Model Building

Gabriel Wurzer,
Vienna University of Technology

www.iemar.tuwien.ac.at
PREFACE

or: filling the academical quarter
When things go wrong, or:

HOW TO DEBUG
Debugging

Today meaning
„to fix software errors“

Previous meaning:


Rear Admiral Grace Hopper, US Navy

NetLogo debugging capabilities

...are astonishingly limited:

• **print** value… prints a value to the command center area, e.g.

```ask turtle 0 [print shape]```

• Concatenation may happen for adding a pre- or suffix to the printed value, using the **word** command:

```print (word ,,The shape is ,, shape)```
Further debugging...

- **self** reports the breed followed by the who number of the turtle, (e.g. „turtle 0“)
- **watch** puts a shiny spot around a turtle
- if you want to watch one randomly chosen agent out of a population, you can use the reporter **one-of**
- in fact, **one-of** is a general command for choosing one member of a population

```
ask turtles with [...] [
  print self
]

watch turtle 0

watch one-of turtles
```
Querying somebody else‘s properties

Often, you „are“ observer or a specific turtle or patch. Querying somebody else‘s values can be done using \texttt{of}:

\begin{verbatim}
print [shape] \texttt{of} turtle 0
\end{verbatim}

\texttt{of} is used to access a property of an entity given by you.

Examples:

\begin{verbatim}
print [pcolor] \texttt{of} patch 0 0
ask patch 0 0 [
    set pcolor [pcolor] \texttt{of} patch 1 1
]
\end{verbatim}
Comparing `ask` and `of`

```
ask patch 0 0 [
  print pcolor ;you are patch 0 0
]
```

; you are observer

```
print [pcolor] of patch 0 0
```

`ask` enters an entity. Therefore, can access a property by giving its name (e.g. `print pcolor`)

`of` accesses a property of an entity you give – you need not be „inside“ the entity

Hint: yes, `of` is a reporter, you might have guessed it!

Using **of** for debugging

- Add a monitor to the Interface
- In the appearing dialog, you can type any query. Type the following query and hit OK:

  ```plaintext
  [heading] of turtle 0
  286
  ```

- The result of this query will be shown in the textfield of the monitor.
ASSIGNMENT
The eXercise Files

- Assignment files are given to you in a folder named „assignment“.
- Please open modeling-exercise-0.nlogo in NetLogo, which we will now elaborate and then extend, one step at a time.
Features used in this version of the model...

- An overlay Image
- Importing per-patch data from a graphics file
General Pattern

OVERLAY IMAGE
Overlaying an image

Can import an image (just for eye candy, does influence NetLogo) using

import-drawing „name.png“

Can clear the imported image using

clear-drawing
Patch pattern

IMPORTING PER-PATCH DATA
Usage

- Data that must be available *per patch* (i.e. in some previously defined properties) should be imported into NetLogo.
- This data is generated e.g. in a GIS or drawing program.
- There can be a variety of these layers, e.g. one per resource (e.g. grain, ore, water).
Exporting the view...

This is a pre-step for importing:

1. Change the pcolor of some patches by right-clicking and choosing „inspect“, then set pcolor e.g. to YELLOW
2. Right-click the view area and choose „Export View“
The exported image is exactly as the NetLogo view.

• Can edit this file in a graphics program and re-import using:

```
import-pcolors ,,name.png```

• The colors found in the file will be mapped to the nearest NetLogo color, with fine details being lost.
Hint: Pixel-exact mapping

Can also do a pixel-exact import when clicking on Settings in Interface, then specifying:

- max-pxcor half image width
- max-pycor half image height
- Patch-size 1 pixels
Mapping color to a value – existence of grain

patches-own [  
  grain  
]
...
ask patches [  
  set grain 0  
  if pcolor = YELLOW [  
    set grain 1  
  ]  
]

if pcolor = YELLOW ...
A closer look at colors

Colors in NetLogo are really numbers. Names like YELLOW are only shorthands.
Greys are interesting

- The color 0 stands for black. This is the default color for patches.
- The color 9.9 stands for white, and all other colors in between 0 and 9.9 are grays.

Observation:
Percentages go from 0-100,
NetLogo colors from 0-9.9
Hmmmmm.....
Mapping color to a value – quantity of grain

patches-own [ 
grain ]
...
ask patches [ 
set grain pcolor ]

Hint:
could also use a scale factor, 
e.g. grain pcolor * scale-factor 10
Mapping color to a value – percent of grain

patches-own [ grain ]
...
ask patches [ set grain (pcolor / 9.9) * 100

* max-grain 50 ]
Next step in the model...

- Agents (mines) are introduced
- Custom turtle shapes
- Temporary storage is introduced
Turtle Pattern

CUSTOM TURTLE SHAPES
Custom shapes for turtles

- From Tools, choose Turtle Shapes Editor
- Click on „New“ to add a shape
The shape editor in brief

- Draw your shape
- In the name field, assign a name
- Should the shape rotate with the heading of the turtle, click on Rotatable
- Under „Color that changes“, give the color that is set to the turtle‘s color
Assigning your newly drawn shape

... create-turtles 1 [ set shape „my shape“ ] ...

Hint: if you do not want your shape to change color, set „Color that changes“ to some color that does not appear in the shape.
TEMPORARY STORAGE
Storing values - revisited

• So far, two type of storages have been introduced:

  – properties of an entity

  – values entered via the user interface (called global variables)
The need for an intermediate storage

intermediate storage makes code easier to read, e.g.:

1. get all turtles which meet a certain criterium
2. filter this intermediate result by a second criterium
3. ask the remaining turtles to do something

example of code which uses intermediate storage

```
to example
  introduce [candidates]
  set [candidates] to turtles with[...]
  ask [candidates] with [...] [ ...
  ]
end
```
Lifespan of intermediate storage is temporary:
the storage is thrown away once the code block that introduced it ends.

```
to example
  introduce candidates
  set candidates to turtles with [...] [ candidates
  ask candidates with [...] [ ...
  ]
end
```
Show me the code...

- an intermediate storage is called „local variable“
- it is introduced using `let`:

```netlogo
let candidates turtles with[...] ask candidates with [...] [
...
]
end
```

- to access the value stored in a variable, its name (`candidates`) is given.
can assign the following values to a variable:

- a concrete value (e.g. 10, false, „person“)
- the value NOBODY, i.e. „no value set“
- a result of a reporter or query
Examples with variables

**Calculation using variables**

let a 12
let b 20
let c (a + b)
print (word ,,c is “ c)

**Incrementing a variable using**

set:

let d 0
set d (d + 1)
print (word ,,d is “ d)

**Setting a variable to ,,no value“**

let e NOBODY
print (word ,,e is “ e)

**Setting a variable to the result of a query or reporter**

let e turtles with [color = SKY]  
let f sin 30  
print (word ,,e is “ e) 
ask e [ print self ]  
print (word ,,f is “ f)
When variables take up agentsets

- with [...] reports a set of turtles or patches
- the returned set is called *agentset*
- you can test whether the set is empty by using the keyword *any*?
- you can refer to the contained agents by asking the agentset

```plaintext
let redones turtles with [color = RED]
if any? redones [ ask redones [print self] ]
```

Hint: The keyword for addressing a population (e.g. turtles, patches, persons, etc.) return agentsets as well, so you might well write: if not any? turtles [ print „All dead.“ ]
Next step in the model...

- Agents (settlements) are introduced
- Reporters are introduced
- Interactions are introduced
WRITING REPORTERS AND USING PARAMETERS
A tale of users and programmers

User of sin

to setup
print sin 30
end

Programmer of sin

to sinus (x)
1. \[ \sum_{n=0}^{\infty} (-1)^n \frac{x^{2n+1}}{(2n+1)!} = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} + \ldots \]
2. give back result to user
end

• We have seen how to use reporters
• As mentioned, „something is being done behind the scenes“, i.e.
  – the parameter or parameters are passed into a reporter
  – this is used like a variable
  – the result is given back to the user
Don‘t be a user – write reporters yourself

• you can write reporters yourself:

```
to-report add [x y]
  let result (x + y)
  report result
end
```

Usage:
```
to-report name [parameters]
  ; calculate something here
  ; using the parameters
  report result
end
```

• beneficial for „hiding the details“ of your model:

```
to setup
  print water-patches
end
```

```
to-report water-patches
  report patches with [color = SKY]
end
```

Parameters

• ...can also be used in procedures, by the way:

```
to move-turtles [ in-direction how-far ]
  ask turtles [  
    set heading in-direction  
    forward how-far  
  ]
end
```
Recap – When to use what

Write a procedure
• to group code lines together under a sounding name (e.g. move-turtles)

Write a reporter
• to perform a calculation and return the result to the user (e.g. add)

Use parameters whenever your code depends on values that need to be supplied by the user.

Hint: Think of parameters as local variables, which are not defined using „let“ but given at the start of your procedure or reporter.
LINKING TURTLES
Usage

Links two turtles together to signify a (social?) relationship between them

- Should be able to query for „the turtle at the other side of the link“
Introducing links

Links are an own breed, which is created by a turtle:

```plaintext
ask person 0 [ create-link-with turtle 1 ]
```

Any breed can be linked. Links can be both directed and undirected, although we will only cover the latter ones in this tutorial (refer to NetLogo Dictionary for directed ones). Furthermore, you can create breeds of links, which is also not covered in this course.
How the rest is done...

```plaintext
ask person 0 [
    create-link-with turtle 1
    create-link-with turtle 2
]

ask person 0 [
    ask link-neighbors with [...] [
        print self
    ]
]
```

- **link-neighbors** reports all turtles connected to the current turtle by a link
- as always, this agentset can be further narrowed down using with [...]
Co-create dependent turtles

breed [persons person]
...
create-persons 1 [ 
  let holder self
  hatch-turtles 1 [ 
    create-link-with holder
    set xcor random-xcor
    set ycor random-ycor
  ]
]

Wish to create persons and turtles at the same time, tied together (Note: it is a house-trained turtle we create).

• turtle is created by the person using hatch-turtles, which is just like create-turtles (but can be used by turtles)
• laying the turtle on the person‘s lash is done by creating a link to the holder
Patch pattern

NEIGHBORS
Usage

ask *patch 0 0 [ ask neighbors [ … ] ]

- A patch needs to address its neighbors
  - Should be possible to either get the 8-neighborhood, 4-neighborhood or patches in any radius
  - Additional criteria should be possible using with, e.g. *with [not any? turtles-here]
How it is done...

ask patch 0 0 [  
  ask [neighbors] with[...] []  
  ask [neighbors4] with[...] []  
  ask patches [in-radius] 4 with[...][]  
]  

• **neighbors** reports all 8 patches that surround the calling patch, **neighbors4** only the four patches in the north, south, west and east.

• **in-radius** is a reporter that filters the agentset given on the left (patches) so that only those members being within the given radius of the caller remain

Hint: in-radius can also be used for turtles, i.e. turtles in-radius 4

DISCOVERING WHO’S (ALSO) HERE

Turtle and patch pattern
Usage

ask turtle or patch [  
  get all other turtles at this spot
  …
]

• A turtle or patch wants to know which turtles stand here
  • Should also be able to select other breeds than turtles
  • Reported entities should be „all on this spot, excluding caller“
How it is done...

... ask turtle 0 [ ask other turtles-here [ print self ] ]

ask patch 0 0 [ ask turtles-here [ print self ] ]

• **turtles-here** is a reporter giving all agents on the current spot as agentset

• if the caller is an agent, he can exclude himself by using „other“

• may specifically target other breeds than turtles using *breed-name*-here (e.g. other persons-here)
Turtle Pattern

INTERACTION PATTERN
Usage

- An agent wants to interact with other agents or patches, probably exchanging some values
- In the course of the interactions, multiple nested asks are used, i.e. the first entity is asked, to...
  ask a second entity, to...
  do something

```
ask turtle 0 [
  ask patch 0 0 [
    ...
  ]
]
```
How it is done...

```
ask turtle 0 [  
    let the-turtle self
    ask patch 0 0 [  
        set pcolor [color] of the-turtle
    ]
]
```

1. The caller temporarily stores himself in a local variable.
2. Then, the patch is asked.
3. Inside the ask, a property of the caller is retrieved using `of`.
4. This value is then assigned to a property of the patch.

Hands on: Dropping a value

turtles-own [  
  carries
]
patches-own [  
  stores
]

to setup
  ask patches [  
    set stores 0
  ]
  create-turtles 1 [  
    set carries 10
  ]
end

stores = 0

carries = 1
Hands on: Dropping a value

to go
ask turtles [ forward 1
    if carries > 0 [ set carries (carries - 1)
        let the-turtle self
    ask [patch-here]
        set stores (stores + 1)
        set pcolor [color] of the-turtle
    ]
]
end

reports the patch under the turtle
Hands on: Retrieving a value

turtles-own [
  carries
]
patches-own [
  stores
]

---
to setup
  ask patches [
    set stores 10
  ]
  create-turtles 1 [
    set carries 0
  ]
end

stores = 10
carries = 0
Hands on: Retrieving a value

to go
ask turtles [forward 1
  if [stores] of patch-here > 0 [
    set carries (carries + 1)
    ask patch-here [
      set stores (stores - 1)
    ]
  ]
end
Next step in the model...

- Settlements are moved (rewrite of go), then re-established
Next step in the model...

- Distance-dependent resource contribution by settlements
- Uses Loops to create „as many settlements as required“
LOOPS
So far...

• The button repeated the code contained in the procedure „go“ until it was depressed unpressed again.

• Sometimes, however, you will wish to repeat a set of code lines based on some boolean condition, e.g.:

```plaintext
to go
  while a settlement is needed
    add a settlement
  end
Usage:
while [ condition ] [ run these commands ]
```

A boolean reporter. The details are hidden from you, it simply tells you if one more settlement is needed (true / false)
Example: doing something a number of times

let counter 0
while [ counter < 10 ] [  
  print (word ,,the counter is “ counter) 
  set counter (counter + 1) 
]

Hint: You should not forget this, or else counter will never reach 10, resulting in an endless loop
Last step in the model...

- Spots near water are prioritized in an elegant way ($\Leftrightarrow$ beauty of code)
ASK ARCHEOLOGISTS [ THANK YOU FOR ATTENDING AIA11 ]