Agent-based modelling + How to program in one e-z lesson

Ben Klemens
How to Program

[As taught by a theorist]

- Data structures
- functions
- function contents
- frames, scope, & encapsulation
- compilation and/or execution
Data structures

The basic types

- **int**: an integer
- **float**: a real number (with a floating decimal point)
- **char**: a character. Java version: string
Data structures

A combination of types, clumped into one header.

typedef struct ppp{
    char* first_name, last_name;
    float height, weight;
    int age;
} person;

Almost all languages call subelements with a dot:

person steve;
steve.height = 175.8;
steve.age = 40;
Data structures

arrays

A numbered list of either pure types or structures.

float grades[10];
person survey_data[200];

grades[3] = 0.68;
survey_data[40].height = 160;
Functions

The black box

All functions take some input, do something, and return an output.

```c
float get_hwr(person p) {
    float ratio;
    ratio = p.height / p.weight;
    return ratio;
}
```
Functions

The black box

All functions take some input, do something, and return an output.

```c
float get_hwr(person p){
    float ratio;
    ratio = p.height / p.weight;
    return ratio;
}
```

Function header summarizes this:

```c
return_type what_i_do (input_types)
```
Function contents

assignment

Notice: one equals sign.

variable = a_value;

math

+  -  *  /  %

Some cute 'n' conveninet forms (Java, C++, C, asst others):

a += b;  a = a + b;
a -= b;  a = a - b;
a *= b;  a = a * b;
a /= b;  a = a / b;
a %= b;  a = a % b;
a++;  a = a + 1;
a--;  a = a - 1;
Function contents

_conditional evaluation_

if (a == b){ //two equals signs
    do_stuff;
}
else
    dont;
Function contents

**conditional evaluation**

```java
if (a == b){ //two equals signs
    do_stuff;
} else
    dont;

Also:

var1 > var2
var1 <= var2
var1
function(x)

If it evaluates to a zero it’s false; else it’s true.

**while loops**

```
Function contents

**conditional evaluation**

```java
if (a == b) {// two equals signs
    do_stuff;
}
else
dont;

Also:

var1 > var2
var1 <= var2
var1
function(x)

If it evaluates to a zero it’s false; else it’s true.

**while loops**

i = 0;
while(i < array_limit){
    use_array_element(i);
    i++;
}
```
Function contents: loops

for loops

for(i=0; i< array_limit; i++){
    use_array_element(i);
}

Comments

Use them.

/* for long comments, start with slash-star,  
   end with star-slash. */

//For short comments, just start with two slashes

#Scripting languages use an octothorpe

%TeX uses a percent sign.
That’s all you get.

To make it interesting, we build and package larger structures which do a lot with little code.
The stack of frames

- The function running now is the current frame. There can be only one.
- If the function calls a new frame, then a new frame is created and runs.
- Picture a stack of frames. Only the top frame is active and visible.
- The bottom of the stack is the `main()` function or the top of the file called.
The stack of frames

- The function running now is the current frame. There can be only one.
- If the function calls a new frame, then a new frame is created and runs.
- Picture a stack of frames. Only the top frame is active and visible.
- The bottom of the stack is the `main()` function or the top of the file called.

An example, with 402 frames.

```c
int main (void) {
    person the_population[400];  //(this won’t actually work)
    the_population = produce_people("data_file");
    for (i=0; i<400; i++)
        print "the hwr of person ". i . " is ". get_hwr(the_population[i]);
    return 0;
}
```
Functions

call-by-value v call-by-reference

One of the key differences between languages.

- **Call-by-value**: In most languages, when a frame is built, a copy of the input variables are sent. C, C++, Matlab, R, Perl &c.

- **Call-by-reference**: Send in the variable itself, to be modified or destroyed inside the function. Always in Java; others use pointers. [Except R, which just can’t.]
Scope

The other key difference between languages.

*scope* of a variable: The frames which can see (a copy of) the variable. Options:

- **global**: every frame gets it.
- **local**: functions see only vars declared inside the function or explicitly passed via reference.
- **file-based**: variables are global only within the text file they’re declared in. Use multiple text files to divide scope.
- **object-based**: next slide.
Scope

**Objects**

An object is a structure with function elements (aka *methods*). Call functions as you would other elements: with a dot. `person.hwr()`.
Scope

Objects

An object is a structure with function elements (aka methods). Call functions as you would other elements: with a dot. `person.hwr()`.

This engenders new scope options:

- **public**: if `person` is in scope, then so are its public elements (via the dot).

- **private**: scope is limited to functions which are part of the object.
Scope

The importance of good scope

The rule: keep all variables’ scope as small as possible.

- Fewer moving parts in every frame \(\Rightarrow\) easier debugging.
- Allows overloading: let person have a years variable and a person.age() function and let dog have a dog.years and a dog.age() function too.
- Allows encapsulation.
Encapsulation

*Or, modular programming*

- File-based scope
  - Each file is a module entire unto itself. Public variables are put into an accompanying header file.
  - `#include file.h` to call the functions or use the structures declared therein.
Encapsulation

Or, modular programming

- File-based scope
  - Each file is a module entire unto itself. Public variables are put into an accompanying header file.
  - `#include file.h` to call the functions or use the structures declared therein.

- Object-based scope
  - The declaration of the object structure explicitly lists the public/private components.
  - Usually, each object is defined in a separate file anyway, which is included.
Encapsulation

*inheritance*

- Files may `#include` other files, which in turn `#include` others, &c.

- Objects may inherit from other objects, e.g., Players are a type of Cell-Occupant:
  ```java
  public class Player extends CellOccupant
  ```
Encapsulation

inheritance

- Files may `#include` other files, which in turn `#include` others, &c.

- Objects may inherit from other objects, e.g., Players are a type of Cell-Occupant:
  ```java
  public class Player extends CellOccupant
  ```

Assembling a program from parts

How to write a program:

- find the modules (files or objects) which embody the structures and functions you are interested in.

- Call the functions in your own program.
Encapsulation

*inheritance*

- Files may `#include` other files, which in turn `#include` others, &c.
- Objects may inherit from other objects, e.g., Players are a type of Cell-Occupant:
  
  ```java
  public class Player extends CellOccupant
  ```

Assembling a program from parts

How to write a program:

- find the modules (files or objects) which embody the structures and functions you are interested in.
- Call the functions in your own program.

So what’s the difference between a program and a package?
The program includes a `main()` function (or other code for immediate evaluation).
Compilation and/or execution

A two step process:

- Compilation: convert your text into machine-language instructions. Produces an illegible file.
  - C: an object file, file.o.
  - In Java: a class file, file.class.
  - Interpreted languages skip this step, and do it real-time.

- linking
  - Find all of the modules you called, and put them together into one file.
  - Either explicitly list them, or set a path to search.
Compilation example

#!/usr/bin/bash
AROOT=/home/bklemens/Ascape
CCROOT=src/edu/brook/currencycrisis

gcj -C -d $AROOT/lib/ --classpath=$AROOT/ascapecore.jar:"
$AROOT/lib/edu/brook/ascape/model/:
$AROOT/lib/:\$AROOT/collections.zip:
$AROOT/QTJava.zip:\$AROOT/jcchart362J.jar\n
$AROOT/\$CCROOT/CurrencyModel.java  $AROOT/\$CCROOT/Bank.java
$AROOT/\$CCROOT/Investor.java  $AROOT/\$CCROOT/ParameterReader.java
$AROOT/\$CCROOT/MatrixOperator.java  $AROOT/\$CCROOT/Bond.java
$AROOT/\$CCROOT/MarketMaker.java
**execution example**

Java links real-time, so you need to give it a class list when you run the program:

```plaintext
set AROOT=c:\cygwin\home\bklemens\Ascape
set JAVAEXE=c:\pfiles\java\bin\java
%JAVAEXE% -cp %AROOT%\lib\;%AROOT%\ascapecore.jar\;
%AROOT%\collections.zip;%AROOT%\jcchart362j.jar\;
%AROOT%\QTjava.zip edu.brook.ascape.model.Scape \
edu.brook.currencycrisis.CurrencyModel
```
Part II: Agent-based modelling
Complexity and emergence

The Mandelbrot set

- $x_0 = 0$
- $x_{n+1} = x_n^2 + z$
- If $x_n$ converges, $n \to \infty$, then $z \in$ Mandelbrot set.
Complexity and emergence

The Mandelbrot set

- \( x_0 = 0 \)
- \( x_{n+1} = x_n^2 + z \)
- If \( x_n \) converges, \( n \to \infty \), then \( z \in \text{Mandelbrot set} \).

The only way to determine whether \( z \in \text{set} \) is to do the darn calcualtions. Therefore, the set is:

- Deterministic
- Unpredictable
Agent-based modeling

- Specify simple rules for the micro-level behavior of the agents.
- Let them interact.
- Observe what the system converges to.
Agent-based modeling

- Specify simple rules for the micro-level behavior of the agents.
- Let them interact.
- Observe what the system converges to.

Again the setup is:

- Deterministic
- Unpredictable
Existential issues

Or, Why?

- Find parsimonious explanations for complex behaviors.
Existential issues

Or, Why?

- Find parsimonious explanations for complex behaviors.
- Replace models which make macro assumptions and get macro outputs with micro assumptions and macro outputs.
When to use an A.B.M. instead of an equation-based model

- When the game is iterated and period $t + 1$ depends heavily on period $t$ (like the Mandelbrot set).
When to use an A.B.M. instead of an equation-based model

- When the game is iterated and period $t + 1$ depends heavily on period $t$ (like the Mandelbrot set).
- When there are multiple equilibria, and the theory says nothing about which will prevail (e.g., anything with a tipping point)
When to use an A.B.M. instead of an equation-based model

- When the game is iterated and period \( t + 1 \) depends heavily on period \( t \) (like the Mandelbrot set).

- When there are multiple equilibria, and the theory says nothing about which will prevail (e.g., anything with a tipping point)

- When functional forms are expected to be nonlinear (e.g., anything with a tipping point)
When to use an A.B.M. instead of an equation-based model

- When the game is iterated and period \( t + 1 \) depends heavily on period \( t \) (like the Mandelbrot set).

- When there are multiple equilibria, and the theory says nothing about which will prevail (e.g., anything with a tipping point).

- When functional forms are expected to be nonlinear (e.g., anything with a tipping point).

- When you have no idea what the macro functional forms are.
When to use an A.B.M. instead of an equation-based model

- When the game is iterated and period $t + 1$ depends heavily on period $t$ (like the Mandelbrot set).
- When there are multiple equilibria, and the theory says nothing about which will prevail (e.g., anything with a tipping point)
- When functional forms are expected to be nonlinear (e.g., anything with a tipping point)
- When you have no idea what the macro functional forms are
- When selling to non-mathematicians
The Agents

- Many of them
- generally dumb.
  - limited processing ability
  - limited information
  - limited choices (e.g., location, network, buy/sell)
the game of life

- A 2-D grid
- if an empty space has 3 neighbors, then there's a birth
- if a filled space has <2 neighbors or >3 neighbors, there's a death.

We can do this with a space and agents on the space.
public class agent{

private: int neighbors;

public agent(int location){
    age =
    is_dead =
    last_update = 0;
    location = position;
}

void update(int t){
    if (t != last_update){
        last_update= t;
        neighbors = position.count_neighbors();
        if (neighbors > 3 || neighbors < 2)
            is_dead = 1;
    }
}
The location class

public class location{
    public: int is_alive, prior_state, last_update;

    private: location neighbor_list[8];
            int row, col, living_neighbors;
            agent occupant;

    public location (int row, int col){
        set_up_neighbor_list(row, col);
        is_alive = 0;
        prior_state = 0;
    }

Continued.
The location class

```java
public update(int t) {
    if (t != last_update)
        last_update = t;
    prior_state = is_alive;
    living_neighbors = count_neighbors();
    if (is_alive) {
        occupant.update(t);
        if (occupant.is_dead)
            is_alive = 0;
    } else {
        if (neighbors == 2)
            is_alive ++;
        occupant = new agent(this);
    }
}

public int am_i_alive(int t) {
    if (t == last_update)
        return prior_state;
    else
        return is_alive;
}
```
The program

The agents (and the space) do all the work ⇒ the main loop just asks the agents to keep updating.

```plaintext
space.initialize()
for (t=0; t<limit; t++){
    foreach(location)
        location.update()
    do_accounting();
}
```