artisoc（旧 JAVA 版 KK-MAS）チュートリアル

2006 年 03 月 12 日
2006 年 05 月 05 日更新

人工社会の作り方：

マルチエージェント・シミュレーションへの誘い

東京大学大学院総合文化研究科・山影進研究室
株式会社構造計画研究所・創造工学部

JAVA 版 KK-MAS は、一般リリースに先立ち、
artisoc(「アーティソック」ないし「アルティソック」)
という名称になることが決まりました。
一層身近な MAS として、ますます活用いただけることを
期待しております。

このチュートリアルでは、私たちが開発したシミュレータを実際に動かすことを通じて、マルチエージェント・シミュレーションの基本を学びます。使い方に少しずつ慣れていきながら、シミュレーションするためのモデルをいくつも作り、実際に実行してみます。簡単なモデルをいくつか作ることをとおして、マルチエージェント・シミュレーションの基本的な考え方を会得してもらうことを目指しています。

たった半日のチュートリアルですが、必ず自力でモデルを作れるようになります。そして、独学でさらに高度な技法を身につけることが容易になります。このチュートリアルが終われば、そこはもうマルチエージェント・シミュレーションの世界の中です。

人工社会の作り方：

マルチエージェント・シミュレーションへの誘い

はじめに

マルチエージェント・シミュレーションの世界へようこそ！

マルチエージェント・シミュレーションは社会現象の分析にとってきわめて有望な技法です。欧米の社会科学の学界では、注目され始めています。しかし日本では、理工系の学問分野では急速に蓄積されつつあるものの、社会科学ではまだまだマルチエージェント・シミュレーションに対する理解が不足しています。

なぜでしょうか。十分に理解されていない理由は次のようにまとめることができるでしょう。

×参考になる具体的な適用事例が乏しい
×プログラミング言語やプログラミング技法を学びたくない
×社会科学の方法としては、シミュレーションはうさんくさい
×マルチエージェント・シミュレーションの先達（せんだつ）がいない

これらは皆もっともな理由です。そして、どれも互いに結びついています。したがって、このままでは、マルチエージェント・シミュレーションは社会科学になかなか浸透していかないでしょう。

そこで、私たちはマルチエージェント・シミュレーションが社会現象の分析に広範に使われるようになるように、次のことをめざしています。

◎ 参考になる具体的な適用事例を示す
◎ プログラミング言語やプログラミング技法を学ぶ必要がない
◎ 社会科学の方法として学界に認めさせる
◎ マルチエージェント・シミュレーションの先達をふやす

上のようなことを実現するには、何よりも、マルチエージェント・シミュレーションを簡単に実行できるソフトウェアが必要です。そこで、マルチエージェント・シミュレータの開発を進めることにしました。

この度、ようやくプログラミング言語やプログラミング技法を学ばないでも利用できるシミュレータができました。このことにより、簡単にマルチエージェント・シミュレーションを行える環境ができました。今までマルチエージェント・シミュレーションを敬遠してきた人たちにとっても、マルチエージェント・シミュレーションの世界を覗いてみることが容易になりました。

このシミュレータを使って、マルチエージェント・シミュレーションをするモデルも簡単に作れるようになりました。簡単なモデルなら５分で完成させ、ただちに実行することが可能です。大げさではありません。

ここで作ってもらうモデルはどれも簡単です。しかし、モデル作りの中で学ぶ技法だけを用いて、２００５年ノーベル経済学賞を受賞したトマス・シェリングが考案した「分居モデル」を完成させることができます。それどころか、もっと洗練したモデルにすることも可能です。

「習うより慣れよ」ということわざがあります。その趣旨を活かしながら、ここでは同じ「ならう」でも、「習う」よりは「倣う」の方を重視して、「倣うことで慣れよ」というふうに読み替えたいと思います。つまり、実際にモデル作りを例題に沿って真似していくうちに、自分でモデルを作れるくらいにシミュレータに「慣れていき」、そしてマルチエージェント・シミュレーションについて「分かった」という実感を得ることをめざしています。

目次

第１章　シミュレーションの準備

◎ モデルを作るための枠組み（土台）の設定
◎ 実行過程を見るための出力画面の設定

１．１　はじめに

マルチエージェント・シミュレーションを実際にやってみるには、少なくとも、次のようなプロセスが必要です。

シミュレーションするためのモデルを作る
モデルを実際に動かす　　（つまりシミュレーションの実行）
実行の過程・結果を見る　（つまりシミュレーションの出力）

これに対応して、次のような準備作業が必要です。

モデルを作るための枠組み（土台）の設定
シミュレーションを実行させる環境の設定
実行過程を見るための出力画面の設定

artisoc では実行ボタンさえ押せば、シミュレーションが実行されます。その意味で、実行はとても簡単です。しかし、ここでうっかり忘れがちなのが、シミュレーションの出力です。これを設定しておかないと、実行されても、それを「見る」ことができません。

そこで、この章では、

モデルを作るための枠組み（土台）の設定
実行過程を見るための出力画面の設定

を学びます。

なお、シミュレーションを実行させる環境の設定は、取り敢えず、artisoc が予め設定してある条件をそのまま使うので、わざわざ設定する必要はありません。

１．２　シミュレーションをするということの全体像


artisoc を早く動かしたい人はとばしても構いません。

マルチエージェント・シミュレーションのためのモデルは、

（１）　「エージェント」と呼ばれる行動主体
（２）　個々のエージェントの属性（性質や役割）を表す「変数」
（３）　エージェントが行動する（他のエージェントと関係する）「ルール」
（４）　エージェントが行動する「空間」ないし「場」
（５）　エージェントたちの全体的な状態を知りたいときに設定する「マクロな変数」
（６）　モデル全体に関わる「マクロなルール」

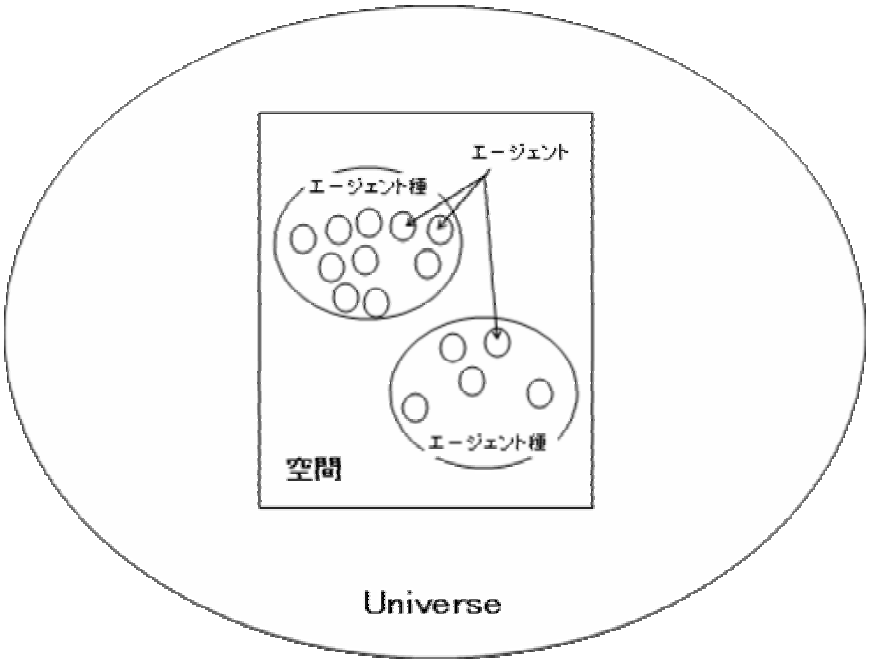
などからなります。

artisoc でモデルを作るための枠組みは、上のようなモデル作りに必要な要素を次のようにまとめています。

（１）　Universe と呼ばれる「全体」-「空間」–「エージェント種」という３段階からなる「ツリー画面」
（２）　Universe とエージェント種にルールを書き込むための「ルール・エディタ」
（３）　Universe、「空間」、「エージェント種」の各々についての「変数」の指定（エージェント種の「変数」は必須です）

ここで「エージェント種」という言葉が出てきました。artisoc では、複数のエージェントでも同じタイプならば、単一のエージェント種としてまとめて、同じルールを指定することができます。

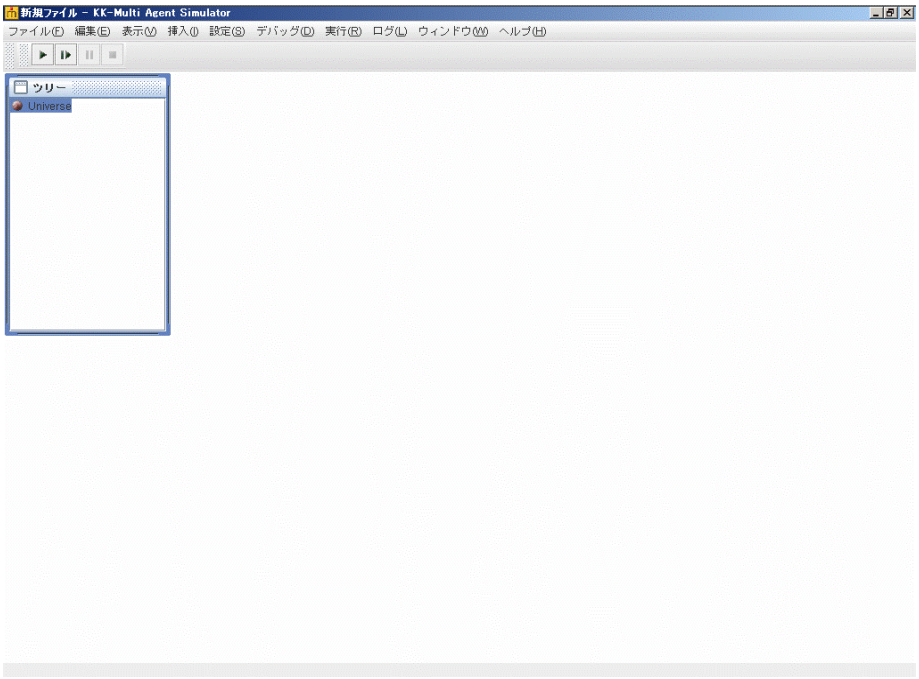
「変数」といっても、それがとる値が数値であるとは限りません。文字列など、さまざまな値をとることができます。変数のとる値を「型」と呼びます。整数値をとる変数は整数型です。



１．３　モデルを作る準備

この章と次章とで鳥が空を飛ぶモデル（「tobutori」）を作って、シミュレーションを実行します。この章では、鳥をとばすまでの準備作業をします。

まず、実際に artisoc を起動してみましょう。すると次のような画面が出てきます。



左上にある箱がツリー画面です。Universeがすでに書き込まれています。

これから、宇宙に神様（創造主）が被創造物を作り出していくというイメージで、具体的なモデルを作っていきます。

新しい空間やエージェントを作っていくのは、「挿入」という操作です。

（1）

カーソルを Universe に重ねてクリックして下さい。フォーカスされている状態（Universe だったら、Universe）となります。この状態で、メニューから「挿入」をクリックし、空間の追加を選んで下さい。（Windows マシンなら Universe を右クリックし、「空間の追加」を選んでもＯＫです。）すると、空間を設定するためのボックス「空間プロパティ」が開きます。空間の名称欄に、ローマ字で oozora と書き込んで下さい。他は、そのまま（予め書き込まれている数字や選択されている選択肢のことをデフォルト値と呼びます）にしておいてください。了解ボタンを押して下さい。これで、ツリーには Universe の下に oozora という空間が設定できました。

空間名や変数名では、日本語の名前を付けても全く構いません。「大空」とか「おおぞら」とかいう名前の空間を作っても問題なく作成できます。開発途上の都合で以下では、空間名や変数名がローマ字表記になっています。

（2）

同様に、oozora とフォーカスされている状態にして、メニューから「挿入」をクリックし、エージェントの追加をクリックして下さい。すると、空間を設定するためのボックス「エージェント・プロパティ」が開きます。エージェント名の欄に、ローマ字で tori と書き込んで下さい。それから、エージェント数の欄に入っている 0（デフォルト値）を、100 にして下さい（0 を delete して、100 と書き込む）。了解ボタンを押して下さい。これで、ツリーには Universe の下の oozora という空間に tori というエージェント種が設定できました。

　ここで注意して欲しいのは、tori という名前がついた１００羽の鳥が、ツリーの中では tori というエージェント種でまとめられている点です。エージェントという言葉は、文脈により、複数のエージェントをまとめたエージェント種を意味したり、個々のエージェントを意味したりするので、気を付けて下さい。

（3）

ツリーの中の tori のすぐ左の小さな○印をクリックして下さい。すると、tori の下に、ID, X,

Y, Layer, Direction という文字が出てくるはずです。これは、エージェントを設定するとシミュレータが自動的に作ってくれるエージェントの４種類の属性変数です。たとえば、X, Y は各々、oozora という空間（デフォルト値により、50x50 に既設定）の X 座標と Y 座標を表しています。ID や Layer についてはとりあえず無視してください。ここでは、１００羽の tori を設定しましたが、１羽毎に ID, X, Y, Layer, Direction が自動的に設定され、これらは個別の値をとることができます。

（４）

ここまでの作業が無駄にならないように、これまでやってきた作業結果を保存しましょう。あるモデル（ここでは、「tobutori」）は、それについてのさまざまな設定を含めて、ひとつのファイルとなっています。この段階では、単に「新規ファイル」という名称になっているはずです。そこで、これに「tobutori」という名前をつけます。まず「ファイル」メニューから「名前を付けて保存」を選択して下さい。tobutori とモデル名を付けて下さい。ここで、tobutori.model と、「.model」が付け加えられています。これは、artisoc でシミュレーションを実行するためのファイルであることを示しています。これを「拡張子」と呼びます。

せっかく作ったモデルが「なにかのひょうし」で消えてしまうと、とても悲しく（虚しく）なります。今後、ファイルは時々「上書き保存」しておく癖を付けましょう。

１．４　シミュレーション結果を見るための準備

次にシミュレーション実行を見るための設定をします。

（１）

メニューの「設定」から「出力設定」を選んで下さい（「設定」をクリックし、いくつかあるメニューの中から「出力設定」をクリックします）。すると、「出力項目リスト」というボックスが開きます。中央下の「追加する出力種類」が「マップ出力」になっている（デフォルト値）のを確認したら、左隅の「追加」ボタンを押して下さい。すると、「マップ出力設定」というボックスが開きます。そこで、「マップ名」に oozora「マップタイトル」に OOZORA と書き込んで下さい。

（２）

「出力項目リスト」の下部にある「マップ要素リスト」の「追加」ボタンをクリックして下さい。「要素設定」というボックスが開きます。「要素名」に tori と書き込んで下さい。

（３）

以上の作業をしたら（他は何もしないでデフォルト値のままにしておいてください）、「要素設定」の「了解」ボタンを押し、「マップ出力設定」ボックスの「マップ要素リスト」に tori が追加されているのを確認し、「マップ出力設定」ボックスの「了解」ボタンを押して下さい。「出力項目リスト」に oozora が追加されているのを確認したら、「了解」ボタンを押して下さい。

以上の作業で、パソコンの artisoc のウィンドウには、最初のツリー画面だけが残っているはずです。

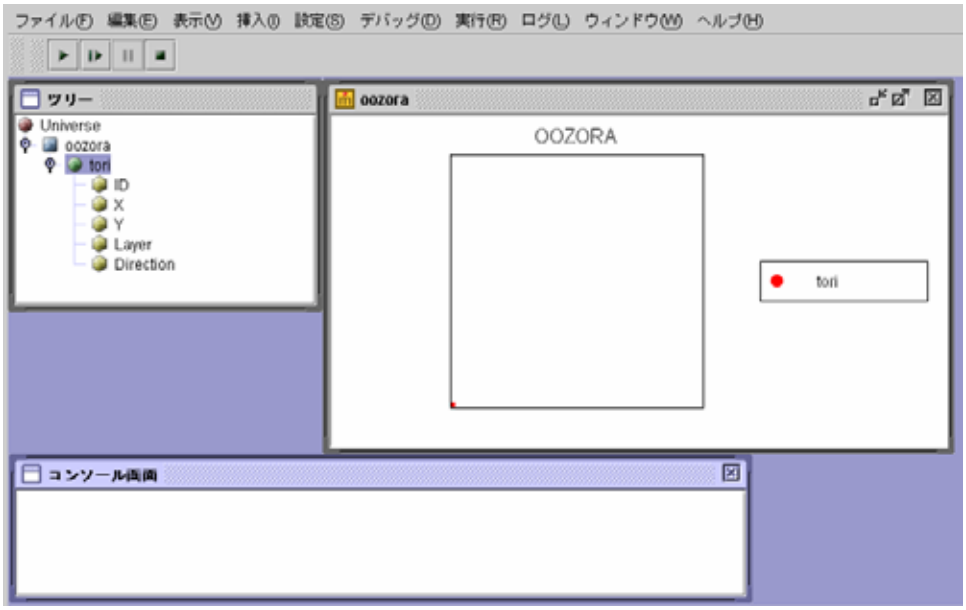
１．５　作業が正しかったかの確認

これでモデル作りのための設定とシミュレーション実行を見るための出力設定は完了です。正しくできたかどうか、確認しましょう。

まず「実行ボタン」を押して下さい。

実行ボタンは、ウィンドウの左上方にあります。実行、ステップ実行、一時停止、停止とボタンが４つ並んでいます。「実行」を押すとモデルが動きます。「ステップ実行」はコマ送りのようにちょっとずつモデルが実行されます。「一時停止」はモデルが一旦止まります。（実行を押すと再開されます。）「停止」を押すとモデルは終了します。一時停止ではなく終了するので気を付けてください。

図のように、OOZORA とタイトルが書かれたマップ（その中には、凡例として、tori があるはずです）とコンソール画面とがあるはずです。（コンソール画面については第６章で説明します）



マップの左下の隅に注目して下さい。小さな赤いドットが見えるはずです。ここに１００羽の tori が重なり合って存在しているのです。ただ、tori がどのように羽ばたくのかについては、まだ何の作業もしていません。ただ、「存在している」だけです。

以上を確認したら、「停止」ボタンを押してください。

以上、第１章

第２章　エージェントを動かす

◎ エージェントの行動ルールを書き込む
◎ 動かすルールの初歩を学ぶ
◎基本的な設定変更のしかたに慣れる

２．１　エージェントが自律的に行動するとはどういうことか？


早く artisoc を動かしたい人はとばしても構いません。
マルチエージェント・シミュレーションの最も基本的なところは、個々のエージェントが自律的に行動するという点です。この章では、前章に続いて、鳥を飛ばすモデルを完成させて、鳥を飛ばせてみましょう。つまり、tori というたくさんのエージェントの一体一体を、自律的に飛ぶという行動をとらせます。

「自律的な行動」については、弱い捉え方と強い捉え方とがあります。

まず弱い捉え方を説明します。他人に何かをさせたいとき、それを正確に指示する必要があります。コンピュータの中でエージェントを動かそうとするときにも、同様なことが言えます。もし、１００羽の tori を動かしたければ、１羽毎に、どのように動くべきかを厳密に指示する必要があります。これに対して、エージェントに弱い意味での自律的行動をとらせるということは、tori に行動の「ルール」を教えておけば、エージェントはそのルールに従って行動するということを意味しています。つまり、モデルを作る人間（つまりわれわれ）は、tori がどこに動くのかを事細かに指定するのでなく、動き方のルールさえ指定すれば良いのです。tori というエージェントは与えられたルールに従って、勝手に行動してくれます。

しかし、この弱い捉え方だと、どのような行動をとるのかエージェントが自律的に選択しているとは限りません。つまり、エージェントに選択の自由（複数の選択肢からどれかひとつを自分の基準で選べること）があるとは限らないのです。そこで、強い捉え方は、ルール自体がエージェントに自分の行動を選択できるようになっている場合のみに限定して自律的行動と呼びます。

たとえば、個々の tori エージェントが、自分の状態（満腹だ/空腹だ）や周囲の状態（仲間が多すぎる/少なすぎる）によって、特定の行動をとる（少し離れて休む/一緒にえさを探しに行く）ようなルールになっているとき、tori は強い意味で自律的行動をとると見なせるでしょう。

この章では、弱い捉え方による自律的行動のルールを設定します。強い捉え方に関しては、第３章および第４章で学びます。

２．２　「動かす」ルールを書き込む

前章で準備した鳥が空を飛ぶモデル（「tobutori」）を完成させて、シミュレーションを実行しましょう。この章の最も大事な課題は、エージェント・ルールを指定することです。

前章の最後で、実行ボタンを押すと、tori が左隅に「存在」してはいましたが、全く動きませんでした。tori を動かすためには、動かすルールが必要です。ここでは、次のような簡単な行動ルールを指定しましょう。

（１）　全ての tori（前章で、１００羽に設定しました）が oozora の中央部（X 座標の値 25,Y 座標の値 25：これを簡潔に(25, 25)と以降表記します）にいる。
（２）　各 tori は、そこから自分の好きな方角を決める。
（３）　各 tori は、毎ステップ、「１」ずつ、その方角に飛び続ける。

artisoc では、座標系は通常の数学での用法と同じになっています。左下が原点(0, 0)で、右上方に進むほど、Ｘ座標もＹ座標も大きくなります。また、角度は基準点から右水平方向を０度とし、左回り（反時計回り）に角度が増えます。真上が９０度です。（座標系については、以上がデフォルトで、左上を原点に設定することも可能です。）

ここで「ステップ」という言葉が出てきました。ステップとは、マルチエージェント・シミュレーションの時刻単位です。時刻や時間は、ふつう、日・時・分・秒という単位で測ります。このような「流れる時間」の測り方に対して、コンピュータのなかで実行するマルチエージェント・シミュレーションでは、全てのエージェントがルールにしたがって１回行動することを１時点

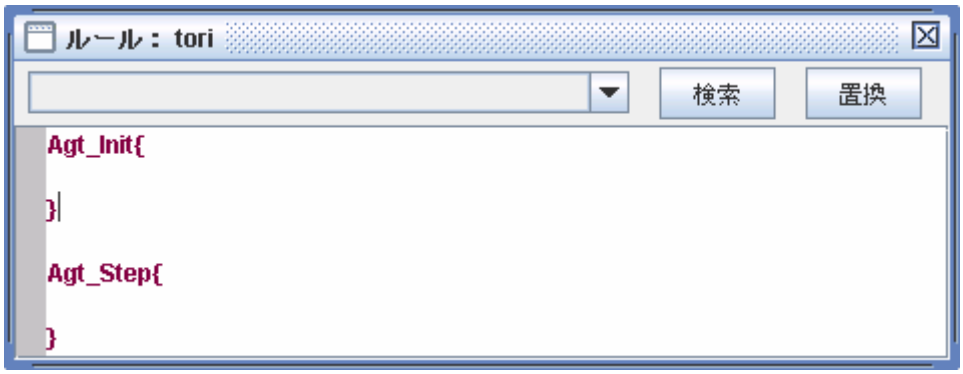
（これを「ステップ」と呼ぶ）として時刻を刻みます。ある時点と次の時点との間（ステップ間隔）が現実の社会でどれだけの時間に対応するかは、別に考えます。

さて、artisoc では、エージェントの行動ルールは、エージェントの「ルールエディタ」に書き込みます。

まず、artisoc を起動してみましょう。すると、何もしていない新しいモデルの画面が現れます。それを無視して、ファイル・メニューの「開く」をクリックして、tobutori.model のファイルを開いて下さい。前章で作成した、tobutori.model が開かれたと思います。

「ツリー」にある tori を選択（クリック）して、tori にしてください。そしてメニューの「表示」から「ルールエディタ」を選択して下さい。ルールエディタを開くことは、ツリーの tori をダブルクリック、または、右クリックして「ルールエディタ」を選択してもできます。

「ルール：tori」というルールエディタの画面が開いたと思います。



ここで、Agt_Init{　} と Agt_Step{　}という２つの括りが、既にルールエディタの中に書き込まれていることに注目して下さい。Agt_Init{ から　}の部分はシミュレーションを実行したとき、最初だけに実行させるルールを書き込みます。そして、Agt_Step{ から　}の部分には、毎ステップ実行させるルールを書き込みます。

最初に１回だけ実行させるのは、上の（１）と（２）です。これを、artisoc のルールエディタにルールとして指定しましょう。それは、次のように書きます。

まず、Agt_Init{　の次の行に、my.と半角英数字で書き込んで下さい。すると、どうでしょう。「.」を入力したとたん、エージェント変数が縦に並んだメニューが現れたはずです。これは、モデルを作る上での artisoc が用意した「支援機能」で、その中から適当な変数を選択すれば良いのです。この支援があるおかげで、入力する手間が省けるだけでなく、ミスタイプのエラーが

起こらなくなります。この段階で現れる ID, X, Y, Layer，Direction の５種類の変数は artisoc が自動的に設定する変数ですが、しばらく X，Y，Direction の３つだけ使います。（ID と Layer については無視してください。）

最終的には、次のようなルールを書き込みます。

```
Agt_Init{
my.X = 25
my.Y = 25
my.Direction = rnd()*360
}
```

何となく判ると思いますが、４点注意を要します。

（１） my.X とか my.Y など、各エージェント変数(X, Y,D irection など)の頭に「my. 」を付けた変数は、沢山あるエージェントの１体毎に指定される自分(my)だけの変数を表します。

（２） Direction とは、移動方向を指定する特別の変数です。空間（oozora）上で、エージェントのいる地点から９時方向を０度とし、反時計回りに１回転を３６０度とする値をとります。

（３） rnd()とは、０から１の間の一様擬似乱数を発生させる関数です。「rnd()」そのものが０以上１未満のランダムな値になります。「*」はかけ算を表します。したがって、rnd()*360 というのは、０度以上３６０度未満の値になります。

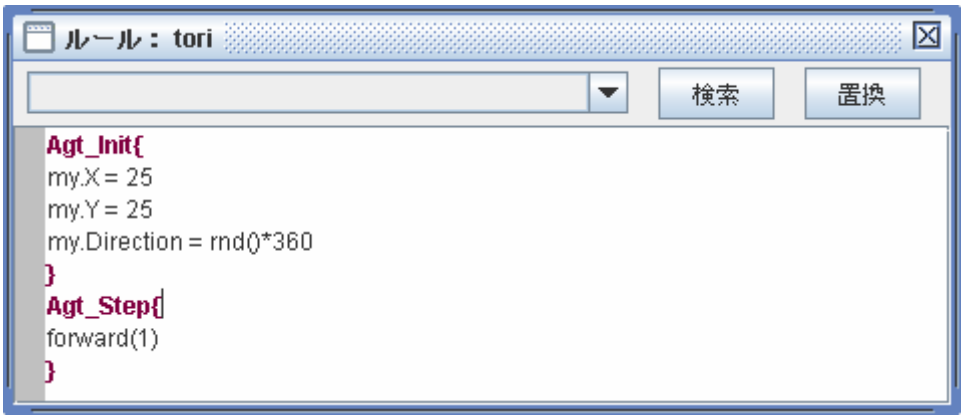
（４） 全ての式について、等号（=）は、その右の値を左の変数の値にしまう（代入する）という操作を意味します。等号の右側と左側が等しい、と比べているわけではありません。数学の記号とは意味が違いますので気をつけて下さい。

次に、毎ステップ実行するルールを指定します。それは、最初に飛び立つ方向に、「１」ずつ飛んでいく、というものです。これは、Agt_Step{　　}部分に

```
Agt_Step{
forward(1)
}
```

と書きます。ここで注意を要するのは、「１」というステップ毎の飛翔距離は、oozora という空間を 50x50 に設定したことに対応する「１」です。つまり、左下端から真横または真上に飛び続けると、５０ステップ後に右端または上端に着くという意味です。

結局、「ルール：tori」というルールエディタには、下図のようにルールが書き込まれます。



これでルール指定は完了です。この辺で、「上書き保存」しておきましょう。

もうシミュレーションを実行できます。では、実行ボタンを押して下さい。１００羽の tori が中央部から四方八方に飛び去るのが、出力画面 OOZORA に見えるはずです。

コラム　『実行過程が速すぎてエージェントの動きが見にくいとき』

高性能なコンピュータで簡単なモデルを実行すると、描写のスピードが計算のスピードに追いつかず、表示が途切れ途切れになってしまうことがあります。こんな時には、一旦、シミュレーションを終了させてください（キーボードの ESC キーを押して実行を終了させるという割り込みも可能です)。artisoc では、実行速度を遅くして、マップでのエージェントの動きをなめらかにする方法には２種類あります。ひとつの方法は、メニューの「設定」から「実行環境設定」を開き、「実行ウェイト」を大きくすることです。たとえば、この所に１００と入れてみてください。動きがゆっくりと見えるはずです。 もうひとつの方法は、「ガーベージコレクション」(artisoc を動かしているソフト JAVA の機能です）を１にすることです。(ガーベージコレクションのデフォルトは０で、自動的に最適化（なるべく速く実行するようにする）されるので、速いですが、なめらかさに欠けます。)

しばらく経つと、tori は上下左右の縁で反転しているように見えますが、そうではありません。oozora という空間は、ループしているのです（「空間プロパティ」のデフォルト値）。つまり、上端と下端、左端と右端とはつながっているのです。したがって、エージェントが左から右に移動して右端に着くと、すぐに左端から再登場するのです。（artisoc でのループした空間（X-Y２次元平面）は、ちょうどドーナツの表面を縦と横に開いたものを想像して下さい。）

２．３　動かすルールに慣れる

このようにして完成した tobutori モデルをいろいろと修正してみましょう。

◎tori の数を１００羽から１０００羽に増やす。

（１）「ツリー」の中の tori を選択する

（２）メニューの「表示」から「プロパティ」を選択する

（３）「エージェントプロパティ」画面が開くので、その中のエージェント数を 1000 にする。

これで完了。実行ボタンを押して下さい。

◎tori を左上端から飛び立たせる

（１）「ツリー」の中の tori を選択する

（２）メニューの「表示」から「ルールエディタ」を選択する

（３）my.X，my.Y の代入式の値を(25，25)から(1，49)に変える。

これで完了。実行ボタンを押して下さい。

◎tori は飛ぶ方向を毎ステップでたらめに選ぶ

（１）「ツリー」の中の tori を選択する

（２）メニューの「表示」から「ルールエディタ」を選択する

（３）rnd()*360 の式を上の Agt_Init{ }内から、下の Agt_Step{ }内に移す。

これで完了。実行ボタンを押して下さい。

このように簡単に修正できます。そして、各々、tori の飛び方にもいろいろな違いが現れたはずです。

２．４　復習用の課題

 後日、試して下さい。

そこで、少し練習問題です。各自、試してみましょう。

＜練習問題２．１＞

中心から上半分の様々な方向に tori を飛ばしてみましょう。

（ヒント：My.Direction = rnd()*180）

＜練習問題２．２＞

毎回ランダムなスピードで tori を飛ばしてみましょう。

（ヒント：Forward(rnd())）

以上、第２章 

第３章　エージェントに判断させる

◎ 状況に応じて、異なる行動をさせる
◎ 「場合分け」の基本を学ぶ
◎artisoc のルール表記「イフ文」を学ぶ

３．１　本格的な「自律的な行動」とはどのようなものか

 早く artisoc を動かしたい人はとばしても構いません

前章では、エージェントに弱い捉え方の「自律的な行動」をさせる第１歩を学びました。この章では、強い捉え方の「自律的な行動」をさせる基本を学びます。強い捉え方は、ルール自体がエージェントに自分の行動を選択できるようになっている場合のみに限定して自律的行動を定義します。

例を用いて、強い捉え方の自律的行動を説明しましょう。今、Ｐ地点にいるエージェントがＱ地点に移らなければいけないとしましょう。Ｐ地点からＱ地点に行くには、Ｊ、Ｋ、Ｌの３通りの方法があります。移動費用ではＪ、Ｋ、Ｌの順で高額なのですが、移動時間はＪ、Ｋ、Ｌの順で短時間です。他方、気持ちの良い順では、Ｋ、Ｊ、Ｌです。エージェントが急いでいれば、高額でも最速のＪを選ぶでしょう。もし時間に余裕があればＫを選び、懐が寂しければＬを選ぶでしょう。この例では、エージェントにとっての望ましさを決める複数の評価基準から特定のものが選ばれ、その基準に従って複数の選択肢から特定のものが選ばれます。このように、強い捉え方での自律的行動とは、複数の可能性から一つを選択する、ということが前提になります。

あらかじめ、さまざまな可能性を設定して、ある場合にはどの可能性が選択されるのかを指定することを「場合分け」（または「条件分岐」）といいます。この章では、「場合分け」を artisoc のルールとして、エージェントのルールエディタにどのように書き込むかを学びます。

３．２　何をしたいのか（エージェントに何をさせたいのか）を図で表す

プログラミングに慣れている人はとばしても構いません。早く artisoc を動かしたい人でも、プログラミングに慣れていない人は読んで下さい。

モデルを作る作業にはさまざまな段階がありますが、エージェントの行動ルールを明記することは基本中の基本です。この、行動ルールの明記には、２つの段階があります。ひとつは、まず、私たちがエージェントに何をさせたいかをはっきりさせることです。もうひとつは、それをシミュレーション用のモデルの一部としてはっきりさせることです。つまり、artisoc のルールエディタに正しく書き込むことです。

前章では、すでにこの２段階を実践しています。

第１段階：ルールの決定

> （１）全ての tori が oozora の中央部にいる。
> （２）各 tori は、そこから自分の好きな方角を決める。
> （３）各 tori は、毎ステップ、「１」ずつ、その方角に飛び続ける。

第 2 段階：シミュレーション・モデルでの表現（artisoc のルールエディタへのルールの書き込み）

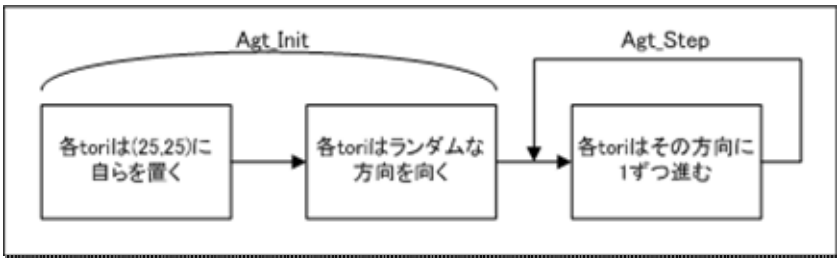
```
Agt_Init{
my.X = 25
my.Y = 25
my.Direction = rnd()*360
}

Agt_Step{
forward(1)
}
```

この例の場合、第２段階の作業はもちろん必要ですが、第１段階については、自分の頭のなか

で、エージェントに「こういうことをさせる」内容がはっきりしていれば、必ずしも、メモを書いたり、絵を描いてみる必要はありません。

これから学ぶ「場合分け」は、ルールが複雑になります。第１段階の作業を、頭のなかだけでなく、紙に図示したりする習慣をつけましょう。図示する方法のひとつに、「フローチャート」があります。これは、もともとコンピュータに何かをさせようとするときに、その作業の「流れ（フロー）」を「図（チャート）」に描いたことに由来します。フローチャートを描くのに際しては、「場合分け」を見やすく描く方法が標準化されています。それは、コンピュータにさせたいことをコンピュータに分かる方法で指定する（プログラミング）ときに、「場合分け」がとても重要で、しかも頻繁にあるからです。artisoc でルールエディタにルールに書き込むことは、プログラミングとは言えませんが、考え方の基本は同じです。

単純ですが、上の例では、フローチャートは次のようになります。



複雑な例は、これから登場します。

３．３　「場合分け」を図示して、正しく理解する

「場合分け」の基本は、当たり前ですが、どのような場合があるのか、そして場合毎にエージェントは何をするのかをはっきりさせることです。ここでは、あまり「自律的行動」らしくありませんが、「場合分け」のルール化の基礎を学びます。

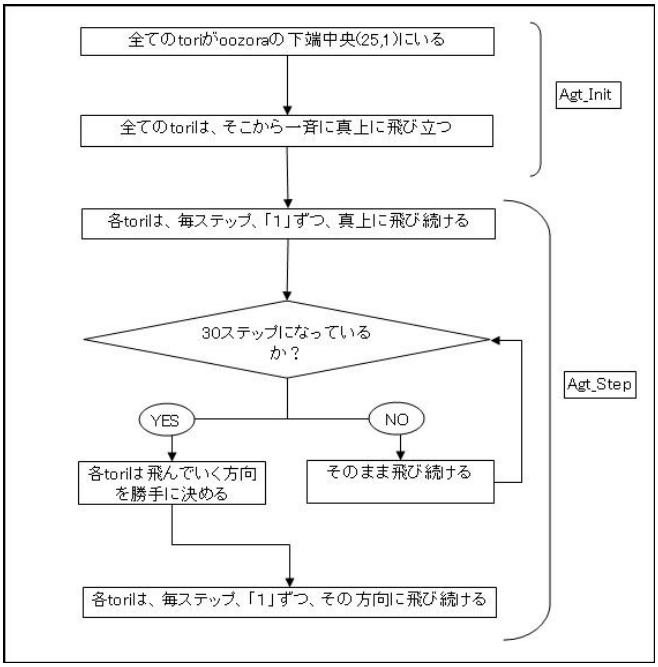
次のようにエージェントを行動させてみましょう。

（１）　全ての tori が oozora の下端中央(25,1)にいる。
（２）　全ての tori は、そこから一斉に真上に飛び立つ
（３）　各 tori は、毎ステップ、「１」ずつ、真上に飛び続ける。
（４）　30 ステップ目に、各 tori は自分の好きな方角をめざす
（５）　その後、各 tori は、毎ステップ、「１」ずつ、その方向に飛び続ける。

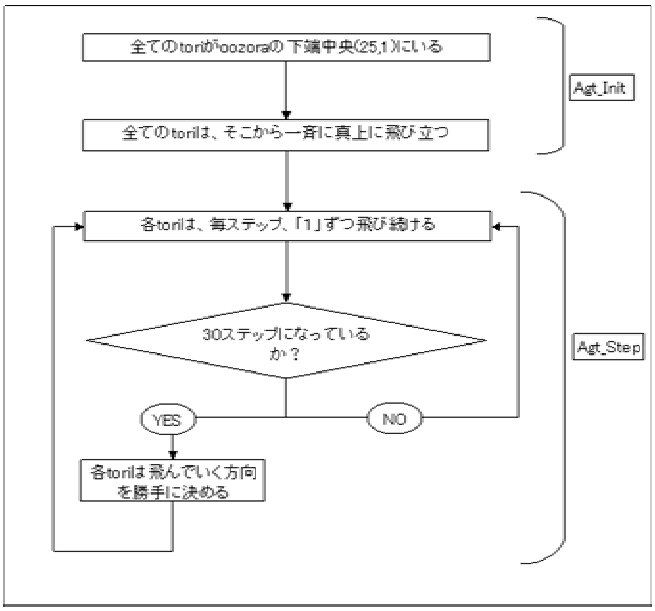
このような行動ルールのどこが「場合分け」なのでしょうか。一見すると、どこにも「場合」がなさそうです。しかし、それがあるのです。それでは、上の行動ルールを次のように書き換えてみましょう。

（１）　全ての tori が oozora の下端中央(25,1)にいる。
（２）　全ての tori は、そこから一斉に真上に飛び立つ
（３）　各 tori は、毎ステップ、「１」ずつ、真上に飛び続ける。
（４）　30 ステップになっていなければ、そのまま飛び続ける。
（５）　30 ステップ目に、各 tori は飛んでいく方向を勝手に決める。
（６）　30 ステップを超えたら、各 tori は、毎ステップ、「１」ずつ、その方向に飛び続ける。

ここで、（４）（５）（６）が「場合分け」です。30 ステップ「より少ない場合」、「ちょうどの場合」、「より多い場合」で、各々の場合に、行動ルールが異なっています。ではこの行動ルールをフローチャート化しましょう。



このフローチャートを眺めてみると、30 ステップ「より少ない場合」と「より多い場合」とが、同じ行動ルールになっていることに気づくはずです。ですから、実は、スマートなフローチャートは次のようになります。（もちろん、上でもまちがいではありません）



３．４　「場合分け」ルールでエージェントに判断させる

まず、モデル作りの準備作業をしましょう。これは、第３章で tobutori モデル作りの準備を

したのと、全く同じ作業をして下さい。この新しいモデルは、hanabi.model と名前を付けて、保存して下さい。

本来なら、最初から作り始めるのが慣れるためには良いのですが、チュートリアルでは時間がもったいないので、tobutori.model を開いてください。それを、hanabi.model と名前を付けて、保存して下さい。それから、自分で入力したエージェントのルール（Agt_Init{, }, Agt_Step{, } を除くエージェント・ルールエディタの中身）を削除して下さい。

では、スマート版フローチャートにしたがって、いよいよルールを書き込みましょう。tori のルールエディタを開いて下さい。Agt_Step{ と }との間には、「場合分け」した行動ルールを次のように書き込みます。新しい表現がいくつか登場しますが、すぐ後で、説明します。

```
Agt_Init{
my.X = 25
my.Y = 1
my.Direction = 90
}

Agt_Step{
If 30 == getcountstep() then
        my.Direction = rnd()*360
Else
        forward(1)
End if
}
```

以上を書き込んだら、上書き保存をしてから、取り敢えず、実行してみましょう。なぜ、このモデルを hanabi と名付けたのか分かりますね。

では、上のルールを説明します。まず、大きな構造である

If XXXXX then YYYYY Else ZZZZZ End if

に注目して下さい。これは「もし(If) XXXXX ならば(then) YYYYY をしなさい。そうでなければ(Else) ZZZZZ をしなさい。これで場合分けは完了(End if)」という「場合分け」の基本構造です。(英語の文法に似ています。コンピュータに関する技術が、アメリカで発展してきた名残です。)

ここで、XXXXX の部分にある 30 == getcountstep()が新しく登場した表現です。まず、getcountstep()は artisoc でシミュレーションが実行され初めてからのステップ数です。30 == はステップ数がちょうど３０であることを意味しています。数学の等号（=）を二つ並べた記号は、記号の左辺と右辺の値が同じであることを表しています。つまり、右辺の値と左辺の値を比較して等しいことを表現しているのです。この記号は、通常、If XXXXX then の XXXXX のところでしか使われません。(getcountstep() == 30 と書くことも可能です。しかし、代入(=)と等しい関係（==）とを特に初心者は混同しがちなので、代入文との違いをはっきりさせるために、30 == getcountstep()と書くことを勧めます。こう書けば、「getcountstep()で得られたステップ数を 30 に代入する」というわけの分からない文と混同する心配はないでしょう。)

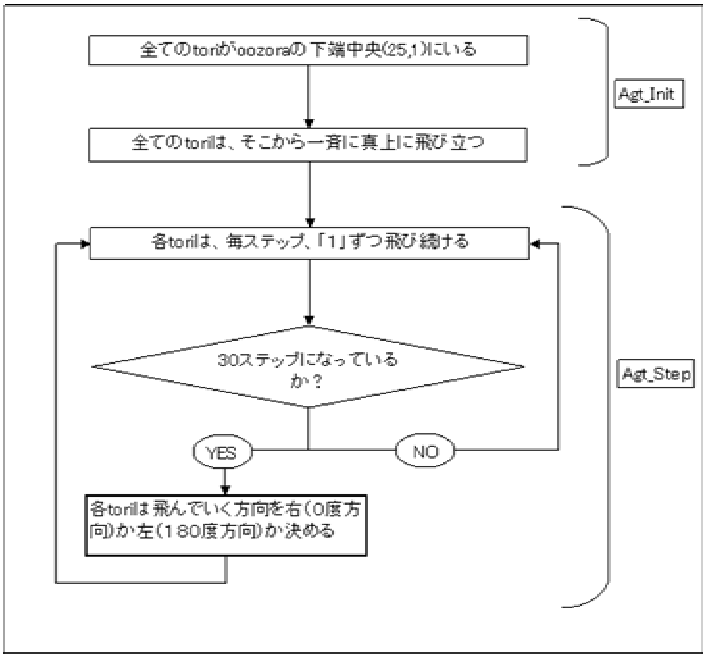
したがって、上のルールは「もしステップ数がちょうど３０なら、各 tori の飛ぶ方角をランダムに選びなさい。そうでなければ、決まっている方角に毎ステップ１だけ飛び続けなさい。これで場合分けは完了。」となります。このルールがスマート版フローチャートに対応していることを確認して下さい。

文法事項の説明は以上です。

３．５　「場合分け」のルールに慣れる

さて、完成した hanabi モデルをいろいろと修正してみましょう。その過程で、新しい表現方法も学びます。

◎３０ステップで、各 tori は右（０度方向）か左（１８０度方向）に飛ぶ方向を選択する。
にモデルを変更してみましょう。(見た目はあんまり面白くないですが…)



３０ステップ目で「飛んでいく方向を勝手に決める」だった部分を「各 tori は飛んでいく方向を右か左か決める」に変更することがポイントになります。

```
Agt_Step{
If 30 == getcountstep() then
      If rnd() > 0.5          then
            My.Direction = 0
      else
            My.Direction = 180
      End if
Else
         forward(1)
End if
}
```

rnd() > 0.5 という表現に注意が必要ですね。rnd（）は前にも登場しましたが、0 以上１未満の一様乱数を発生させる命令です。なので、rnd（）>0.5 は、50%の確率で「真」、50%の確率で「偽」となります。半分の確率で自分の方向（My.Direction）を右（0）にして、そうでなけれ

ば左（180）にしてねということになります。

３．６　復習用の課題

 後日、試して下さい。

そこで、少し練習問題です。各自、試してみましょう。次のページのコラムの内容も参考にしてみてください。

＜練習問題３．１＞

前節で作った hanabi モデルの改良版をさらにちょっと変更してみましょう。３０ステップ目で左右に分かれた tori たちが、４０ステップ目で各 tori がランダムに好きな方向を選ぶというルールにしてみましょう。

（ヒント、次ページのコラムで説明しているエルスイフ（Elseif）文を使ってみてください）

コラム　　『複雑な「場合分け」』：エルスイフ（Elseif）文

「場合分け」のことを「条件分岐」と呼ぶこともあります。本文でお教えしたより複雑な条件分岐を表現するためには以下のようなルールの書き方があります。簡単に表現すると、「○○だったら××して、そうじゃなくて△△だったら＊＊して、」といったルールを表現する方法です。

```
If      条件文１        then
        AAA
Elseif  条件文２        then
        BBB
Elseif  条件文３        then
        CCC
Else
        DDD
End if
```

条件文１が成立していれば、AAA を実行して、end if に飛びます。条件文１が成立していなくて、かつ、条件文２が成立していれば、BBB を実行して end if に飛びます。条件文１と２が成立していなくて、かつ、条件文３が成立していれば、CCC を実行して end if に飛びます。もし、条件文１、２、３が全部成立しない場合には、DDD が実行されます。

このように、If で始まり、End if で終わる構文を「イフ文」と呼びます。

If, Elseif, Else の使い分けに注意しましょう。なお、Elseif にはスペースがありません。End if にはスペースがあります。スペースの有無で、エラーになったりします。

コラム　『単純な「場合分け」』

複雑なものを勉強したら、単純なものも勉強しておきましょう。条件分岐の最も単純な形は以下のようなものです。

```
If　条件文１　　then
　　AAA
End if
```

条件文１が成立していれば、AAA を実行しなさいというルールです。

コラム　『「等しい」以外の関係』

教科書の本文で、ちょっとフライング気味で出てしまいましたが、等しい以外の関係の書き方をまとめておきましょう。条件文をいろいろと書くのに便利だと思います。

＝＝　　等しい
＜＞　　等しくない
＜　　より小さい
＜＝　　より小さいか等しい（以下）
＞　　より大きい
＞＝　　より大きいか等しい（以上）

以上、第３章

第４章　エージェントに周囲の環境を調べさせる

◎ エージェントに周囲を観察させる
◎ エージェントに変数を追加する
◎ 周囲にエージェントがいるかどうかを調べさせる

４．１　環境に応じた「自律的行動」


早く artisoc を動かしたい人はとばしても構いません）

前章では、エージェントに強い捉え方の「自律的な行動」をさせる第１歩として、「場合分け」の方法を学びました。前章の例では、ステップ数とか、エージェント自身が動いている方向とかにしたがって、行動の場合分けを実行しました。しかしこれでは、エージェント自身が自分の置かれた周囲の状況を認識して行動を変えた、というわけにはいきません。「エージェントが自律的に行動している」と見なせるためには、「エージェントの周囲の環境をエージェント自身が認識して、その認識如何にしたがって行動を選択する」というように、場合分けの条件を各エージェントに固有の環境条件にすることが重要です。

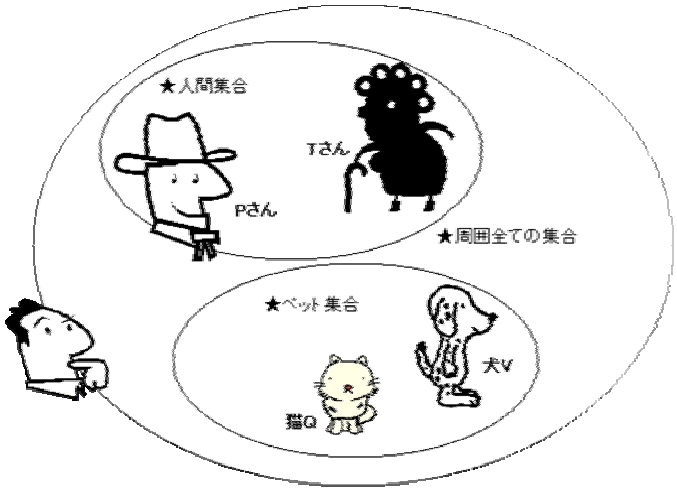
マルチエージェント・シミュレーションの特徴の一つに、エージェントは全体についての知識は持っていなく、自分の周囲の環境だけを知ることができる、という設定方法があります。この方法を用いれば、エージェント毎に周囲の環境は異なりますから、周囲の環境に応じて異なった行動を各エージェントにとらせることが可能になります。

４．２　自分の周りのエージェントたち

artisoc に備わったエージェント周囲の認識方法を利用して、周囲の環境によってエージェントの行動を変えさせる基礎を学びましょう。前章で学んだ「場合分け」の条件として、周囲の環境についての情報を用いるのです。いよいよ「自律的行動」（強い捉え方）のルール化の入り口

に来ました。

特にここでは、エージェントの周囲にいる他のエージェントを認識する方法の基本を学びます。図のように、特定のエージェント（自分）の周囲にどのようなエージェントがいるかを認識するということは、（１）「周囲」の具体的な範囲（「視野の広さ」）、（２）周囲を見回して、そこにいる他のエージェントを「認識」したことを表す変数、をはっきりさせることが必要です。（１）については直感的に分かると思いますが、（２）については少し説明が必要かも知れません。周囲に、友達のＰさんとＴさん、それにＰさんが飼っている猫のＱ、Ｔさんが飼っている犬のＶがいるとしましょう。周囲を認識するということは、自分の周囲に「Ｐさん、Ｑ、Ｔさん、Ｖ」がいると判ることです。周りにいる人間を認識するなら「Ｐ、Ｔ」、ペットなら「Ｑ、Ｖ」です。したがって、周囲にいるエージェントを認識したことを表す変数は、数値ではなく、「Ｐ、Ｔ」、「Ｑ、Ｖ」、「Ｐ、Ｑ、Ｔ、Ｖ」といったエージェントの集合になります。このように、エージェントたち（エージェントの集合）を値とする変数を「エージェント集合型」変数と呼びます。（コラムも参照にしてください。）



artisoc では、エージェントに自分の周囲にいるエージェントたちを認識させる方法のひとつとして、

```
MakeAllAgtSetAroundOwn()
```

という関数があり、

```
MakeAllAgtSetAroundOwn(my.Neighbor, 2, False)
```

というふうに記述します。ここで、my.Neighbor とは認識した周囲のエージェントを記録しておく変数で「エージェント集合型」です。次の２は、視野が自分を中心として２の範囲であることを意味します。最後の False は、自分自身を含めない認識方法を表しています。つまり、周囲に自分以外に誰もいないとき、「自分しかいない」と認識する方法が True で、「誰もいない」と認識する方法が False です。この例では、my.Neighbor の値は「Ｐ、Ｑ、Ｔ、Ｖ」になるでしょう。

コラム　『「エージェント集合型変数」って何のこと？』

エージェントが１体、２体、、、、Ｎ体いるとき、何体いるかを表すために、Number という変数を用いることにしましょう。この変数は 0, 1, 2, , , , N, , ,といった整数の値をとります。このことを、Number は「整数型変数」である、といいます。このことは、また、Number が 1.4 とか 7.3 とかの値をとれない（エージェントが 1.4 体いると勘定することはない）ことを意味しています。

さて、自分の近隣にいるエージェントが、A, B, C, , , , N, , ,などのとき、だれとだれとがいるのかを表すために Neighbor という変数を用いることにしましょう。では、Neighbor はどんな値をとるのでしょうか。近隣にいるエージェントの数ではないことに注意して下さい。Neighbor という変数の値としては、A, B, C, , , , N, , ,と近隣にいる全エージェントをリストアップするしか表しようがありません。つまり、A, B, C, , , , N, , ,というエージェントの集合が Neighbor の値なのです。エージェント集合を値としてとるので、Neighbor を「エージェント集合型変数」と呼びます。

コラム　『そもそも変数の型って何のこと？』

第１章でも少しふれましたが、変数には「型」があります。「型」はそれぞれの変数がどんな種類の値をとるのかを前もって決めたものと考えてください。例えば、整数型の変数は、０、１、２といった値をとります。実数型の変数は、3.1415…とか 2.236…とかいった値をとります。変数のとる値は数字だけとは限りません。文字列型の変数は、YES とか YAMAKAGE といった文字列を値としてとります。

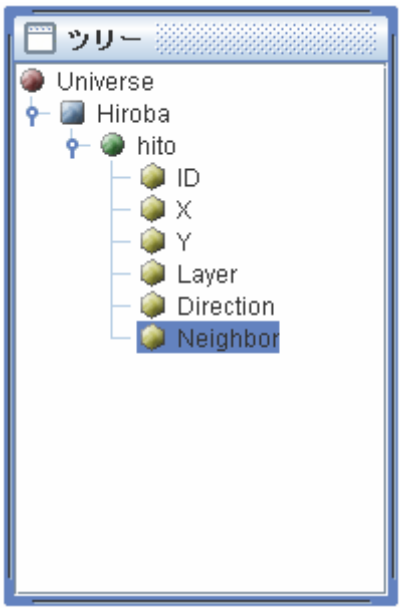
そして、artisoc には、エージェント型とかエージェント集合型といった変数の型があります。エージェント型とはエージェントをその値としてとります。エージェント集合型はエージェントの集合をその値としてとるわけです。

例えば、周囲に A、B、C、D、E、F といった人々がいるとします。「恋人」という変数は、エージェント型変数ですね。A とか D とかいう値をとるわけですね。「友達」という変数はエージェント集合型変数です。その値は、{A、B} だったり、{B、D、F} だったりするわけです。友達がいないときは {}（空集合）ですね。

４．３　モデル作りの準備で、新しい変数を追加する

まず、今までの復習を兼ねて、全く新しいモデル作りの準備作業をしてください。空間の名前を Hiroba（プロパティはデフォルト）、エージェント（種）は hito として、エージェント数を１００人に設定して下さい。

エージェントには、Neighbor という変数を追加して下さい。artisoc が自動的に設定する ID などの変数だけでは、周囲を認識させることはできません。Neighbor は自分の周りにどんなエージェントがいるかを調べた結果を記録しておくための変数です。まず、hito を選択して hito とハイライトしてから、「挿入」から「変数の追加」をクリックして、変数プロパティ画面を出します。「変数名」のところに Neighbor と書き込み、「変数の型」をエージェント集合型にしてください。



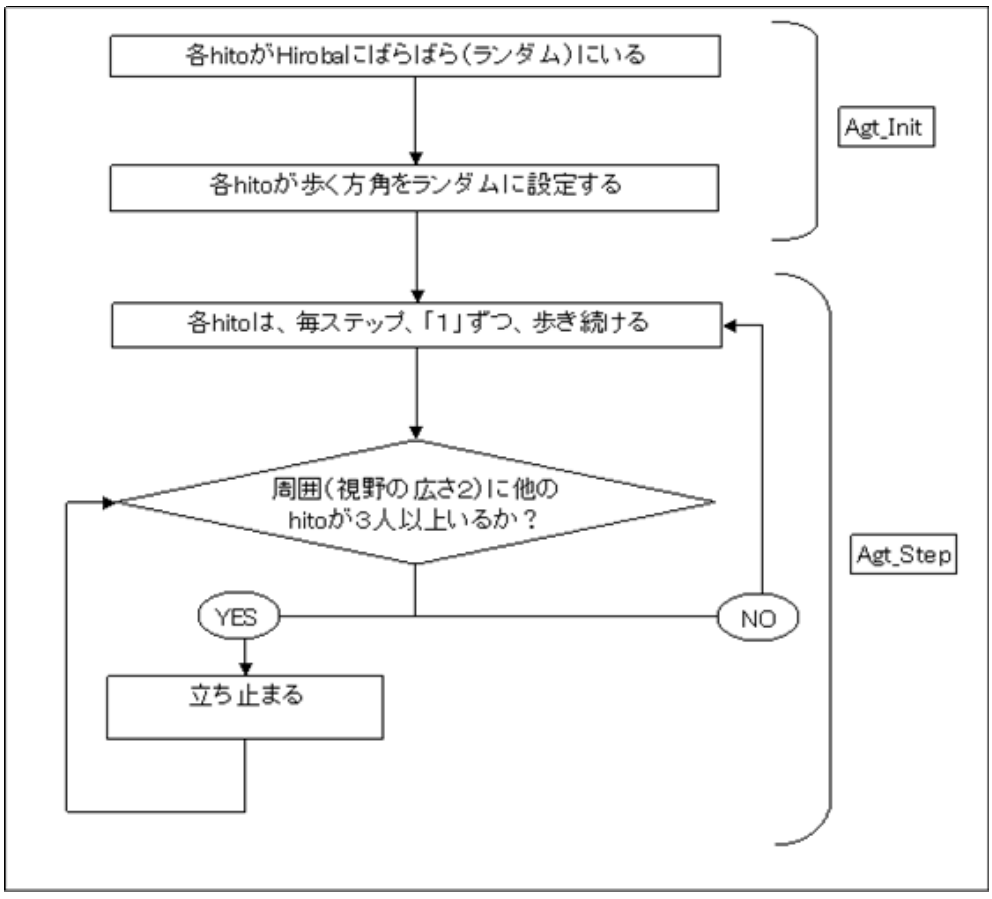
出力設定（マップ出力）も、忘れずにしてください。このモデルは、tachibanashi と名付けて保存して下さい。

４．４　周囲にエージェントはいくついるか調べさせる

さて、次のようにエージェントを行動させてみましょう。

（１）各 hito が Hiroba にばらばら（ランダム）にいる。
（２）各 hito が歩く方角をランダムに設定する。
（３）各 hito は、毎ステップ、「１」ずつ、歩き続ける。
（４）ただし、周囲（視野の広さ２）に他の hito が３人以上いるときには、そこで立ち止まる。

（１）と（２）は、Agt_Init{ }の中に書き込むルールです。（３）（４）が Agt_Step{ }の中に書き込むルールで、周囲の環境に応じた行動の場合分けです。



ルールエディタには次のようにルールを書き込みます。新しい表現がいくつか登場しますが、すぐ後で、説明します。

```
Agt_Init{
my.X = rnd()*50
my.Y = rnd()*50
my.Direction = rnd()*360
}

Agt_Step{
MakeAllAgtsetAroundOwn(my.Neighbor, 2, False)
If 3 <= CountAgtset(my.Neighbor) then

Else
        foward(1)
End if
}
```

では、上のルールを説明します。上半分（Agt_Init{　}）はもう説明不要でしょう。hito の初期の居場所や歩こうとする向きを、rnd()という乱数発生関数を利用して、ランダムに設定しています。

後半部分は、まず MakeAllAgtsetAroundOwn(my.Neighbor, 2, False)で周囲にいる（視野２の範囲で）エージェントを全て認識します。それに続く大きな構造である If XXXXX then YYYYY Else ZZZZZ End if は、前章で学んだ「場合分け」の基本構造です。

ここで、XXXXX の部分にある 3 <= CountAgtset(my.Neighbor)が新しく登場した表現です。これは、my.Neighbor というエージェント集合型変数に記録されているエージェントの数を求めるための関数です。この変数は整数型です。つまり、周囲にいるエージェントが３以上なら YYYYY というルールで行動し、そうでないなら（３未満なら）ZZZZZ というルールで行動する、ということです。ここで YYYYY のルールが何も書かれていないことに注意して下さい。これは何もしない（じっとしている）ことに対応しています。このように、本来なら何かルールが書き込まれるはずの文章（ここでは YYYYY の部分）が空（くう）のとき、これを「空文」と呼びます。

ルールを書き込んだら、上書き保存をし、実行してみて下さい。

４．５　パラメータを変えてみる

ここで、マルチエージェント・シミュレーションの醍醐味を少し味わってみましょう。

マルチエージェント・シミュレーションの目的の一つに、ミクロな状態（典型的にはエージェントの自律的行動の仕方）の変化がマクロな状態（典型的には空間での集団行動のあり方）の変化にどのような影響を与えるか、を調べることにあります。ここで作ったモデルに従えば、視野の広さや立ち話をするときの最少人数をオリジナルなモデルの設定（視野２、最少人数３）を変えることによって、立ち話集団の数や規模がどのように変わるかを調べることです。

このように、モデルの基本を変えないで結果がどのように変わるかを調べるために変化させる変数のことをパラメータと呼ぶことがあります。ここでは、視野の広さや立ち止まる最少人数がパラメータになっています。

tachibanashi モデルのルールエディタを開いて、

（１）視野の広さを１に減らす、２のままにする、３に増やす
（２）最少人数を２人に減らす、３人のままにする、４人に増やす

という変更作業を全ての組み合わせについて行い、その度に何回か「実行、終了」をしてみて、Hiroba の全体的な様子の違いを観察しましょう。立ち話する集団の数や大きさに大きな違いが現れてくることが容易に見て取れたことと思います。

４．６　頭の体操（同じルールを異なる表現で表してみよう）

◎自分を含めるか含めないか

```
MakeAllAgtsetAroundOwn(my.Neighbor, 2, False)
If 3 <= CountAgtset(my.Neighbor) then
```

この記述について、周囲の認識部分と場合分けの部分をほんの少し変えた（false を True に、3 を 4 に）次の２行

```
MakeAllAgtsetAroundOwn(my.Neighbor, 2, True)
If 4 <= CountAgtset(my.Neighbor) then
```

は、機能的には同一であることは分かりますね。

◎何もしない場合を明示するかしないか

```
If 3 <= CountAgtset(my.Neighbor) then

Else
        Forward(1)
End if
```

この記述では、周囲のエージェントが３以上の場合と３未満の場合とでルールがどのように異なるのかがはっきりと分かります。わざわざ何もしない場合を「空文」として明記してあるからです。そこで、何かする場合だけを明記する表現方法を考えて下さい。解答例は次のようなものです。

```
If 3 > CountAgtset(my.Neighbor) then
         Forward(1)
End if
```

同じルールで記述が短くなっていますが、必ずしも短ければ良いというわけではありません。分かりやすい表現が最も大事な点です。

４．７　モデルを複雑にしよう

自律的行動を「周囲の認識」と「場合分け」とから表す基本のまとめです。tachibanashi モデルに次のような変更を加えましょう。今まで学んだ技法を総動員します。

（１） 新しく pet というエージェントを１００匹追加します。
「ツリー」の中の空間「Hiroba」の下に hito を作ったのと同じように作業します。（ヒント：「挿入」メニューから「エージェントの追加」を選択します。）
（２） 出力マップに pet エージェントの出力を追加します。
新しい出力マップではなく、hito を出力させたマップに追加する必要があります。（ヒント：「設定」メニューから「出力設定」＞ 「マップ出力」を開き、「マップ要素リスト」に pet を追加します。）
（３） pet エージェントは初めは Hiroba にランダムにいます。
（４） pet の走る方向は毎ステップ「ランダム」です。
（５） pet は毎ステップ「１」だけ走ります。

このように修正したモデルを tachibanashi-X という新しい名称で保存して下さい。

実行しましょう。注意深く観察しないと判りにくいのですが、一旦立ち止まっても、また歩き出す hito がときどきいます。なぜでしょうか？

実は、このモデルでは、hito が認識している周囲のエージェントは hito だけでなく pet もいます。そのため、hito と pet を合わせた数が一定以上だと、そこで止まってしまいます。ところが、pet は勝手に走り回っていますから、周囲のエージェントが減ってしまうとまた歩き出す、といった現象が生じているのです。

では、pet を無視して、他の hito がそばにいるかどうかだけを気にするように hito エージェントの行動ルールを変えましょう。ここで活躍するのが、MakeOneAgtsetAroundOwn()です。上で

学んだ MakeAllAgtsetAroundOwn に似ていますが、この新しい認識方法は１種類のみのエージェント種を認識します。ここでは hito を認識させることにしましょう。

```
MakeOneAgtsetAroundOwn(my.Neighbor, 2, Universe.Hiroba.hito, False)
```

ここで、hito エージェント種を Universe.Hiroba.hito と表している点に注意して下さい。「宇宙(Universe)の中に作った Hiroba という空間で行動する hito エージェント種」と厳密に表記して誤解が生じないようにします。これは artisoc でのルール表記の鉄則です。

tachibanashi-X モデルの周囲のエージェントを認識する部分を新しいものに変えて下さい。それを保存して、実行して下さい（なお、視野と最少人数はもとの２と３に戻して下さい）。pet は無視されていますから、hito の行動は Hiroba モデルを同じになっているはずです。確認しましょう。ちなみに、私個人としては、ペットを無視して人間とだけ立ち話をする人よりは、人間だけでなくペットも（ペットだけでも）自分のそばにいると立ち止まるような人の方が好きですが。

４．８　復習用の課題：もっと自然に

後日、復習を兼ねて、トライしてみて下さい。

＜練習問題４．１＞

tachibanashi モデルでも tachibanashi-X モデルでも、hito や pet は最初に歩く（走る）方向が決まるとずっとそのまま真っ直ぐ進みます。「等速直線運動」ですから、これでは不自然ですね。そこで、hito は毎ステップ左右１０度の範囲でランダムに、pet は毎ステップ左右３０度の範囲でランダムに進行方向を変えるモデルに修正して下さい。また、hito も pet も同じ速さで動いています。これも不自然かも知れません。そこで、pet は毎ステップ「２」移動するように修正して下さい。
（ヒント、次ページのコラムのターン（Turn）という行動ルールを使ってみてください）

練習問題を解いてみてから読んでください。

petやhitoの動き方に変化がでたことはすぐに気が付くと思います。hitoはぶらぶら歩きに、pet は活発に走り回る感じになったでしょう。たしかに、このように修正すると、不自然さが減りました。それでは、Hiroba の全体的な特徴には大きな変化が生まれましたか。そんなに大きな違いはなかったはずです。

このモデルで本質的な部分は、エージェントの細かな動き方ではなく、周囲の環境をどのように認識して行動に結びつけるか（視野の広さ、hito だけか hito も pet もか、どれだけそばにいれば立ち止まるか）にあることが判ったと思います。「シミュレーションはなるべく現実に近づけた方が良い」という考え方があります。しかし、どんな場合でも現実らしいモデルを作らないといけない、というわけではないことは、この例で分かってもらえたでしょうか。

コラム　ターン（Turn）：方向を変える

Forward は「前に進む」というルールでした。それと組み合わせて用いると便利なのが、Turn というルールです。

Turn（15）

というふうに表記して、() 内の角度だけ自分の方向を変えるルールです。() 内の値を自分の方向（Direction）に加えるという変更を行います。上のルールだと、毎ステップ、自分の方向（Direction）に１５ずつ加えることになるので、進行方向から見て左へと方向を変えことになります。

Turn（-15）

右に変更する場合には、上記のように負の値を入れておきます。

以上、第４章

第５章　モデルの設定値をモデル外部から操作する：「神様」になる

◎ エージェント数をコントロールパネルで変える
◎ ルール条件をコントロールパネルで変える

５．１　モデルの条件を簡単に設定し直したい

ここまでで、

（1）エージェントに行動させる
　　（具体例としては「動く」ルール--「方向」と「速さ」--を書き込む）
（2）エージェントに自律的行動をさせる
　　（具体的には「場合分け」で異なる行動様式を選択させる）
（3）エージェントに周囲の環境を調べさせる
　　（具体例としては周囲のエージェントを認識させる）
（4）周囲の環境の状態に応じて、行動を変える（以上の総合）

というマルチエージェント・シミュレーションの基本的考え方を学びました。

その際、エージェントのプロパティを開いて、エージェント数を変えたり、エージェント・ルールエディタを開いて、さまざまな条件の数値を変えたりして、異なる状態でシミュレーションを実行し、違いを比較しました。言い換えれば、マルチエージェント・シミュレーションを本格的に行うことは、基本的なモデルの中の一部をいろいろ変えてみながらシミュレーションを実行して結果に表れる変化を観察することなのです。

artisoc では、さまざまな設定を簡単に変えることが可能な機能を備えています。この章では、いちいちエージェントのプロパティやルールエディタを開くことなし、さまざまな設定を変える方法の基本を学びましょう。具体的には、Universe というツリーの最上位のレベルでさまざま

な操作をすることによって

（1）エージェント数をモデルの外から（コントロールパネルで）設定する
（2）「場合分け」の条件をモデルの外から（コントロールパネルで）設定する

という外からの操作が可能なモデルを作る方法を学びます。

５．２　好きな数だけエージェントを生まれさせる

artisoc でモデルの外から設定した数のエージェントを生まれさせるためには、

（1）Universe にエージェント数を表す変数を作る
（2）その変数をモデルの外から操作できるように、コントロールパネルを作る
（3）Universe のルールエディタにエージェント数だけのエージェントを生成させるルールを書き込む

という手順が必要です。

本来なら、最初から作り始めるのが慣れるためには良いのですが、チュートリアルでは時間がもったいないので、tachibanashi.model を開いてください。それを、tachibanashi2.model と名前を付けて、保存して下さい。hito エージェントのルールは、３２ページのものにしておいてください。

なお、hito エージェントのプロパティで、エージェント数は０にしておいてください。エージェント数は、モデルの外部から指定するからです。

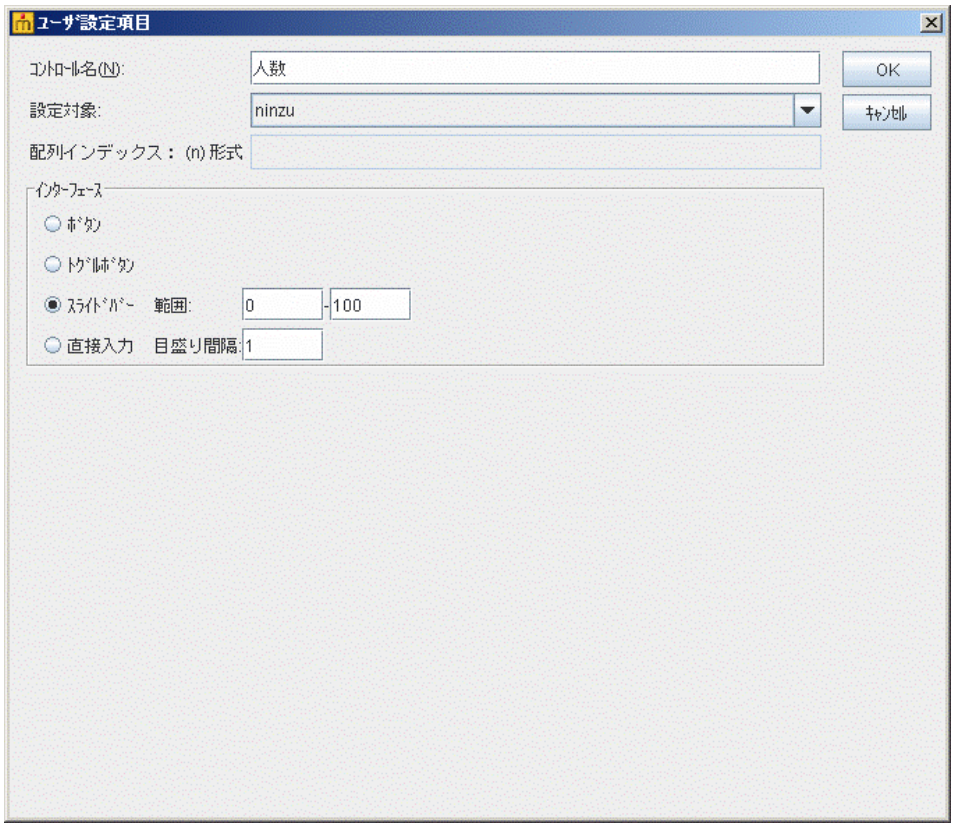
それでは、エージェント数を表す変数（整数型）を ninzu として、（1）（2）（3）の作業をしましょう。

（1）

まず、Universe をクリックして Universe とフォーカスし、メニューから「挿入」をクリックし、変数の追加を選んでください。（または、Universe を右クリックし、「変数の追加」を選んでも、ＯＫです。）すると、変数を設定するためのボックス「変数プロパティ」が開きます。変数名の欄に ninzu と書き込み、変数の型が「整数型」になっているのを確認したら、そのまま「了解」を押して下さい。ツリーの Universe と同じ階層に変数 ninzu ができましたね。

（2）

つぎに、いま作った ninzu の値を簡単に設定できるようにしましょう。メニューの「設定」から「コントロールパネル設定」を開いて下さい。「ユーザ設定項目リスト」が出てきます。ここで「追加」を押すと、新しい「ユーザ設定項目」の画面が出てきます。「コントロール名」に、「人数」と打ち込みましょう。そして、そのすぐ下の「設定対象」で、ninzu を選びます。すると、インタフェースが選べるようになります。いま設定したいのは人数ですから、「スライドバー」か「直接入力」が適しています。今回は「スライドバー」を使いましょう。「範囲」は変数を設定できる幅を示しています。0 から 100 と入れておきましょう。「目盛り間隔」は 1（デフォルト値）のままでも構わないでしょう。「ＯＫ」を押し、ダイアログを閉じて下さい。



これで、コントロールパネルの設定が終わりました。一度保存してから、実行ボタンを押してみましょう。エージェント数が 0 なので何も起こりませんが、コントロールパネルが出てきたと思います。スライドバーのつまみをクリックしたまま横に動かしてみてください。数字が 0 から 100 まで動いたでしょうか? ちなみに、スライドバーはキーボードの「←」「→」の矢印ボタンでも操作できます。試してみて下さい。このスライドバーの値にしたがって、ツリーの中の ninzu という変数の値も変わっていくのです。ここまでの作業で、モデルの外部からモデルの中の変数を操作するインタフェースが完成しました。

（３）

最後に、ninzu という変数の値だけエージェントを作るルールを書くましょう。まず、どのようなルールにするかを明記してから、新しい文法などを説明します。

（１） Universe のルールエディタを開く。
（２） Univ_Init{ と }の間に必要なルールを書き込む。Univ_Init{ と } の間に書き込まれたルールは、シミュレーション実行時の冒頭に１回だけ実行されるルールです。
（３） ルールのポイントは ninzu ぶんのエージェントをどうやって生成するか、です。

```
Univ_Init{
Dim i as Integer
For i = 0 to Universe.ninzu-1
        CreateAgt(Universe.Hiroba.hito)
Next i
}
```

（１）

まず、Dim i as Integer という構文に注目してください。これは、 i という変数を整数型の変数として（as integer）を作ってくださいというルールです。ルールのこのセクション（この場合、Univ_Init{}）でのみ使用する「一時的な変数」が必要な場合、このように宣言文を書いて、変数をつくっておきます。この i という変数は、ルールのこの部分（この場合、Univ_Init{}）でのみ使用可能な変数となります。「ツリーに作成した変数」と異なり、宣言文を書いたセクションのみでしか用いることはできません。また、その値は保存されず、ルールの処理が終わるとその値は初期化されてしまいます。簡単に作って（その値を）使い捨てにする変数というふうに思っておいてください。

（2）

つぎに、For i = 0 to Universe.ninzu -1 XXXXX Next i という構文に注目して下さい。

```
For i = 0 to  Universe.ninzu -1
    XXXXX
Next i
```

これは、XXXXX という作業を　0から Universe.ninzu-1 回（つまり合計 Universe.ninzu 回）くりかえすということです。（ここでは、For i = 1 to Universe.ninzu と書いても、正しく実行されるのですが、artisoc ではさまざまな設定が１からではなく０から始まっているので、他の箇所での汎用性を考慮して、なるべく０から始まるように繰り返し文を書く習慣を身につけて下さい。）なお、Universe.ninzu というのは、「Universe の下にある ninzu という変数」を意味しています。曖昧さをなくすための表記法です。

くりかえす作業の内容は CreateAgt(Universe.Hiroba.hito)ですが、これは Universe の下の Hiroba という空間上で定義されている hito というエージェント種を１固体だけ生み出す、というものです。それを ninzu 回繰り返すわけですから、ninzu 人の hito が生成されたことになります。

それでは実行してみて下さい。コントロールパネルが現れます。取り敢えず、終了して下さい。コントロールパネルは残っているはずです。これからが本番です。Hiroba に登場する hito の数をコントロールパネルで適当に設定しましょう。それから、実行ボタンを押します。

５．３　モデルの中のパラメータを簡単に変える

このモデルでは、hito エージェントの行動は、視野の広さと視野に入る他の hito の数で決まります。このモデルでは、視野の広さや立ち止まるときの最小人数は全ての hito エージェントに共通です。そこで、視野の広さを soba、行動を左右する人数を nakama という変数で表すこと

にして、上の ninzu と同様に、モデルの外からコントロールパネルで設定できるようにしましょう。ここで、注意を要するのは、必ず Universe の直下にこれらの変数を追加する必要がある、という点です。

作業の手順を簡単に説明します。

（１）「設定」から「コントロールパネル設定」を開きます。「ユーザ設定項目リスト」には、先ほど作った「人数」という項目があります。下の「追加」ボタンを押して、新たな項目「視野」(soba を設定対象に)と「仲間」(nakama を設定対象に)を設定しましょう。

（２）次に、hito エージェントのルールエディタを開いて下さい。そして、視野の広さや条件文の中の整数値を、Universe.soba と Universe.nakama という変数に置き換えて下さい。

具体的には、つぎの２カ所です。

```
Agt_Init{
my.X = rnd()*50
my.Y = rnd()*50
my.Direction = rnd()*360
}

Agt_Step{
MakeAllAgtsetAroundOwn(my.Neighbor, Universe.soba, False)
If Universe.nakama <= CountAgtset(my.Neighbor) then

Else
        foward(1)
End if
}
```

＊MakeOneAgtsetAroundOwn を用いている人は、

MakeOneAgtsetAroundOwn(my.Neighbor, Universe.soba, Universe.Hiroba.hito, False)

となります。

保存してから実行しましょう。

実行中に、コントロールパネルで、soba や nakama の値を変化させてみましょう。hito の行動パターンが大きく変化する（急に立ち止まったり、また歩き出したりする）ことが観察できるはずです。これは、モデルの中で（具体的にはエージェント（種）のルールエディタの中で）、Universe.soba や Universe.nakama の値が毎ステップ参照されるルールになっているために、実行途中で変化させると、すぐにそれが行動パターンの変化につながるのです。

５．４　エージェントを生まれさせて、初期配置する

今まで、エージェントの初期配置などの初期値設定は、エージェント自身のルールで行ってきました。いわば、エージェントが個別に自分の居場所を最初に決める、という方式です。これに対して、（神様が）エージェントを生まれさせるのと同様に、（神様が）生まれたエージェントを配置するという方式も可能です。やはり、Universe のルールとして設定します。この方法をこれから学びます。

ここで取り上げるのは、エージェントをシミュレーションの最初にランダムに配置する作業です。Tachibanashi2 モデルを用いて説明しましょう。

（１）hito エージェントのルールエディタを開いて、Agt_Init{ }のなかにある、

```
my.X = rnd()*50
my.Y = rnd()*50
```

の２行を削除して下さい。

（２）Universe のルールエディタを開いて、Univ_Init{ }の先ほど書いたルールの後に次のルールを書き加えて下さい。

```
Dim Hitobito as Agtset
MakeAgtset(Hitobito, Universe.Hiroba.hito)
RandomPutAgtset(Hitobito)
```

１行目は、Hitobito という変数をエージェント集合型変数として(as Agtset)、これから一時的に使うことを宣言しています。これは、４ページ前で学びましたね。２行目は、既に書き込まれているルールに従って生成されているhitoエージェントをHitobitoという変数のなかにしまう操作です。そして３行目は、Hitobito というエージェント集合型の変数にしまわれているエージェントをランダムに配置します。

Dim XXXX as integer や Dim YYYYY as Agtset といった変数の宣言文は、そのルールのセクション（この場合、Univ_Init {}）の冒頭にまとめておくのが「お作法」です。実際には、その変数を使う箇所より前で宣言してあれば、どこでも良いのですが、ミスの原因になるので、ルールのセクションの冒頭にまとめるクセをつけておきましょう。

エージェントのルールとしてできることをなぜわざわざ Universe のルールとして書き換えてみたのでしょうか？書き換えるメリットは何でしょうか？ここで削除したやり方では、空間の大きさが 50x50 に限って、正しいルールです。これに対して、新しく登場した RandomPutAgtset() は空間の大きさをあらかじめ指定していません。したがって、ルールを全く変えずに、空間の大きさを自由に変えることが可能です。

書き換えたモデルを上書き保存して、実際に以前と同じように動くかどうか、実行して確かめて下さい。

５．５　復習用の課題：コントロールパネルの設定に慣れる

この節は復習です。後日、次のようにモデルを修正して下さい。

＜練習問題５．１＞

hito エージェントをもっと多人数（たとえば２００人まで）生成できるように設定を変えてみましょう。その際、スライドバーによる設定ではなく、直接入力で指定できるように設定してみ

ましょう。

＜練習問題５．２＞

tachibanashi_X モデル（前章の復習課題）で、pet エージェントについても、ペットの数やペットの移動速度をモデルの外部から設定できるように修正してみましょう。

必ず実行してみて、思った通りに hito や pet が動いているか確かめて下さい。このように修正したモデルは、上書き保存せずに、tachibanashi2 とは別の名前で保存してください。

 tachibanashi2 モデルは、次章でそのまま使います。

以上、第５章 

第６章　シミュレーションの過程や結果を把握して、表す

◎ 時系列グラフに出力させる
◎ 終了条件を設定する

６．１　シミュレーションの経過をもっと知るには

シミュレーションの実行過程は、今まで、空間をマップ出力して、観察してきました。しかし、この方法だけでは、状態が刻一刻変化する状態は実感できても、どのように変化したかを正確に記録することはできません。

この章では、シミュレーション実行中のモデルの状態を出力する方法を学びます。具体的には、モデルの全体的特徴を把握するために集計作業を行い、その結果を時系列グラフとして見ることを可能にする方法を学びます。

また、シミュレーションを管理したりモニターする方法の初歩も学びます。具体的には、ある条件が満たされら、シミュレーションを修了させる方法を学びます。

６．２　モデルの中で集計する

ここでは、前章で作った tachibanashi2 モデルを使って、モデルの状態をどのように把握するのか、それをどのように表すのか（出力するのか）を学びましょう。その作業の基本は、

（１）　状態を調べるための変数を追加する、
（２）　その変数を出力するための設定をする、
（３）　その変数を利用してモデルの状態を知る方法をルールエディタに書き込む、

です。

モデルの状態としては、各ステップに、どれだけの人数が立ち止まっているのかを調べることにします。

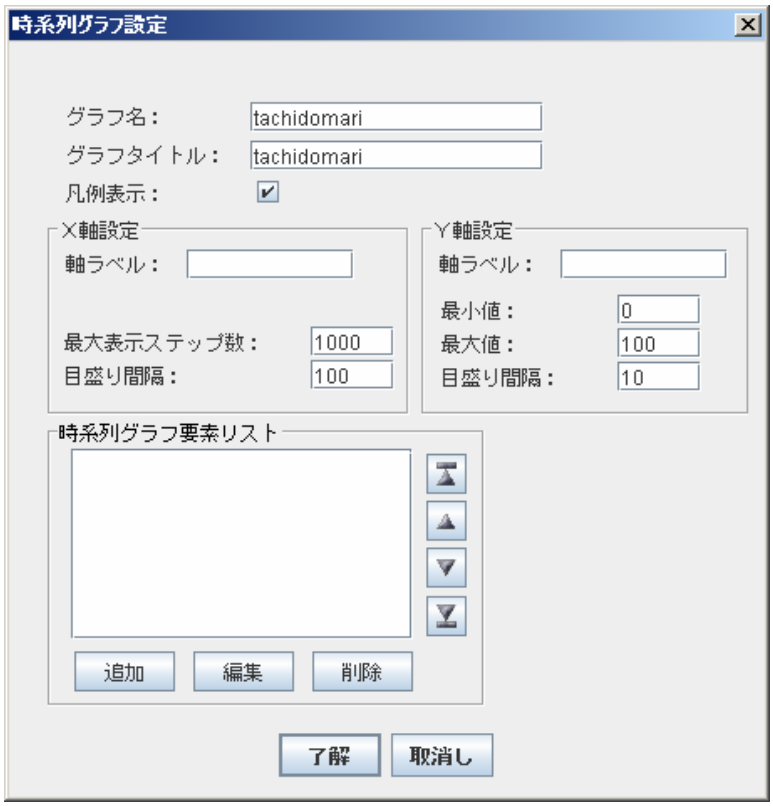
（1）

Universe の下に tachidomari という変数（整数型）を追加する。

（2）

tachidomari を時系列グラフとして出力するように設定する。

「設定」から「出力設定」を開きます。下にある「追加する出力種類」をデフォルトの「マップ出力」から「時系列グラフ」を選びます。そうしておいて「追加」を押してみましょう。「時系列グラフ設定」の画面が開かれたと思います。



グラフ名を「tachidomari」としグラフタイトルも「tachidomari」と入力してください。そうしておいて、「時系列グラフ要素リスト」の「追加」を選択します。時系列グラフで何を表示するのかを指定するわけです。

要素設定画面が開かれると思うので、要素名を「tachidomari」とし、出力値を「Universe.tachidomari」と入力してください。グラフの線の色や線の太さはお好みにしていただいて結構です。

（3）

Universe のルールエディタを開いて、Universe_Step_Begin{ }の中に次のような変数の初期化を毎ステップの冒頭に行う。

```
Univ_Step_Begin{
Universe.tachidomari = 0
}
```

Univ_Step_Begin{ }の中には、毎ステップの冒頭に実行するルールを書き込むます。こうすることによって、tachidomari という立ち話しているエージェントの数を各ステップでゼロに初期化して、そのステップで何人のエージェントが立ち話しているのかを数えることが可能になります。

コラム　『Universe のルールエディタが複雑なのはなぜ？』

エージェントのルールは、Agt_Init {} のルールと Agt_Step {}のルールという二つのセクションから構成されていました。それに対し、ユニバースのルールは、Univ_Init{}、Univ_Step_Begin{}、Univ_Step_End{}、Univ_Finish{}の４つのセクションで構成されています。それぞれ、つぎのようなルールが書き込まれます。

Agt_Init {}	**最初に一度だけ実行させるルール**
Agt_Step {}	**毎ステップ実行させるルール**
Univ_Init{}	**最初に一度だけ実行させるルール**
Univ_Step_Begin{}	**毎ステップの最初に実行させるルール**
Univ_Step_End{}	**毎ステップの最後に実行させるルール**
Univ_Finish{}	**最後に一度だけ実行させるルール**

ルールの処理順序についてまとめてみましょう。

（４）

hito のルールエディタを開いて、次のようにして、立ち止まっている人数をカウント（積算）するルール（Universe.tachidomari = Universe.tachidomari + 1）を挿入する。

```
If Universe.nakama <= CountAgtset(my.Neighbor) then
      Forward(0)
      Universe.tachidomari = Universe.tachidomari + 1
else
```

ここで、Universe.tachidomari = Universe.tachidomari + 1 というルールは、各エージェントについて、動かない場合（forward(0)または何も書き込まない）に、「１」ずつ増やしていく、というものです。（artisoc での「=」は数学のイコールではなく、代入するという操作を表すものだったことを思い出してください。普通の数式（右辺と左辺とを比べる関係式）のように考えると変ですが、これは、現在の Universe.tachidomari の数値に「１」を加えて、その結果を、再び Universe.tachidomari にしまう、というコンピュータの処理を表しています。）

Universe.tachidomari の値は、毎ステップの冒頭、Universe_Step_Begin {} に書き込んだルールによって、ゼロに初期化されますから、各ステップに立ち止まっている hito の数を数えていることになるのです。

以上の変更を終えたら、それを tachibanashi3 という新しい名前で保存しましょう。では実行ボタンを押して、エージェントが動き回るマップと時系列グラフの両方が出力される様子を確認して下さい。

６．３　時系列グラフの利用法

時系列グラフを出力させる方法をうまく利用すると、他の数値をグラフにすることも可能です。時系列グラフの出力設定を開いて、次のような作業をしましょう。

たとえば、立ち止まっているエージェント数の割合（パーセント）を出力したければ、「wariai」という新しいグラフ要素を追加して、

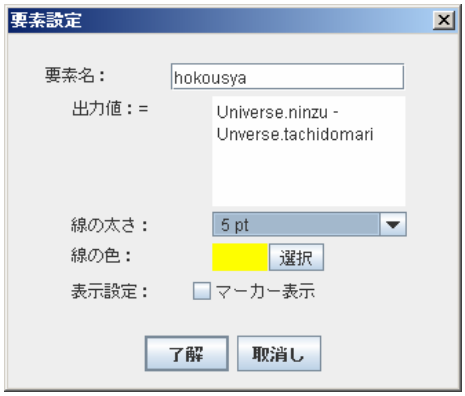
100*Universe.tachidomari/Universe.ninzu

という計算結果を出力させてみましょう。

歩き回っているエージェント数を出力したければ、さらに「hokousha」という新しいグラフ要素を加えて、出力値を

Universe.ninzu - Unverse.tachidomari

にすれば良いですね。



同じ時系列グラフに出力要素を適当な方法で追加すればいろいろな値をグラフで見ることができます。

６．４　自動的にシミュレーションを終了させる

今まで作ってきたモデルは、終了ボタンを押さない限り、シミュレーションはいつまでも実行され続けます。しかし、tachidomari モデル（とその修正版）では、やがて、全てのエージェントが止まってしまい、その後、いつまでシミュレーションを続けても変化ありません。そこで、モデルの中で、シミュレーションを終了させる方法を学びましょう。

シミュレーションを終了させる条件を、全てのエージェントが止まったとき、にしましょう。この条件を毎ステップの最後にチェックすることにします。その場合、Universe_Step_End{}の中にルールを書き込みます。

```
Universe_Step_End{

If Universe.ninzu == Universe.tachidomari then
        Exitsimulation()
End if

}
```

Exitsimulation()というのは、そのステップでシミュレーションを終了させるルールです。hito の数(ninzu)と tachidomari の数とが等しい(==)ときに限り、このルールが活性化します。

６．５　シミュレーション終了時に仕事をさせる

モデルを一定条件が満たされれば、自動的に終了させる方法を学びました。では、何ステップ

目に終了したのでしょうか。何ステップで終了したかをコンソール画面に書き出してみましょう。その方法を以下で学びます。

ひとつの方法は、Universe_Step_End{}の中で、シミュレーション終了条件が満たされたときに、シミュレーションを終了させるだけでなく、コンソール画面に書き出すルールも実行させてしまうことです。

```
Univ_Step_End{

If Universe.ninzu == Universe.tachidomari then
  Exitsimulation()
  PrintLn("Simulation Completed at " & Getcountstep() & "Steps")
End if

}
```

PrintLn("Simulation Completed at" & Getcountstep() & "Steps")は、コンソール画面に"Simulation Completed at" という字句、続いて Getcountstep()という関数の値、そしてさらに "Steps"という字句を書き出すというルールです。Getcountstep()という関数は、そのときのステップ数を入手する関数ですから、この場合、終了した際のステップ数に相当します。とりあえず、ここではこのように写して下さい。

なお、シミュレーションを修了させてメッセージをプリントさせるという作業は

```
ExitsimulationMsgLn("Simulation Completed at" & Getcountstep() & "Steps")
```

とまとめることもできます。

もうひとつの方法もあります。シミュレーションが終了するときだけ、実行されるルールとして、その時のステップ数を書き出すというふうに書くこともできます。

```
Universe_Step_End{
If Universe.ninzu == Universe.tachidomari then
     Exitsimulation()
End if
}

Universe_Finish{
     PrintLn("Simulation Completed at" & Getcountstep() & "Steps")
}
```

Universe_Finish{}の中に書き込まれたルールは、シミュレーション終了時のみ、実行されます。この章のコラムを参照してください。

以上、第６章 

第７章　格子型空間の構造を活用する

◎ ほんとうの「格子型」空間を理解する
◎ 格子があるようにエージェントをふるまわせる

７．１　「格子」がないことを実感する

今まで、oozora とか Hiroba とかいう名前の空間をモデルに作ってきました。実は、これらは「格子型」と呼ばれている空間です。(確認したければ、各モデルを起動して、空間のプロパティを開いてください。デフォルトで「格子モデル」という「空間の型」に設定されています。)しかし、今まで「空間の型」を全く意識せずに（こちらも、この点をあえて説明せずに)、空間を設定してきました。artisoc における格子型空間は、名称は「格子型」となっていますが、格子は存在せず、格子があたかも単なる罫線の役割しかしません。エージェントは格子があることを気にしないで移動できるのです。(ちなみに、もう一つの空間の型である「六角モデル」ではこうはいきません。ハチの巣状の空間構造が設定されて、エージェントは六角形の「穴」をとびとびに移動します。)

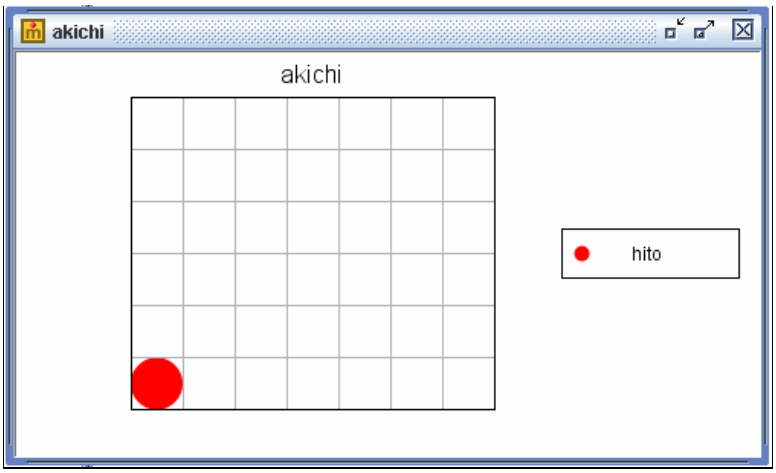
まず、このことを実際にモデルで確認してから、本来の「格子型」として空間を利用する方法を説明します。

空間のデフォルトの大きさは 50×50 ですが、６×５に設定して小さな空間で(言い換えると、大きく見える空間で)、エージェントが実際にどのように動くのかを観察しましょう。

（１）Universe の下に akichi という名前で、Ｘが６，Ｙが５の空間を作ります。空間の種別は格子型（デフォルト）です。
（２）akichi の下に hito エージェントを作ります。エージェント数は 1 にします。
（３）出力設定の作業に入ります。マップ出力（デフォルト）の設定です。設定画面に注目しましょう。罫線表示をクリックし、表示型（チェス型か囲碁型を選ぶ）はチェス型をクリックしてください。今までのように、マップ要素リストに hito エージェントを追加してく

ださい。

このように出力設定をすると、空間の構造が良く分かります。このモデルを、cell-test という名前で保存してください。実行ボタンを押して下さい。格子模様がみえる akichi の左下端の格子の中に hito が重なって存在しているはずです。



さて、ここで画面の表示に注目して下さい。いくつかの重要な特徴があります。

（1）hito エージェントは、まだ何も設定していないので、位置は、Ｘ座標も０，Ｙ座標も０のはずです。（確かめましょう）しかし一目瞭然ですが、原点（左下）ではなく、原点を左下端に持っている格子の中に位置付けられています。

（2）空間の大きさは６ｘ５で設定したはずです。（確かめましょう）格子の数はいくつでしょうか？横に７つ、縦に６つ並んでいます。

実は､エージェントを図示する図形の中心はエージェントの位置ではなく､それより右に０.５，上に０．５ずれているのです。そうすると、右上端に位置するエージェントを図示する図形はどこに来るのでしょうか。エージェントは（６，５）にあるのですから、そのアイコンは（６．５，５.５)が中心となります｡すると６Ｘ５で空間を表示するとアイコンは図示されなくなります。そのようなことが起こらないように、artisoc では、マップ出力を設定すると、自動的に最右列と最上行が「のりしろ」になるように、７Ｘ６で図示されるのです。なぜ、このように面倒なことをするのか（見かけをわざわざずらすのか)、は次節で格子型を活かすルールを学ぶときに分かります。

エージェントを動かしてみましょう。hito エージェントのルールエディタを開いて、

```
Agt_Init{
my.X = rnd()*6
my.Y = rnd()*5
my.Direction = 30
}

Agt_Step{
forward(0.3)
}
```

と書き込んで下さい。

保存してから、「ステップ実行ボタン」を押して下さい。（このボタンを押すと、１ステップだけ実行されます。）エージェントがどこにいるかを確認しながら、ステップ実行ボタンを一回ずつ押して、エージェントの動きを観察して下さい。特に次の点に注目して下さい。

（１）エージェントを表すアイコンと罫線とが重なることもある（Ｘ座標やＹ座標は整数ではない）
（２）右や上の端で一瞬消えて、同時に下や左の端に現れる（空間がループしている）
（３）動き方も細かい（小数単位で毎ステップ移動できる）

このように、格子状に罫線が入っていても、エージェントの位置や動きは、格子を単位にしているのではなく、まるで実数空間のような構造（厳密に言うと、コンピュータのなかですから真の意味の意味での実数はあり得ないのですが）をしているのです。言い換えると、格子型空間になってはいるが、格子を無視してエージェントは動き回ることができるのです。

artisoc がこのような設定になっていることは、モデルを作る側（私たち）に大きな自由度を与えてくれています。一方で、あえて空間を本当の「格子型」としてモデル化するときには、今

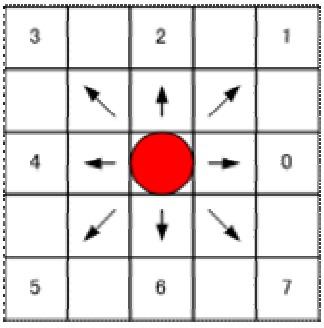
までとは少し異なる技法が必要になるのです。

７．２　「格子」があるようにエージェントを動かす

格子型の空間というのは、将棋盤（チェス・ボード）のように空間が格子状に区切られている空間です。そして格子で区切られた枠のひとつひとつがエージェントのいることができる場所になっているような構造なっています。（格子状に区切られていても、碁盤のように交点のみがエージェントのいることができる場所であると考えることも可能です。）格子型の空間では、エージェントの存在する場所がちょうど独房（セル）になっているので、セル型空間と呼ばれることもあります。（ちなみに、六角モデルの枠、つまりハチの巣の「巣室」もセルです。さらに付け加えれば、セル・オートマトンのセルにも通じています。）

それではいよいよ空間が格子型になっていることが意味を持つように、エージェントの動きに制約を加えることにしましょう。

まず、エージェントは格子から格子へと、とびとびにしか移動できません。（厳密には、「格子から格子」ではなく、「格子のひとつの枠から別の枠」へと移動すると言わなくてはなりません。つまり、セルからセルへ、というわけです。しかし、エージェントは格子の交点から交点にしか移動できないと考えることも可能です。そこで、格子、セル、交点は、エージェントの「居場所」という点で同じ意味で使うことにしましょう。）したがって、移動距離はとびとびの整数値でなくてはなりません。たとえば、foward()のかっこ内の数値にしても、横（Ｘ）方向や縦（Ｙ）方向なら整数値に、４５度方向なら１．４１（２の平方根）にする必要があります。また、移動方向も３６０度どこでも良いというわけには行きませんから、この関数を使うわけにはいきません。ある格子にいるエージェントから見ると、周囲の格子は８個ですから、移動できる方向も８通りしかありません。（ここでは、将棋の桂馬（チェスのナイト）のような動きは除外します。）



では、格子の中をとびとびに移動するエージェントを作ってみましょう。ルールを次のように変えて下さい。

```
Agt_Init{
my.X = Int(rnd()*6)
my.Y = Int(rnd()*5)
}

Agt_Step{
ForwardDirectionCell(0, 1)
}
```

最初の部分は、格子のどこかにランダムに配置するために、座標の値を整数化しています。次に、動く方角（Direction）を決めるルールを削除しています。その代わりに、毎ステップ移動するルールが新しい表現になって、そこに動く方角（Direction）も埋め込まれています。また、格子型（セル型）に対応（cell）しています。つまり、

```
ForwardDirectionCell(0, 1)
```

という新しい関数になっています。これは、エージェントを毎ステップ、右方向（上図の「0」方向）に「1」歩（1枠分）だけ動かす関数です。

保存して「ステップ実行」して下さい。エージェントの動きを確認しましょう。これで、格子型空間の上を格子の中だけ動き回るエージェントの特徴と、格子型空間の特徴がつかめたことと思います。ForwardDirectionCell()関数の最初の数値（現在は0）を他の数値（7以下の整数）に変えて、エージェントの動き方がどのように変わるかを、上の図と比べながら、確認して下さい。

７．３　格子を利用したモデルを作る

ここで、格子型空間の構造を活用したエージェントの動かし方を学びましょう。ある劇場に１０×２０の椅子があり、そこにモデルの外から操作するｎ人が観劇に訪れるが、周囲に人がいて混んでくると空いている席に移動するモデルを作ってみましょう。まず準備作業です。

（１）Universe の下に Gekijo 空間を 10x20 の大きさで設定する（ループなし）。
（２）Universe の下に整数型変数 Kankyaku を作る
（３）Gekijo の下に hito を作る（エージェント数は０）
（４）マップ出力を設定する（罫線あり、チェス型）
（５）Kankyaku を０から１２０人までの範囲で変えられるようにコントロールパネルを設定する

Eigakan というモデルの名前で保存して下さい。

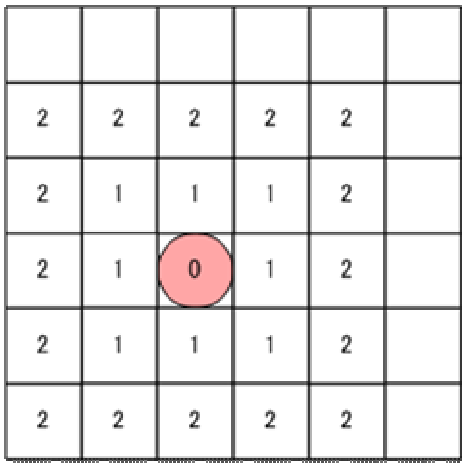
まず、Univ_Init に以下のルールを加えてください。
Kankyaku の数だけエージェントを発生させるルールです。

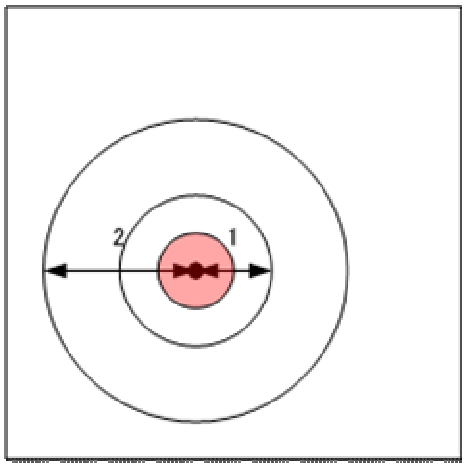
```
Univ_Init{
Dim i as integer
For i = 0 to Universe.Kankyaku-1
     CreateAgt(Universe.Gekijo.hito)
Next i
}
```

次に、エージェント（hito）の動くルールを書き込む作業です。

```
Agt_Step{
Dim Mawari as Agtset
MakeAllAgtsetAroundOwnCell(Mawari, 1, false)
If 4 <= CountAgtset(Mawari) then
      MovetoSpaceOwnCell(3)
End if
}
```

上のルールは既に第４章で学んだ形（周りの様子を認識し、状態に応じて行動を変える）の応用です。そのなかで、MakeAllAgtsetAroundOwnCell(Mawari, 1, false)は、既に使ったことのあるMakeAllAgtsetAroundOwn()の「格子(cell)」版です。MovetoSpaceOwnCell(3)は初めて出てきた関数ですが、格子型で視野「３」の範囲で、空いている格子があれば、そこに動くというルールです。





格子状の「視野」は、今までの視野の捉え方と少し異なります。下に、その違いを図示してあります。

さて、上のルールでは、自分の周囲（視野が「１」に設定されているので、隣接している８つの格子）に４人以上いると、込みすぎだと感じて、近くの（周囲４８席のなかで）空いている席に移動します。このモデルを保存してから、実行しましょう。観客数が少なければ、すぐに移動

するエージェントがいなくなりますが､多いとなかなか落ち着かなくなるのが観察できるはずです。

７．４　「格子」の扱いに慣れる

我慢の限度（このモデルでは、周囲の混雑具合の認識と空席をさがす範囲）には個人差があります。そこで、混み具合と空席をさがす範囲を変数化して、エージェント毎に乱数を用いて異なる数値を与えてみましょう。

hito エージェントに、kyukutsu と kyorokyoro という変数を追加します。今のままだと、kyukutsu の値は「４」、kyorokyoro の値は「３」です。シミュレーションの最初に、各エージェントが kyukutsu の値を４以上７以下、kyorokyoro の値を２か３をとるように設定し、エージェントのルールを全て書き換えて下さい。

```
Agt_Init{
my.kyukutsu = 4 + Int(rnd()*4)
my.kyorokyoro = 2 + Int(rnd()*2)
}

Agt_Step{
Dim Mawari as Agtset
MakeAllAgtsetAroundOwnCell(Mawari, 1, False)
If my.kyukutsu <= CountAgtset(Mawari) then
     MovetospaceOwnCell(my.kyorokyoro)
End if
}
```

もし、混雑が気になる人は広範囲で空席をさがす、と仮定すると、上のルールではこの仮定がうまく設定されていません。たとえば、次のように変えてみましょう。

```
Agt_Init{
my.kyukutsu = 4 + Int(rnd()*4)
my.kyorokyoro = 3 – my.kyukutsu/6
}
```

my.kyukutsu は４から７までの整数値ですから、 my.kyukutsu /6 は、０か１になります。したがって、my,kyorokyoro の値は、混雑を我慢できない人(kyukutsu が４か５)は視野「３」、我慢強い人(kyukutsu が６か７)は視野「２」で空席をさがすことになります。

７．５　復習を兼ねた応用問題

モデルの内部状態を調べましょう。ここは第６章の応用です。
満足している観客数を数えて、時系列グラフで出力して下さい。

（１）Universe に整数型変数設定
（２）時系列グラフの出力設定
（３）Universe のルールで毎ステップ初期化（ゼロに）
（４）Hito のルールで、満足している観客を勘定

たとえば、Manzoku という変数を用いると仮定すると

```
If my.kyukutsu <= CountAgtset(Mawari) then
      MovetospaceOwnCell(my.kyorokyoro)
Else
     Universe.manzoku = Universe.manzoku + 1
End if
```

というふうにルールを修正すれば完了です。

７．６　復習用の課題

後日、試みてください。
まず、観客が全員満足したらシミュレーションが終了するルールを考えてみましょう。
Eigakan モデルは、観客数が多すぎるといつまでたっても全員が満足する状態になりません。そこで、１００ステップになったら、自動的にシミュレーションを終了するように終了条件を追加して下さい。

```
Univ_Step_End{
If Universe.manzoku >= Universe.Kankyaku then
    Exitsimulation()

Elseif  100<= Getcountstep() then
    Exitsimulation()
Else

End if
}
```

シミュレーション終了時にメッセージをコンソール画面に出しましょう。たとえば、次のようなものです。これも第６章の応用です。

```
If Universe.manzoku >= Universe.Kankyaku then
    Exitsimulation()
    PrintLn("All guests satisfied at " & Getcountstep() & " steps")
Elseif  100<= Getcountstep() then
    Exitsimulation()
    PrintLn("Not all guests satisfied until 100 steps")
End if
```

以上、第７章 

終章　名作を自分で作ろう：シェリングの「分居モデル」に挑戦！

◎ 新しく学ぶ技法はありません
◎ アイデアをモデルに結びつける練習です

今まで学んできたマルチエージェント・シミュレーションの技法だけで、シェリングの分居モデルが作れます。ノーベル賞学者の鮮やかな着想とモデル化のプロセスを後追いしましょう。そして、コンピュータ技術の発達を利用して、オリジナル・モデルを改良してみましょう。

８．１　オリジナルモデルの説明

トマス・シェリングは、おそらく最初にマルチエージェント・シミュレーションの手法を社会科学にとりいれた研究者です。彼は、アメリカの都市で人々が民族集団ごとに分かれて生活する現象（チャイナタウンやリトルトウキョウ）を取り上げ、特に個々人の民族差別意識がさほど強くなくても、地域社会が画然と分かれるという点に注目しました。

そして、メカニズムをシミュレーションという手法を使って解明しようとしたのです。モデルの中身はあとで見てみることとしましょう。簡単なルールで社会現象の本質的な部分を表現しており、個々人の意思とその相互作用で生まれる社会現象の関係をうまくとらえた古典といえます。

ただ、シェリングが研究をおこなった１９６０年代には、今日のような計算能力のある PC はなく、そして artisoc はありませんでした。

８．２　オリジナルモデルの再現

シェリングはコイン、サイコロと本物のチェス盤を使いました。わたしたちは、それを artisoc のモデルとして再現しましょう。

まず、例によって準備作業から入りましょう。

（１）　Universe の下に chessboard とい名前の空間を８ｘ８で作る

（２）　chessboard の下に penny を２３個、dime を２２個作る。

（３）　Universe の下に satisfied という整数型変数を作る

（４）　出力設定の作業として、chessboard のマップ出力（サイズを７ｘ７，罫線あり、チェス型など）と satisfied の時系列グラフ出力（目盛り間隔１０）を設定する。

（５）　shellingoriginal という名前で保存する

（６）　実行ボタンを押して、設定が正常か確認（終了ボタンを忘れずに）

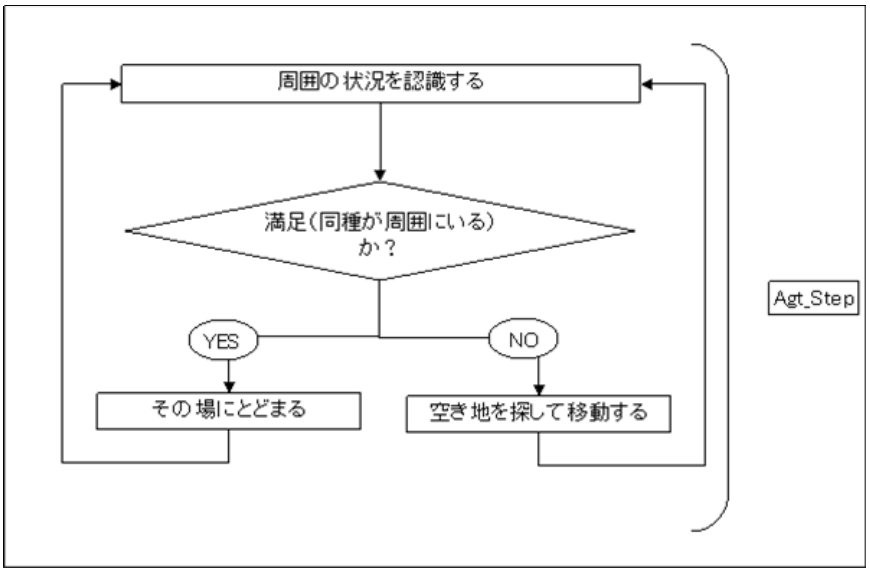
次にルールを書き込む作業に入りましょう。

（１）コインをチェス盤にランダムに置くルールを書き込む

```
Univ_Init{

Dim i as Integer
Dim coin as Agtset

For i = 0 to 22
  CreateAgt(Universe.chessboard.penny)
Next i
For i = 0 to 21
  Createagt(Universe.chessboard.dime)
Next i

MakeAgtsetSpace(coin, universe.chessboard)
RandomPutAgtsetCell(coin, false)

}
```

（２）１セント硬貨(penny)のルールを書き込む

・周囲の状況を認識し、満足かどうか判断させる。不満な場合には、空き地を探して移動する。
（フローチャートで書くと以下のようになります）



・満足な場合には、時系列グラフに出力させるための計算作業を行う（この部分は、厳密に言うとエージェントの行動ルールではなく、モデルを作る側の操作です）。

```
Agt_Init{
}
Agt_Step{
Dim Neighbor as Agtset
MakeOneAgtsetAroundOwnCell(Neighbor, 1, Universe.chessboard.penny,
  false)
If  3 <= countagtset(neighbor) then
  Universe.satisfied = Universe.satisfied + 1
Else
  MovetospaceOwnCell(2)
End if

}
```

（３）１０セント硬貨(dime)についても同様のルールを書き込む。

１カ所だけ違います。

```
Agt_Init{
}
Agt_Step{

Dim Neighbor as Agtset
MakeOneAgtsetAroundOwnCell(Neighbor, 1, Universe.chessboard.dime,
  false)
If  3 <= countagtset(neighbor) then
  Universe.satisfied = Universe.satisfied + 1
Else
  MovetospaceOwnCell(2)
End if

}
```

あとは同じ。

（４）満足しているエージェントをカウントし、時系列グラフに出力させるための初期化をルール化する。

```
Univ_Step_Begin{
Universe.satisfied = 0
}
```

（５）シミュレーションの修了条件を整えるルールを書き込む。

```
Univ_Step_End{

If 45 == universe.satisfied then
   Exitsimulation()
   PrintLn("Every one satisfied at " & getcountstep() & " steps")
Elseif 100<= getcountstep() then
   Exitsimulation()
   PrintLn("Not all satisfied until 100 steps")
End if

}
```

これで完了です。保存しましょう。シミュレーションは自動的に終了しますから、何回でも実行ボタンを押して、サイコロを振る手間が無くなった幸福を味わって下さい。

８．３　若干の補足

以上のモデルは、実は厳密に言うと、シェリングのオリジナルなモデルと２点で違っています。

第１点は、満足か不満かを認識する方法です。シェリングは、不満なる閾値を３分の１とし、周囲の格子の数を分母にしました。したがって、周囲を格子で囲まれているエージェントは異分子が５以上で不満になりますが、端のエージェントは周囲の格子が５ですから４以上で、四隅のエージェントは周囲が３ですから２つ以上異分子が占拠していると不満になり、移動しようと思います。

これは、「場合分け」で処理する問題です。artisoc のルールで再現することは可能ですが、ここでは問題の本質をクリアにするために、そのような処理をせずに、空間がループしているものと考えましょう。そうすると、どこに位置するエージェントにとっても、周囲の格子は８にな

ります。上のモデル作りでは、空間を設定するときにデフォルトの「ループあり」を選択していたのです。

第２点は、不満な場合の移動方法です。シェリングは不満なコインはどこかの空き地に移動するルールを用いています。ここでは、視野「２」で空き地を探すようにルール化しました。ループしているチェス盤全体を見回すには、視野「４」が必要（３でほぼ全体を見回せるが）です。しかし、こうすると、すぐに全てのコインが満足してしまいます。（実際に、視野を「３」に書き換えて、実行してみて下さい）

シェリングは、コインが空き地を探し回るプロセスではなく、分居してしまうという結果を分析したかったので、視野の問題は重要ではありませんでした。artisoc では、結果を求めることも大事ですが、過程を観察できるというメリットがあります。そこで、取り敢えず、視野を「２」に設定してみました。

８．４　シェリングの分析を再現する

シェリングは、不満なエージェントが移動する結果、全体として分居が進むという現象を明らかにしようとしました。そこで、分居の程度を表す指標が必要になります。分居の指標としては、チェス盤全体で異なるコイン同氏が隣接している（対角線方向も含めて）箇所を集計するというやり方があります。各エージェントが異種エージェントといくつ隣接しているのかは、既にモデルの中で計算しています（エージェントのルールエディタの中で一時的に用いた neighbor）。それを利用して、分居度（百分率）を求めてみましょう。

（１）各ステップで、各エージェントにとって隣接している異種エージェントの数を足し合わせる。
（２）隣接している格子の全体からその数を引くと、異種エージェントが存在しない隣接格子の数になる
（３）上で求めた数を隣接している格子の全体で割り、パーセント化する。

ここで（１）の作業は、ルールエディタで、（２）（３）の作業は時系列グラフの出力要素の設定で行います。そこで、

（１） Universe に整数型変数 stranger を追加する

（２） Univ_Step_Begin で毎ステップ初期化する（ゼロにする）

（３） penny と dime の両方のルールエディタの適当な箇所（自分で見つけて下さい）に

```
Universe.stranger = Universe.stranger + Countagtset(neighbor)
```

を挿入する

（４） 時系列グラフに要素リストを追加（たとえば、Bunkyodo）し、出力値の欄に

```
100*(8*45 - Universe.stranger)/(8*45)
```

と書き込む。（ここで、わざわざ８＊４５としているのは、エージェントの数を以下の改良作業で可変にするときに修正を容易にするためです。）

８．５　モデルの改良

 後日、試みてください。

＜練習問題　８．１＞

空間を広くしてみる。

＜練習問題　８．２＞

２種類のエージェント数をコントロールパネルで操作できるようにする。

＜練習問題　８．３＞

エージェントのもつ視野をコントロールパネルで操作できるようにする。

以上、第８章