The World— Due Fri Jan 17th

The World Abstraction

The WorldType supports the abstraction of a rectangular playing field with objects scattered at various locations. Locations are specified using 2-dimensional (x,y) coordinates. Each object in the world is at a location, and only one object can be at a location at a time. The world keeps track of all the objects and their locations for you. So the WorldType is useful for any program which needs a 2-dimensional space with a lot of things in it, like Battleship, a soccer simulator, and, by staggering coincidence, the Darwin program we're doing next week. The WorldType is most appropriate when the world is "sparse"— that is, most of the world is empty space which relatively few objects scattered about.

Initially the world contains no objects. You add objects to the world with the `AddObject` procedure. Each object is given an identification number when it is added. Subsequently that object can always be referenced by its number. For example the `LocationOfObject` function takes an object number and returns that object's current location. The object numbers are always allocated in the range 1..<number of objects in the world>. So if there are 10 objects in the world, then the objects will be identified by the numbers 1, 2, ...10. This object numbering scheme provides a fairly easy way to iterate through all the objects in the world with a for-loop.

Here is the interface for the WorldType wich defines how it behaves.

```
interface
 uses
  LocationADT; {Defines a simple 2-dimensional (x,y) coordinate type.}
               {This ADT is already written for your convenience.}

 const
  MaxThings = 1000;   {Maximum number of objects in the world.}
  MaxHeight = 10000;  {These are the maximum allowable dimensions of a world.}
  MaxWidth = 10000;   {The size of a particular world is given when it is initialized.}

 type
  FoundType = (Something, Nothing, OutOfBounds);
{Used to indicate the possible results of a search in the Look procedure below.}

{Stores a 2-dimensional (x,y) coordinate.}
  Location = record
    x, y: Integer;
   end;

  WorldType =
    <the type declaration of WorldType goes here, but it is not material to the abstraction>

 procedure InitWorld (var world: WorldType; height, width: Integer);
{Initialize a WorldType variable so that it is empty and has the given dimensions.}
{As is usual with initialization, any previous contents of the variable are destroyed.}

 function WorldWidth (world: WorldType): Integer;
{Returns the width of a world.  This will be whatever width was specified when the world}
{was initialized.}

 function WorldHeight (world: WorldType): Integer;
{Returns the height of a world.}
```

```
 procedure AddObject (var world: WorldType; where: Location; var objNum: Integer);
{Add a new object to the world at the given location.  The new object will be given a new}
{identification number which is returned in objNum.  AddObject will generate an error if you}
{try to add an object at a location which is out of bounds or where there is already something.}

 function NumObjects (world: WorldType): Integer;
{Returns the number of objects defined in the world.  The objects will have numbers}
{in the range 1..NumObjects.}

 function LocationOfObject (world: WorldType; objNum: Integer): Location;
{Returns the location of the given object in the world.  LocationOfObject generates}
{an error if there is no such object.}

 procedure Look (world: WorldType; where: Location; var found: FoundType; var objNum: Integer);
{Look returns what is at a location in the world.  There are three possible results of}
{this search which are enumerated in the 'found' parameter.  If there was nothing at location}
{then found will be Nothing.  If the location was beyond the bounds of the world, then found}
{will be OutOfBounds.  If there is an object at the location, then found is Something and}
{objNum is the number of the object at that location- in the other cases, objNum will}
{have a random value.}

 procedure MoveObject (var world: WorldType; objNum: Integer; newLocation: Location);
{Change the location of an object.  MoveObject generates an error if the object}
{does not exist or if there is already something at newLocation or the new location}
{is out of bounds.}
```

Your Mission

For this assignment, your mission is to implement the WorldADT. We will build on the WorldADT next week to build the Darwin simulation. Since the worlds may be quite large (10000 by 10000), a 2-dimensional array implementation is inappropriate. Instead you can use an array to keep track of the locations of all of the objects in the world. All of the files you need to get started are on the LAIR servers in the CS106B folder. Turn in a printout of your program to your section leader in class, Fri Jan 17th. TV people can just turn in this week's assignment along with Darwin next week since they are getting it a day later.

Darwin's World— Due Fri, Jan 24th

Introduction

Darwin's World is a simulation of a bunch of "creatures" running around in a simple square world. Every creature has a particular species. There might be 20 Flytraps, 10 Wallflowrs, and 10 Predators. All 40 creatures roam about the world abusing each other attempting to become the dominant species—sort of a post apocalyptic Karel's world gone mad. The program relies on everything you learned in 106A: decomposition, data structures, simple file processing, and using abstract data types.

Darwin's world is a two dimensional grid. Each square in the world can contain at most one creature. Each creature belongs to a "species". A species has two properties. First of all, each species has name which is a string such as "Flytrap" of "Creeping Death". Secondly, each species contains a short program which dictates how creatures in that species behave. A creature operates in the world by following the program of its species. You can think of each creature on the world as a little robot following a simple program. All of the creatures in the world take turns. When a creature gets its turn, it executes a few steps of its program. The creature programs constitute an extremely simple programming language.

Species Programming

Here's an example species program. There is one instruction per line. Creatures begin executing at line 1. There are two types of instructions: those which cause the creature to perform some action in the world, and those which possibly change which instruction the creature will execute next.

```
Line#     Instruction:
1:        ifwall 4    If there is a wall in front, go do line 4
2:        hop         hop forward
3:        go 1        go do line 1
4:        left        turn left
5:        go 1        go do line 1
```

In English, this is what a creature with this program would do: "First, I'll check to see if there is a wall in front of me. If there is, then I'll do whatever is on line 4. If not, I want to do whatever is next, in this case line 2..." In essence, what this creature does is: hop forward whenever the way is clear. When there is a wall in front, the creature turns to the left. As soon as there is no wall in front, the creature begins to hop forward again. Locations outside the Darwin grid appear as walls so creatures can't hop off the world.

Since there are many creatures on the world, they all take turns executing one instruction at a time. Let's say there was a wall in front of the creature -- then it goes on and reads line 4 "I should turn left." That's the end of the current turn for this creature. The next time this creature gets a turn, it starts executing at line 5. Each creature keeps track of the next line it wants to execute. (this is in fact exactly how computers work) So when it's their turn again, they know which instruction to execute. So, the next time this creature gets a turn, it reads line 5: "do the instruction on line 1." Then it reads line 1, and it's back to where it was in the beginning.

Here are the 10 instructions in the species language:

Hop means that the creature should move forward. Next turn the creature will be one space "forward". So, if a creature is at (2,2) and facing north, it ought to end up at (2,1) after doing a hop. You will not have to be concerned with row and column number detail since all of that is handled for you by the Location and Orientation ADT's below. A creature can only hop onto an empty space. In general, if a creature tries to do something stupid, like hop into a wall, then your program should just not perform the action. If a creature tries to hop onto an occupied square -- you should just leave the creature where it is. Essentially, the creature just loses a turn.

Infect means that the creature is attacking a creature directly in front. Infect changes the species of the attacked creature to be the species of the attacker. In a sense, the attacked creature is changed over to being on the attacker's team. The location and orientation of the infected creature are unchanged. The infected creature will begin executing its new program at line 1 on its next turn. Infect should only work when there is a creature of a different species in front of the current creature Again, if a creature tries to do something illegal like infect an empty square then your program shouldn't crash -- the creature just loses its turn.

Left and **Right** are the easiest things to do -- the creature just changes which direction it's facing. A creature can always turn left or right.

Ifenemy x means that if there is a creature of a different species in front of the current creature, then the current creature will do instruction at line x next. Otherwise, the current creature will go on to the next instruction. The other **if**xxxx instructions operate similarly. Every square in the world is in one of the following states: empty, wall, containing a creature.

Go x means that the creature will execute instruction x next.

Here's a summary of the instructions and what they mean:

INSTRUCTION:	MEANS:
hop	go forward one space
infect	make the creature in front of me be the same species as me
left	turn left 90 degrees
right	turn right 90 degrees
ifenemy x	if there is an enemy in front of me then follow instructions from line x on, otherwise execute the next instruction
ifwall x	same as above, if there is a wall in front of me ...
ifempty x	same as above, if there is an empty space in front of me...
ifsame x	same as above, if the creature in front of me is the same species as me...
ifrandom x	just to make things interesting, half of the time this will go on to the next line and half of the time it will go to line x
go x	continue reading and executing instructions at line x

A creature's turn ends after it tries to execute one of the following actions: hop, left, right, infect. So in one turn, one creature gets to do as many go's or if's in a row as it wants -- the turn ends after the creature tries to hop, infect, or turn.

Your Mission

The program has two stages. First, the user loads the world with the species they want. The information which defines a single species is stored in a file. Each time the user loads a species, 10 creatures of that species should be created randomly oriented at a random locations in the world. Once all the creatures are loaded, the simulation can begin. For each round of the simulation, your program should cycle through all of the creatures giving each one a turn. The module below provides graphics routines you can call to display the world at the end of each round. The program continues to run until the user hits the mouse button. There is a working version of the program on the server, so you can see how it works.

Data Structures

You will want to exploit the LocationADT, and you can use the WorldADT to keep track of all the creatures in the world. There is additional information for the species and for each creature that you will need data structures for. **You should avoid storing any information redundantly**. (This includes Locations.)

Species Files

The information for each species is stored in a file. Here's the actual file "Flytrap" from the "Sample Species" folder on the airport servers. The Flytrap sits and spins, waiting for something to come along.

```
Flytrap
ifenemy 4
left
go 1
infect
go 1
<blank line>
```

The first line of each species file is the name of the species which will be at most 31 characters long. The rest of the lines are the list of instructions that creatures of this species will follow. There will always be a blank line after the list of instructions. Some instructions occur on a line by themselves, and some instruction are always followed by an integer line number. Comments can appear off to the right as in the example on page 1.

You'll need to read in the name "Flytrap" and store it, then read in each line of instructions. Think Pascal has a non-standard capability to read and write enumerated types directly. Here's a tiny example which reads from standard input:

```
program Carpe;
type
     TimeType = (Post,Meridium);  {an enumerated type}
var
     time:TimeType;
begin
     read(input, time);     {here READ notices the type of its param}
                            {and does the right thing.}
     writeln(time);
end.
```

Read compares the text that was just typed to the values in the enumerated type (in this case "Post" and "Meridium"). Read sets the parameter to have the value corresponding to what the user types. So if the user types "Post", then read will set the variable `time` to its enumerated value `Post`. This feature is invaluable for reading in the instructions for each species. The read will crash if the user types anything other than "Post" or "Meridium".

OldFileName

The Think Pascal function `OldFileName` is convenient to let the user pick which file to open. `OldFileName` returns the empty string (') when the user selects cancel, so you can use something like the following to loop until the user hits cancel to indicate they are finished adding creatures.

```
  var fileName: Str255;  {Str255 is a string which can hold up to 255 chars}
      speciesFile: Text;
  begin
    repeat
      fileName:= OldFileName;
      if fileName<> '' then begin
        reset(speciesFile, FileName);
 {deal with reading that species into the world and creating 10 creatures}
      end;
    until FileName = '';
  end;
```

Getting Started

Your first challenge for this program is writing the code to read the species file and correctly set up your data structures. Once the world is set up, you need to write the code to give each creature its turn of execution. It's a good idea to develop the program in stages—it's very difficult to debug your creature execution code if your data-structures are wrong; also. When your program has everything except creature execution, you should be able to test it by running as if it were finished. When the simulation begins, the creatures should show up on the screen correctly. The creatures just won't move or do anything since they are not actually running their programs. At this time, you may also want to walk through your data structures with the debugger to verify that they are set up correctly.

When working on the creature execution, you should start with a simple creatures like the `Hop`, `Food`, and `Flytrap`. For debugging purposes, you may want to temporarily bypass the call to `OldFileName` to always use the same file name such as `'Hop'`. Just make sure the Hop file is in the same directory as your project. This simple trick can actually make your debugging go a lot quicker since `OldFileName` is not pestering you to click on a file every time you run. When everything is working, the most entertaining battles are between the Rovers and the Creeping Deaths with some Food thrown in. The Creeping Death might more accurately be called the "Paranoid Clumping Death".

All sorts of files to get started are in the class folder at the LAIR. The project file is already set up. You can just open and start adding your code to "darwin.p". Put your final version of "WorldADT.p" in the darwin folder. The project is set up to find the WorldADT in a file by that name.

Facts

- Each time a species is loaded, 10 more creatures of that species should be added to the world at random locations with random orientations. You do not need to deal with the case where the user loads the same species twice, although the version I'm handing out does deal with this correctly.

- Creatures begin executing at line 1 of their program. This holds for creatures at the very beginning and creatures which have just been infected.

- There will be at most 5 species in a single run of the program. A species program will be at most 200 lines long.

- Your program is not required to behave reasonably if a species program has syntax problems or bugs.

- As an optional feature, you may want to extend your program to
  a) keep track of the number of creatures in each species to help judge who "won" at the end.
  b) realize when the world is in a steady state, so the simulation can stop itself.

- A species wins a game if there are more of that species than any other . Just when the game is over is more subjective. For the tournament (described below) a game is considered over after 500 rounds with no infections or 5000 rounds total.

Hints

- The WorldADT keeps track of locations. You will need a data structure to keep track of other information for each creature— since the WorldADT always uses object numbers in the range 1..<the number of objects>, you can use the object numbers from the WorldADT directly as an indeces into an array to store other information for each creature. The "Bad Yogurt" program uses this trick for the "moving" array.

- The WorldADT also keeps track of the number of objects (creatures) in the world, so you don't need to keep track of that yourself.

- If you properly remove the redundancies from your data-structures, you will find that you are forced to pass a few arrays around where you wouldn't otherwise which is a little disturbing. We will see the solution to this problem in a few weeks, but it's ok in the mean time.

- Probably the most interesting problem in the program is getting code to "execute" a creature to be as short and clean as possible. Even the nicest possible solution ends up being somewhat long since it needs to contain a case statement which deals with all 10 opcodes. I found the `with` statement to be handy for this procedure.

- It's possible for the simulation to get stuck in a vicious cycle where two creatures are stuck in a patter of alternately infecting some third creature. It looks wrong, but it's right according to the rules.

- The function `Button` returns true if the mouse button is down.

- Your program will run much faster with the range-checking and debugging flags off— but it's safest to leave them on until you have gotten all the bugs.

The Darwin Module

The Darwin Module provides a few simple things to make your life easier. The opcode type is just an enumerated type which contains all of the different opcodes/instructions that creature programs contain. This module also contains the graphics routines. This module contains the consts which control the size of the world and the size that things are drawn. Here is the interface for the Darwin Module.

```
unit DarwinModule;
{This module groups a few simple routines which support}
{the Darwin program.  In particular the module contains a few basic}
{constants, some basic types, and all the graphics routines.   With better}
{planning, RandomNum would be in here too, but as it is, it ended up}
{in LocationADT.  Oops.}

interface
 uses
  LocationADT;
 const

  MaxName = 31;     {max number of chars in a species name}
  MaxX = 20;        {a common size of the world, feel free to change these.}
  MaxY = 20;
  DrawRadius = 14;  {controls how big each creature appears on screen}
{the above three will adjust the size of the drawing window appropriately}

 type
  SpeciesName = string[MaxName];

(*these are the different instructions/opcodes that the creatures can
understand*)
  OpcodeType = (left, right, hop, infect, go, ifwall, ifempty, ifenemy,
ifsame, ifrandom);

(*Given the creature's location and orientation, draw the creature*)
(*on the screen*)
 procedure DrawCreature (loc: Location; dir: Direction; name: SpeciesName);

(*Erases the window that the creatures are drawn in -- a handy way of*)
(*clearing the screen before drawing the next generation*)
 procedure EraseDrawingWindow;

(*Initialize graphics stuff -- opens a window for drawing, etc.  Call this*)
(*procedure once at the beginning of your program.*)
 procedure InitGraphics;

implementation
<who cares, so long as they work right...>
```

Deliverables

Turn in a printout of your program in class Friday, Jan 24th. Be sure your program includes your name, your section leader's name, and the date. Hopefully we will have a "decomposition and documentation" handout in your hands next week. In the mean time, suffice it to say that your decomposition and documentation will need to be excellent to receive top marks. Your section leader will pick up your program in class, so bringing it by my office later that day is not the same.

Creature Tournament

If you're interested, you can design your own creatures. Some of the creatures from last year are included in the assignment folder. You can create your own text files using Apple-Edit or by saving in text-only-with-line-breaks mode in Word. Tournament rules are the following: 25 by 25 world. 10 each of the contestant species and 20 food. The tournament will proceed round-robin with winners determined by record. Creatures will be due the Wed after the assignment is due. Please include your name as well as comments describing your strategy in your creature. One creature per person please— if you have two or more good stragey ideas, you may want to try a hybrid creature which switches around between several modes of operation. We may also have a mother-of-all-battles where we put all of the entries together in one big world and let them all slog it out. A couple good ideas to come out of previous creatures are: clumping with your own species or walls is good defense, going after the food early on is good, taking over corners is good defense. The Rover is a classic creature which does ok, but is a little weak on defense.

```
Rover
ifenemy 11
ifwall 6
ifsame 6
hop
go 1
ifrandom 9
left
go 1
right
go 1
infect
go 1

The rover constantly moves forward.  When it runs into a wall or another rover, it randomly
turns left or right.
```