推論と自律エージェント(つづき)

新出尚之(生活環境学部)

2025年9月10日

本日の内容

- 自律エージェントと推論
- Prolog言語(前回)のつづき

(参考)G401室の無線LAN

お手元のPCを使いたい場合はご利用ください

- SSID: `ICS`
- ユーザ名: 「アカウント通知書」記載のログインID
  (`lics`で始まる8文字)
- パスワード: 「アカウント通知書」記載のパスワード
- 以下は入力を要求される場合のみ入力 (聞かれない場合は入力不要)
  - ドメイン名: `e.ics.nara-wu.ac.jp`
  - セキュリティ: WPA/WPA2-Enterprise
  - 認証: 保護つきEAP (PEAP)
  - 内部認証: MSCHAPv2

(参考)アプリで利用するSWI-Prolog

アプリをインストールすれば、ブラウザでネットに繋がなくてもSWI-Prologを利用できます

1. `https://www.swi-prolog.org/download/stable` から自分のPCのOSに合うものを選んでインストール
   - インストール画面で「Add swipl to the system PATH for all users」にチェックを入れるのがおすすめ
   - インストール画面で「Create swipl Desktop Icon」にはチェックを入れる
   - OSがLinuxの場合は、ディストリビューションが提供するパッケージからのインストールがおすすめ
2. SWI-Prologのアイコンができ、それをクリックでアプリ版のSWI-Prologが起動(SWI-Prologのコンソールウィンドウが開く)

アプリで利用するSWI-Prolog(continued)

3. コンソールウィンドウでプログラムの作成
   - File→Newでプログラムの新規作成
   - File→Editで以前に作成したプログラムの再編集
   - どちらの場合も編集用ウィンドウが別に開く
4. プログラムを作成したら、編集用ウィンドウからCompile→Compile Bufferでプログラムを読み込ませる
5. コンソールウィンドウの「?-」のところにゴールを入力、Enterキーで実行

自律エージェントと推論

時間の都合で、演習は省いて、説明だけにします

- 人間の行為決定のモデル
  - ⋆ 信念(回りの状況に関する知識)
  - ⋆ 目標(何かを達成したい)
  - ⋆ 意図(目標を達成するために、ある行為をとろうとする)
  - ⋆ プラン(条件Aが成り立っているとき、行為Bをとれば、目標Cが達成できる)
- 自律エージェント
  - ⋆ 自身の目標を持ち、それを達成する方法を**自身で決定**して行為を決める、プログラムやロボットなど

自律エージェントと推論(continued)

- 実践的推論
  - ⋆ **行為**に関する推論
    - ∗ これに対し、通常の推論は「何かが成り立つかどうかに関する推論」
  - ⋆ Cを目標に持っていて、信念から推論によってAが導け、プラン「条件Aが成り立っているとき、行為Bをとれば、目標Cが達成できる」を持っていれば、「そのプランに従って行為Bをする」という意図を持てる
    - ∗ 例えば「昼飯食べたい」という目標があって、「お金を持っている」という信念があって、プラン「お金があれば食堂に行けば昼飯が食べられる」があれば、「食堂に行く」という意図を持てる
  - ⋆ 推論の部分は、Prologと同様の推論機構で実現可能
  - ⋆ **自律エージェント**は実践的推論によって行為を決める

自律エージェントと推論(continued)

自律エージェント構築向けに、Prologとよく似た考え方(論理型言語)のプログラミング言語「Jason」がある

- エージェント(意図を持って行動する主体)1つごとに1つのプログラムを書く
- エージェントプログラムは、プランと、信念に関する推論ルールからなる
- 各エージェントはプログラムにより「実践的推論」を行って、自分の行為を決める

前回資料の補足

中置記法

- 一部の2引数の組み込み述語は、述語名を2つの引数の間に置いた形でも書ける。これを「中置記法」と呼ぶ
- 該当するのはis述語(前回資料p.35)、数の比較述語(「<=」「>」など。同p.43)、=述語(同p.45)など
- 例えば、「`is(Y1, Y-1)`」(同p.35)は「`Y1 is Y-1`」とも書くことができ、こう書いても同じ意味
- サンプルプログラム(同p.42)の中には、このことを知らないと読めないものがある

Prolog言語

ここからは前回(前回資料p.34まで)の続きになります

Prolog処理系の仕組みを復習

前回資料p.23が基本

- 前回資料例1(p.24)
- 前回資料例2(p.25, 図p.26)
- 前回資料例3(p.27～28, 図p.29)
- 前回資料例4(p.30)
- 前回資料例5(p.31～33, 図p.34)

与えたゴールを導けるような事実や規則を、プログラムの中から探す。
見つかったものが規則なら、その規則の体部を導けるかどうか調べる。

Prolog処理系の仕組みを復習(continued)

(前回資料p.23で説明した)Prologの実行の仕組みを1ステップずつ見せてくれる「トレース」という機能がある[1](デバッグにも使える)

1. プログラムとゴールを(これまでと同様に)入力
2. 実行開始前にまず画面下「Solution」→「Debug(trace)」をクリック
3. 実行開始すると、矢印アイコンがいくつか出てくる
   - とりあえず、黄色の矢印のうち左2つを理解すればOK
   - 黄色の一番左は「1ステップ進む」
   - 黄色の左から2番目は「解が1つ出るまで進む」
4. 解が出たときはこれまでと同様「Next」ボタンが出るので「Next」をクリック
   (「Next」ボタンが出ない場合は最後の解が出てプログラムが終了している)

[1]アプリ版SWI-Prologの場合は、traceという組み込み述語で同様のことができる(詳細略)。

Prolog処理系の仕組みを復習(continued)

Prologプログラムの実行過程を追いかけるもう1つの方法に「デバッグwrite」がある。プログラムの要所で、write述語とnl述語(前回資料p.46)を使ってわざと何かを出力する方法[2]。

例えば前回資料p.31のプログラムのson述語の定義を

```
son(X,Y):-
    write([X,Y]), nl,    /* ←このwriteとnlを追加 */
    parent(X,Y),
    male(Y).
```

とすると、実行結果に影響は出ないが、son述語が呼び出されるたびに、引数のX, Yの値が出力される(XやYが未代入でson述語が呼び出された場合は、変数として出力される)。

[2]write述語は1引数なので、一度にいくつも表示させたい場合は、引数をリストにする必要がある。

サザエさんプログラム·改

サザエさんのプログラムを書き直してみよう[3]。今度は、parentとmaleを使ってfatherを定義するのでなく、逆にfatherやmotherを事実として与えて、それらを使ってparentを定義してみる。さらに、サザエさん一家の情報(性別、父子·母子関係の事実)を全て含めてみよう。

```
male(namihei).
male(masuo).
male(katsuo).
male(tara).
female(fune).
female(sazae).
female(wakame).
```

(次ページへ続く)

[3]前回までのプログラムを修正するのではなく、SWISHで新しいタブを開いて、新しいプログラムとして書こう。

```
father(namihei,sazae).
father(namihei,katsuo).
father(namihei,wakame).
father(masuo,tara).
mother(fune,sazae).
mother(fune,katsuo).
mother(fune,wakame).
mother(sazae,tara).
parent(X,Y):-
    father(X,Y).
parent(X,Y):-
    mother(X,Y).
child(X,Y):-
    parent(Y,X).
son(X,Y):-
    child(X,Y), male(X).
daughter(X,Y):-
    child(X,Y), female(X).
```

サザエさんプログラム・改(continued)

ゴールをいくつか試してみよう。下記で、「?–」の部分は入力しないことに注意。

```
?- parent(fune,wakame).
?- parent(X,sazae).
?- parent(fune,X).
?- parent(X,Y).
?- child(X,Y).
```

など。

これらの質問にPrologが答える過程を、前回の例を参考にして考えてみよう。複数の答が出る場合、答はなぜその順番に出るのだろうか。(トレース機能で調べてみよう)

parentの定義は2つの節に分かれているのに、sonやdautherの定義は2つの節に分かれていないのはなぜだろうか。

祖父母の定義

追加する部分のみ記載。

```
grandparent(X,Y):-
    parent(X,Z),
    parent(Z,Y).
```

XがZの親で、ZがYの親なら、XはYの祖父母である。このとき、Zは何でもよい。

- `?- grandparent(namihei,tara).` と質問してみよう。

兄弟姉妹の定義

追加する部分のみ記載。

```
sibling(X,Y):-
    parent(Z,X),
    parent(Z,Y).
```

ZがXの親でもYの親でもあれば、XとYは兄弟姉妹である。

- ?- sibling(sazae,katsuo). と質問してみよう。
- ?- sibling(sazae,X). に対し、X = sazae と X = katsuo と X = wakameが2度ずつ出るのはなぜだろうか。

兄弟姉妹の定義(continued)

前ページの定義では、自分自身も兄弟姉妹に含まれてしまう。自分自身は兄弟姉妹に含めないようにしてみよう。

前ページのsiblingの節を**削除して**、代わりにこれを追加する。なお、「\」(バックスラッシュ)[4]は一部の環境では「¥」と書く(よって「\+」は「¥+」と書く)。「\+」と「=」の間は空けること。

```
sibling(X,Y):-
    parent(Z,X),
    parent(Z,Y),
    \+ =(X,Y).
```

ZがXとYの親で、XとYが異なるなら、XはYの兄弟姉妹である。

- `?- sibling(sazae,sazae).` と質問してみよう。

[4]バックスラッシュ「\」とスラッシュ「/」を混同しないこと。

兄弟姉妹の定義(continued)

「\+」は“否定”を、=(X,Y)は「XとYが単一化できる」を表す。従って「\+ =(X,Y)」で「XとYが単一化できない」つまり「XとYが異なる」を表す。

「\+ =(X,Y)」の代わりに「\=(X,Y)」と書いても同じ意味になる。

注: 否定(\+)は節の頭部には使えない。例えば

```
\+ male(X) :- female(X).
```

のようなことは書けない。

否定を使う場合の注意事項(授業では省略予定)

Prologは、否定「\+」のついたゴール\+Gが導けるかどうか調べるには、次のようにする。

- 否定を取り除いたゴールGが導けるかどうか調べる
- Gが導けないなら、\+Gは導けると判定。Gが導けるなら、\+Gは導けないと判定。

これを「失敗による否定」という。

この性質から、Gの中の変数が未代入の状態で\+Gが導けるかどうか調べようとすると、期待に反する動作をすることがある。例えば…

否定を使う場合の注意事項(continued)

siglingの定義を、このように変更してみる。

```
sibling(X,Y):-
    \+ =(X,Y),
    parent(Z,X),
    parent(Z,Y).
```

すると、`?- sibling(sazae,X).`と質問しても、単にfalseと出るだけで、答を返さなくなる。

なぜなら、ゴール`sibling(sazae,X)`とこの節の頭部が単一化され、

```
\+ =(sazae,X), parent(Z,sazae), parent(Z,X).
```

という連言ゴールを導こうとするが、その最初のゴール`\+ =(sazae,X)`は**導けない**からである。それがなぜかというと、否定を取り除いたゴール`=(sazae, X)`は、XとsazaeをXを単一化できるので、導けるからである。

否定を使う場合の注意事項(continued)

このような現象を防ぐため、節の体部の中で否定を使う場合は、否定のついたゴールを導こうとする段階で、そのゴールに未代入の変数が残らないようにする。

例えば、兄弟姉妹の定義に使った

```
sibling(X,Y):- parent(Z,X), parent(Z,Y), \+ =(X,Y).
```

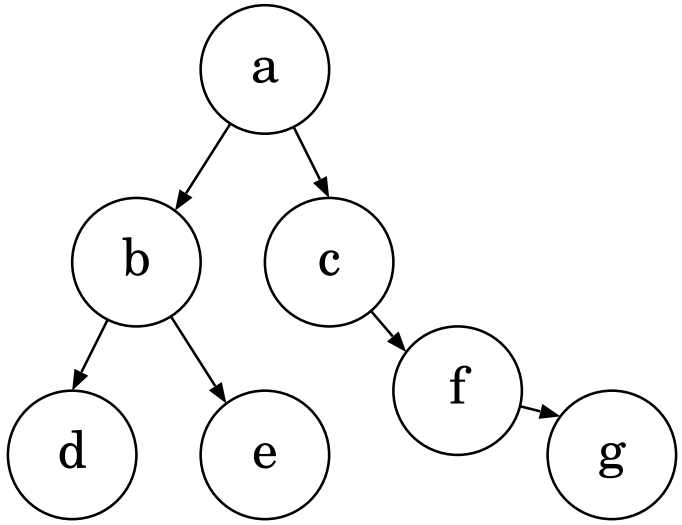
という節で、否定を体部の最後に持っていった理由は、そうすることで、否定付きゴールを導けるかどうか試すのをできるだけ後回しにし、それまでに変数が代入済みになるようにするためである。

(ここまで、授業では省略予定)

到達可能関係

ここからはサザエさんとは別なプログラムなので、新しいプログラムとして作ろう。

右図のように、ノード(円)とエッジ(矢印)があり、ノードからノードへはエッジの向きにしかたどれないとする。どこにエッジがあるかは与えられるものとして、ノードからノードへたどれるかどうか判定するプログラムを書こう。どのノードからも、自分自身へは(エッジ0回で)たどれるものとする。



到達可能関係(continued)

```
edge(a,b).
edge(b,d).
edge(b,e).
edge(a,c).
edge(c,f).
edge(f,g).

reachable(X,X).
reachable(X,Y):-
    edge(X,Z),
    reachable(Z,Y).
```

- 自分自身へはたどれる
- XからZへのエッジがあり、ZからYへたどれるなら、XからYへたどれる(再帰的定義)

到達可能関係(continued)

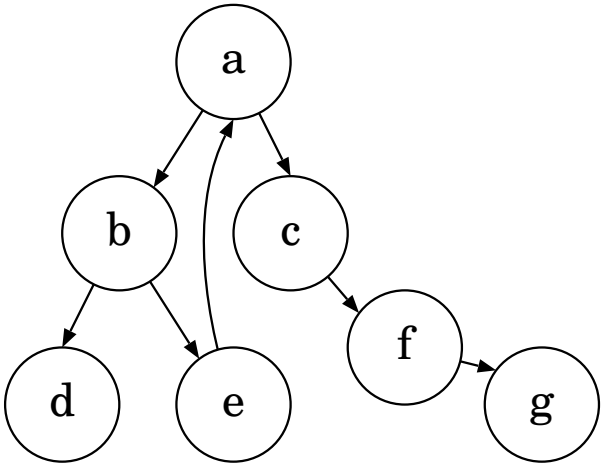
以下のようなゴールを試してみよう。

```
?- reachable(a,g).
?- reachable(b,c).
?- reachable(a,X).
?- reachable(X,Y).
```

最後のゴールに対しては、最初の答として「`X = Y`」というものが出る。これは「X=Yであれば、XやYの具体的な値がわからなくても、reachable(X,Y)はプログラムから導ける」ということ。

到達可能関係(continued)

このプログラムは残念ながら、下図のようにエッジのループがあると破綻する。



先のプログラムに`edge(e,a).`を加えて[5]、ゴール`?-reachable(a,d).`や、`?-reachable(a,c).`を試してみよう。正常に停止する場合とそうでない場合[6]があるのはなぜだろうか。

[5] このとき、他のedgeの事実と離さないようにすること。

[6] SWISHで無限ループに陥った場合、abortをクリックすれば止まる。

到達可能関係(continued)

破綻するのは、a→b→e→a→…のように同じところを回り続けるループに陥るからである。これを回避するには、通ったところを記録しておいて、同じところに戻ってきたらそれ以上進まないようにするしかない。通ったところを記録するには、リスト(前回資料p.39)を使う。

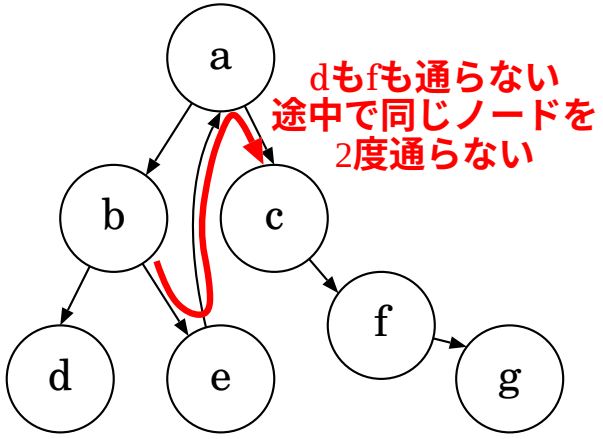
まず、reachable_noloopという3引数の述語を以下のように作る。

- 第1引数のノードから、第3引数のノードまで
- 第2引数のリストの要素であるノードを通らずに
- しかも同じノードを2度通らずに

行けるならばtrue。例えば、

```
?- reachable_noloop(b,[d,f],c).
```

というゴールはtrueになるように作る。



到達可能関係(continued)

reachable_noloopという述語は、以下のように考えて作る。

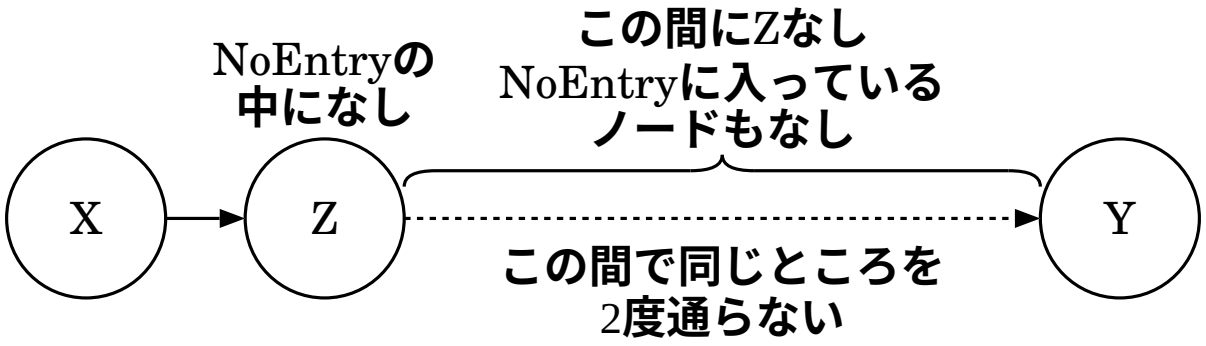
- ノードXからノードX自身へは、いかなるノードも通らず、同じノードを2度通ることもなく、行ける。
  従って、`reachable_noloop(X,_L,X)`は`X`や`_L`の値にかかわらず必ず導けなければならないので、これを事実として用意[7]

次ページへ

[7]変数名`_L`が「_」で始まっている理由は、前回資料p. 38。

到達可能関係(continued)

- ノードのリストNoEntryがあるとして、ノードXからノードYへ、NoEntryに入っているノードを通らず、しかも同じノードを2度通らずに行けるには、次の条件を全て満たせばよい。
  - ⋆ Xからあるノードzへのエッジがある
  - ⋆ ZはリストNoEntryに入っていない
  - ⋆ ZからYへ、NoEntryに入っているノードもZも通ることなく、かつ同じノートを2度通らずに行ける



到達可能関係(continued)

このうち「ZはリストNoEntryに入っていない」の部分は、「ZはリストNoEntryに入っている」の否定でよい。「ZはリストNoEntryに入っている」の判定は、組み込み述語memberでできる。

member述語は以下のように使う。第1引数が第2引数のリストの要素であればtrueになる。組み込み述語なので、プログラムを作らずにいきなりゴールを与えても解が求まる。

ゴール	結果
`?- member(a,[b,c,d]).`	`false`
`?- member(a,[b,a,d]).`	`true`
`?- \+ member(a,[b,c,d]).`	`true`
`?- \+ member(a,[b,a,d]).`	`false`

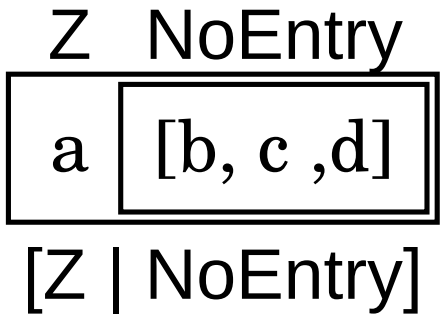
これにより、「ZはリストNoEntryに入っていない」の判定は
`\+ member(Z,NoEntry)`でよいことになる。

到達可能関係(continued)

「NoEntryに入っているノードもZも通ることなく」の部分を実現するには、

> リストNoEntryの先頭に要素Zを追加したリストを[Z | NoEntry]で表現できる

ことを使えばよい(前回資料p.39)。



これにより、「[Z | NoEntry]に入っているノードを通ることなく」と考えればよいので、[Z | NoEntry]を第2引数としてreachable_noloopを再帰呼び出しすることになる。

到達可能関係(continued)

reachable_noloopができれば、2引数のreachable述語は次のように考えて作ればよい。

ノードXからYまで到達できるには、

- ノードXからYまで
- ノードXを通らず、途中で同じところを2度通らずに

到達できればよい。

この条件の部分は、reachable_noloop述語を使って

```
reachable_noloop(X,[X],Y)
```

と書ける。「ノードXを通らず」としているのは、Xから1周してXに戻ってくる経路を排除するため(Xからはノード0個でXへ行けるので、Xから1周してXに戻ることを考慮する必要はない)。

到達可能関係(continued)

以上から、プログラム全体はこのようになる。

```
edge(a,b).
edge(b,d).
edge(b,e).
edge(a,c).
edge(c,f).
edge(f,g).
edge(e,a).

reachable(X,Y):-reachable_noloop(X,[X],Y).

reachable_noloop(X,_L,X).
reachable_noloop(X,NoEntry,Y):-
  edge(X,Z),
  \+ member(Z,NoEntry),
  reachable_noloop(Z,[Z|NoEntry],Y).
```

Prologプログラムのまとめ

まとめると、Prolog言語では

- 「物と物の間の関係」に関する事実や規則をプログラムとして書き
- それらの事実や規則から導けることを推論させることができる

そして、推論は以下のように行われている。

- ゴールを与え、そのゴールを導けるプログラム中の事実や規則を探す
- 見つかったものが規則であれば、その体部を導けるかどうか調べる

Prologプログラムの問題点(continued)

事実や規則を探す過程は網羅的に、かつ一定の順序で行われるので

- ループにはまると、ほしい解が出ない場合がある
- ループにはまらなくても、事実や規則を1つ1つ探すので、事実や規則が非常に多いと遅くなる場合がある

Prologで実用的なプログラムを書こうとすると、それらの問題が起きないように。プログラマが調整する必要がある(例えば先述の到達可能関係プログラムのように)

考えてみよう

- 簡単なプログラムを考えて作ってみよう
  - ⋆ もっといろいろな家族関係(おじ/おば、甥/姪など)を調べるプログラム
  - ⋆ サザエさんプログラムのように、日常に出てくるいろいろな「関係」を記述したプログラム
  - ⋆ 階乗を計算するプログラム(参考: 前回資料p.37)
    - ∗ その他、より複雑な再帰的定義で定義される関数を計算するプログラム
  - ⋆ 到達可能性判定プログラムの改良
    - ∗ 途中の経路も出すようにできるか
    - ∗ さらに、各エッジの距離を与えておいて、経路の長さを出すようにできるか

考えてみよう(continued)

- Prologでできる推論と、人が行う推論の間には、どのようなギャップがあるだろうか。また、そのギャップのうち、工夫によって埋められるものはあるだろうか。

課題

p.36, p.37の「考えてみよう」(2つとも)について、自由にまとめてみてください。

p.36については、作成したプログラム(と、その動かし方、何のプログラムか)もレポートに記載してください。サザエさんプログラムのような簡単なものでも十分です。(高度ですが)サンプルプログラム(前回資料p.42)の改造にチャレンジするのでもかまいません。