任意を式を記述することができる。 2 つ以上の式を実行する場合にはリストにする必要がある。

例 答えによってワーキングメモリにセットする内容を変える。

```
(ask-if (assert (i am male))            ; y の場合
        (assert (i am female))          ; n の場合
        "Are you a male?")
(ask-if ((assert (i am male))           ; y の場合
         (assert (i am japanese)))      ; y の場合
        nil                             ; n の場合は何もしない
        "Are you a Japanese male?")
```

たとえば、最初の式の場合、実行されると以下のようにユーザに尋ねる。

```
Are you a male?(y/n)
```

ここでは y または n 以外の入力は受け付けられず、もう一度尋ねられる。

set (マクロ)

引数 変数 値

機能 変数に値を代入する。

説明 ここで指定される変数は同ルール内でこれ以前に使われていてはならない。したがってここでの代入結果はこれ以降の RHS 式について有効である。この制約に違反した場合の動作は保証しない。

値には以下の 4 種類のものが記述できる。

シンボル: そのシンボルが代入される。

select: メニューを提示してユーザに値を問い合わせる。後述。

ask-what: ユーザに値を問い合わせる。後述。

算術演算: 任意の Lisp 算術演算が記述できるが、簡単のため四則演算のみ示す。足し算 (+ 数 数)、引き算 (- 数 数)、かけ算 (* 数 数)、割算 (/ 数 数)。

例 それぞれ?var に mary, aichi, sag, 30 が代入される。

```
(set ?var mary)
(set ?var (select (("Aichi-ken" aichi)
                   ("Mie-ken" mie)
                   ("Gifu-ken" gifu))
                  "Where do you live?"))  ; ユーザが 1 を選択した時
(set ?var (ask-what "What is your name?")); ユーザが sag と返答した時
(set ?var (* 10 3))
```

select (マクロ)

引数 選択肢 文字列またはシンボル 1 ... 文字列またはシンボル n。ただし選択肢は表示と値の組のリスト。

機能 質問文を表す文字列またはシンボル 1 ... 文字列またはシンボル n を順に端末に表示し、選択肢を表示し、ユーザの答えた番号に対応する値を返す。

説明 文字列、選択肢の表示部は”で囲む必要がある。日本語は不可。シンボルはそのままで良い。

例 set の例参照。

この式が実行されると、以下のようにユーザに尋ねる。

```
Where do you live?
     1: Aichi-ken
     2: Mie-ken
     3: Gifu-ken
     (番号を選んで下さい)
```

メニューの範囲外の数字や数字以外のものを答えた時は、もう一度尋ねられる。

ask-what (マクロ)

引数 文字列またはシンボル 1 ... 文字列またはシンボル n

機能 質問文を表す文字列またはシンボル 1 ... 文字列またはシンボル n を順に端末に表示し、ユーザの答えを set に返す。

説明 文字列は”で囲む必要がある。日本語は不可。シンボルはそのままで良い。

例 set の例参照。

この式が実行されると、以下のようにユーザに尋ねる。

```
What is your name?
```

2.4.6 その他の機能

load (関数)

引数 ファイル名

機能 ファイル名で示すファクトファイルまたはルールファイルを読み込む。

説明 ファイル名は”で囲まなければならない。

例 「起動および実行」参照。

run (関数)

引数 なし

機能 推論を開始する。

説明 続けて実行する時は reset と load を使ってワーキングメモリを初期化する必要がある。

例 「起動および実行」参照。

reset (関数)

引数 なし

機能 ワーキングメモリを空にする。

説明 推論を続けて実行する時は reset のあと、ファクトファイルのみ読み込んで、ワーキングメモリを初期化する必要がある。

例 推論を 2 回続ける例。

```
> (load "test.rl") ↓          ; ルールファイルの読み込み
> (load "test.fct") ↓         ; ファクトファイルの読み込み
> (run) ↓                     ; 1 回目の実行
実行過程の表示
> (reset) ↓                   ; ワーキングメモリを空に
> (load "test.fct") ↓         ; ファクトファイルの読み込み
> (run) ↓                     ; 2 回目の実行
```

forget (関数)

引数 なし

機能 ワーキングメモリおよび知識ベースを空にする。

説明 変更したルールを読み込んで、推論を続けて実行する時は forget のあと、ルールファイル、ファクトファイル読み込む必要がある。

step (関数)

引数 なし

機能 推論のステップ実行

説明 ルールが 1 つ実行される度に、どのルールが実行されたかの表示をして停止する。ここでユーザは以下のようなサブコマンドを実行することができる。

P(Proceed): もう 1 ステップ先に進む。

W(Working memory): 現在のワーキングメモリの中身を表示する。

ルール名: ルールの定義を表示する。

G(Go): ステップ動作を解除して、最後まで推論を続ける。

ステップ動作は、デバッグに有効である。

quit (関数)

引数 なし

機能 TEST を終了する。

説明 TEST を終了して、シェルに戻る。

例 「起動および実行」参照。

2.4.7 記述例

1. 動物分類エキスパートシステム

   先に述べたような動物の分類に関する知識を用いて推論を行なうエキスパートシステムを TEST を用いて記述した例を示す。(一部)

   (プログラム 1)

```
;;;
;;; 動物園のルール
;;;

(defrule identify1
  (?animal has hair)
->
  (assert (?animal is a mammal))
```

```
  (printout ?animal " is a mammal "))

(defrule identify2
  (?animal gives milk)
->
  (assert (?animal is a mammal))
  (printout ?animal " is a mammal "))

(defrule identify3
  (?animal has feathers)
->
  (assert (?animal is a bird))
  (printout ?animal " is a bird "))

(defrule identify4
  (?animal flies)
  (?animal lays eggs)
->
  (assert (?animal is a bird))
  (printout ?animal " is a bird "))

(defrule identify5
  (?animal eats meat)
->
  (assert (?animal is a carnivore))
  (printout ?animal " is a carnivore "))

(defrule identify6
  (?animal has pointed teeth)
  (?animal has claws)
  (?animal has forward eyes)
->
  (assert (?animal is a carnivore))
  (printout ?animal " is a carnivore "))

(defrule identify16
  (?animal is a ?species)
  (?animal is a parent of ?child)
->
  (assert (?child is a ?species))
  (printout ?child " is a " ?species))
```

このプログラムを動かすには、このプログラム (animal.rl) を読み込んだ後、以下のように記述したファクトファイル (animal.fct) を読み込み、 run すれば良い。

(ファクトファイルの記述)

```
(deffacts (robbie has dark spots))
(deffacts (robbie has tawny color))
(deffacts (robbie eats meat))
(deffacts (robbie has hair))
(deffacts (suzie has feathers))
(deffacts (suzie flies well))
```

(実行)

```
> (load "animal.rl") ↓
; loading "animal.rl"
T
> (load "animal.fct") ↓
; loading "animal.fct"
T
> (run) ↓
```

2. 自動車診断エキスパートシステム

自動車の状態から必要な措置を推論するエキスパートシステムを TEST を用いて記述する。

(プログラム 2) 自動車診断エキスパートシステム

```
;;;
;;; 自動車診断エキスパートシステム
;;;                                          coded by sag (94/7/18)

;;; エンジン状態に関するルール

(defrule normal-engine-state-conclusions
  (working-state engine normal)
->
  (assert (spark-state engine normal))
  (assert (fuel-level gas-tank sufficient))
  (assert (charge-state battery charged))
  (assert (rotation-state engine rotates)))

(defrule unsatisfactory-engine-state-conclusions
  (working-state engine unsatisfactory)
->
  (assert (fuel-level gas-tank sufficient))
  (assert (charge-state battery charged))
  (assert (rotation-state engine rotates)))

;;; 質問用ルール
```

```
(defrule determine-engine-state
  (query phase)
  (not (working-state engine ?some-state))
->
  (set ?state (select (("normal" normal)
                       ("unsatisfactory" unsatisfactory)
                       ("does not start" does-not-start))
                      "What is the working state of the engine?"))
  (assert (working-state engine ?state)))

(defrule determine-rotation-state
  (query phase)
  (working-state engine does-not-start)
  (not (rotation-state engine ?some-state))
->
  (ask-if ((assert (rotation-state engine rotates))
           (assert (spark-state engine irregular-spark)))
          ((assert (rotation-state engine does-not-rotate))
           (assert (spark-state engine does-not-spark)))
          "Does the engine rotate?"))

(defrule determine-sluggishness
  (query phase)
  (working-state engine unsatisfactory)
  (not (symptom engine sluggishness))
  (not (symptom engine not-sluggishness))
->
  (ask-if (assert (symptom engine sluggishness))
          (assert (symptom engine not-sluggishness))
          "Is the engine sluggish?"))

(defrule determine-misfiring
  (query phase)
  (working-state engine unsatisfactory)
  (not (symptom engine misfiring))
  (not (symptom engine not-misfiring))
->
  (ask-if ((assert (symptom engine misfiring))
           (assert (spark-state engine irregular-spark)))
          (assert (symptom engine not-misfiring))
          "Does the engine misfire?"))

(defrule determine-knocking
  (query phase)
```

```
  (working-state engine unsatisfactory)
  (not (symptom engine knocking))
  (not (symptom engine not-knocking))
->
  (ask-if (assert (symptom engine knocking))
          (assert (symptom engine not-knocking))
          "Does the engine knock?"))

(defrule determine-low-output
  (query phase)
  (working-state engine unsatisfactory)
  (not (symptom engine low-output))
  (not (symptom engine not-low-output))
->
  (ask-if (assert (symptom engine low-output))
          (assert (symptom engine not-low-output))
          "Is the output of the engine low?"))

(defrule determine-gas-level
  (query phase)
  (working-state engine does-not-start)
  (not (fuel-level gas-tank ?some-level))
->
  (ask-if (assert (fuel-level gas-tank sufficient))
          (assert (fuel-level gas-tank empty))
          "Does the tank have any gas in it?"))

(defrule determine-battery-state
  (query phase)
  (rotation-state engine does-not-rotate)
  (not (charge-state battery ?some-state))
->
  (set ?state (select (("Charged" charged) ("Dead" dead))
              "What is the state of the battery"))
  (assert (charge-state battery ?state)))

(defrule determine-point-surface-state1
  (query phase)
  (working-state engine does-not-start)
  (spark-state engine irregular-spark)
  (not (point-surface-state points ?some-state))
->
  (set ?state (select (("Normal" normal)
                       ("Burned" burned)
                       ("Contaminated" contaminated))
```

```
               "What is the surface state of the points"))
  (assert (point-surface-state points ?state)))

(defrule determine-point-surface-state2
  (query phase)
  (symptom engine low-output)
  (not (point-surface-state points ?some-state))
->
  (set ?state (select (("Normal" normal)
                       ("Burned" burned)
                       ("Contaminated" contaminated))
               "What is the surface state of the points"))
  (assert (point-surface-state points ?state)))

(defrule determine-conductivity-test
  (query phase)
  (working-state engine does-not-start)
  (spark-state engine does-not-spark)
  (not (charge-state battery dead))
  (not (conductivity-test ignition-coil ?some-result))
->
  (set ?result (select (("Positive" positive)
                        ("Negative" negative))
                "What is conductivity test for the ignition coil"))
  (assert (conductivity-test ignition-coil ?result)))

;;; 修理診断ルール

(defrule no-repair-needed
  (working-state engine normal)
->
  (assert (repair No-repair-needed)))

(defrule charge-battery-repair
  (rotation-state engine does-not-rotate)
  (charge-state battery dead)
->
  (assert (repair Charge-the-battery)))

(defrule timing-adjustment-repair
  (working-state engine unsatisfactory)
  (symptom engine knocking)
->
  (assert (repair Timing-adjustment)))
```

```
(defrule replace-ignition-coil-repair
  (working-state engine does-not-start)
  (spark-state engine does-not-spark)
  (conductivity-test ignition-coil positive)
->
  (assert (repair Repair-the-distributor-lead-wire)))

(defrule point-gap-adjustment-repair
  (working-state engine unsatisfactory)
  (symptom engine misfiring)
->
  (assert (repair Point-gap-adjustment)))

(defrule replace-points-repair
  (working-state engine does-not-start)
  (spark-state engine irregular-spark)
  (point-surface-state points burned)
->
  (assert (repair Replace-the-points)))

(defrule clean-points-repair-1
  (working-state engine does-not-start)
  (spark-state engine irregular-spark)
  (point-surface-state points contaminated)
->
  (assert (repair Clean-the-points)))

(defrule clean-points-repair-2
  (working-state engine unsatisfactory)
  (symptom engine low-output)
  (point-surface-state points contaminated)
->
  (assert (repair Clean-the-points)))

(defrule clean-fuel-line-repair
  (symptom engine sluggishness)
->
  (assert (repair Clean-the-fuel-line)))

(defrule add-gas-repair
  (rotation-state engine rotates)
  (fuel-level gas-tank empty)
->
  (assert (repair Add-gas)))
```

```
;;; 診断結果表示ルール

(defrule no-repairs
  (list-repairs)
  (not (repair ?some-item))
->
  (assert (repair "Take your cat to a mechanic")))

(defrule print-repair
  (list-repairs)
  (repair ?item)
->
  (printout ?item))

(defrule spaces-at-end
  (list-repairs)
->
  (printout))

;;; フェーズ切替えルール

(defrule system-banner
  (not (wrote-banner))
->
  (assert (wrote-banner))
  (printout)
  (printout "The Engine Diagnosis Expert System")
  (printout))

(defrule initiate-query
  (not (query phase))
->
  (assert (query phase)))

(defrule initiate-repair
  (query phase)
  (repair ?some-item)
->
  (assert (list-repairs)))
```

3. ワイン選択エキスパートシステム

   料理とユーザの好みを尋ねて、適切なワインを選択するエキスパートシステムを TEST を用いて記述する。選択の方針は以下の通り。

   (a) 肉料理には赤、魚介料理には白、鳥料理ならユーザの好みに合わせる。

(b) 料理から選んだものとユーザの好みが一致しない時は前者を優先する。

(プログラム 3) ワイン選択エキスパートシステム

```
;;;
;;; ワイン選択エキスパートシステム
;;;                                    coded by sag (94/1/12)

;;; 料理との組合せ

;;; 肉料理には赤
(defrule choose-color-for-meat
  (choose-qualities)
  (main-component meat)
->
  (assert (best-color red)))

;;; 魚料理には白
(defrule choose-color-for-fish
  (choose-qualities)
  (main-component fish)
->
  (assert (best-color white)))


;;; 鳥料理には赤白どちらでも良い
(defrule choose-color-for-poultry
  (choose-qualities)
  (main-component poultry)
->
  (assert (best-color white))
  (assert (best-color red)))

;;; 好みと料理との相性チェック

;;; 一致する時は問題なし
(defrule prefered-color-may-be-recommended
  (recommended-qualities)
  (prefered-color ?color)
  (best-color ?color)
->
  (assert (recommended-color ?color)))

;;; 一致しないときは料理との相性を優先
(defrule best-color-always-recommended
  (recommended-qualities)
```

```
  (not (recommended-color ?some-color))
  (best-color ?color)
->
  (assert (recommended-color ?color)))

;;; 質問用ルール

(defrule question1
  (not (main-component ?component))
->
  (set ?component (select (("肉料理" meat)
                           ("魚介料理" fish)
                           ("鳥料理" poultry))
                          "今日のお食事は？"))
  (assert (main-component ?component)))

(defrule question2
  (not (prefered-color ?color))
->
  (set ?color (select (("赤ワイン" red)
                       ("白ワイン" white))
                      "普段お好きなワインは？"))
  (assert (prefered-color ?color)))

;;; 結果の表示

(defrule print-color
  (print-color)
  (not (printed-color))
  (recommended-color ?color)
->
  (printout ?color "がおすすめです。")
  (assert (printed-color)))

;;; フェーズの切替え用ルール
;;; フェーズ０：質問
;;; フェーズ１：料理との相性
;;; フェーズ２：好みと料理との相性のチェック
;;; フェーズ３：結果の表示

(defrule change-to-phase-1
  (not (choose-qualities))
->
  (assert (choose-qualities)))
```

```
(defrule change-to-phase-2
  (choose-qualities)
  (not (recommended-qualities))
->
  (assert (recommended-qualities)))

(defrule change-to-phase-3
  (choose-qualities)
  (recommended-qualities)
  (not (print-color))
->
  (assert (print-color)))
```

3 実験事項

1. (プログラム 1) を実行せよ。ルールファイル (animal.rl) およびファクトファイル (animal.fct) はディスクに入っている。ステップ動作をさせ、ルールのマッチング、ワーキングメモリの変化を観察し、プログラムの動作を理解せよ。

2. (プログラム 1) にルール 7 から 15 を追加せよ。

3. (プログラム 2) を実行せよ。ルールファイル (engine.rl) はディスクに入っている。ステップ動作をさせ、ルールのマッチング、ワーキングメモリの変化を観察し、プログラムの動作を理解せよ。

4. (プログラム 2) を自由に拡張せよ。もしくは (プログラム 2) を参考に簡単な機械の故障診断を行なうエキスパートシステムを作成せよ。

5. (プログラム 3) を実行せよ。ルールファイル (wine.rl) はディスクに入っている。ステップ動作をさせ、ルールのマッチング、ワーキングメモリの変化を観察し、プログラムの動作を理解せよ。

6. (プログラム 3) をユーザが好みがわからない場合も扱えるよう拡張せよ。その場合の選択方針は各自考えよ。

7. (プログラム 3) を拡張して、ボディ (ライト、ミディアム、フル) と甘さ (スイート、ミディアム、ドライ) も判定して色、ボディ、甘さの組合せから最適なワインの種類を選択できるようにせよ。判定材料として新たに料理の味つけ (薄め、普通、濃い)、ソース (スパイシー、甘い、クリーム、トマト) の種類が入る。判定規則は表 2を参考にせよ。なお、ルールが矛盾もしくは最終的に一意に決まらない時にどうするかは各自の判断に任せる。

8. 各自の趣味または業務に関係する簡単なエキスパートシステムを設計し、構築せよ。ルール数は 20 から 50 程度とする。

4 検討事項

1. 今回はルール型の知識のみ用いたが、その他の知識表現法について簡単に述べよ。

2. エキスパートシステムの問題点および構築の際注意すべき点について考察せよ。

3. 今回用いた構築ツールに関する要望があればまとめよ。

4. 本実験の感想を述べよ。

表 2: ワイン選定規則

(1) 味つけ

味つけ	ボディ
薄め	ライト
普通	ライト、ミディアム、フル
濃い	ミディアム、フル

(2) ソース

ソース	色	ボディ	甘み
スパイシー		フル	
甘い		軽め、ミディアム、重め	スイート、ミディアム
クリーム	白 (何料理かわからない場合)	ミディアム、フル	
トマト	赤 (魚介料理以外)		

(3) ワインの種類

種類	色	ボディ	甘み
Gamay	赤	ミディアム	スイートまたはミディアム
Chablis	白	ライト	ドライ
Sauvignon-Blanc	白	ミディアム	ドライ
Chardonnay	白	ミディアムまたはフル	ミディアムまたはドライ
Soave	白	ライト	ミディアムまたはドライ
Riesling	白	ライトまたはミディアム	スイートまたはミディアム
Geverztraminer	白	フル	
Chenin-Blanc	白	ライト	スイートまたはミディアム
Valpolicella	赤	ライト	
Cabernet-Sauvignon	赤		ミディアムまたはドライ
Zinfandel	赤		ミディアムまたはドライ
Pinot-Noir	赤	ミディアム	ミディアム
Burgundy	赤	フル	