# Task B
# Recursive Structures Project (Weeks 7 - 13)

SOFT1002 Software Development 2, 2006 Semester 2

September 13, 2006

## Contents

## 1 Task B Concept Inventory

Use the inventory on the following page to summarize what you have learnt in Task B, and to find gaps in your knowledge compared with what is expected. Score yourself against each item like this:

1. I've never heard of it
2. I've heard of it but know nothing more than that
3. I know this well enough to try to apply it
4. I know this and I can apply it
5. I know this well enough to explain it to a friend

| Topic | Score | Notes on what to do about this item |
|---|---|---|
| Context-free grammars | | |
| Derivations and parse trees | | |
| EBNF notation | | |
| Recursion | | |
| Recursive definitions | | |
| Understanding recursion by tracing | | |
| Understanding recursion without tracing | | |
| Example: The Towers of Hanoi | | |
| Example: Merge sorting | | |
| Example: Binary search | | |
| Writing your own recursive methods | | |
| Trees and the Composite design pattern | | |
| Recursive definitions for trees | | |
| Tree traversal | | |
| The Composite design pattern | | |
| Expression trees | | |
| Lexical analysis | | |
| Recursive descent parsing | | |
| Writing a pure parser | | |
| Decorating a parser | | |

# 2 Task B Problem Statements

For Task B, you form substantially new groups in Week 7, and design and implement a system that performs some useful task on recursively defined input.

You will certainly be using inheritance, but the emphasis here is on recursive definitions, recursive algorithms, the composite design pattern; you may also be using recursive descent parsing (and must do this for credit-level marks).

Be warned that attempting Task B without understanding and applying these techniques will *fail horribly*. Also note that the aim here is once again to produce a sound body of work *as a group*: it is not sufficient for everyone to write great code that doesn't integrate with the others' code in the group!

## 2.1 Overview

This is a simulation project. In this project you will write control programs that describe some behaviour of *agents*, in an artificial language called Sim$^2$, which we will describe. There will be two kinds of programs involved: those written in Sim$^2$, which describe what the agents' intentions *are*, and one written in Java, which will actually *implement* the actions. The Sim$^2$ program will not describe how the action takes place: that is the job of your Java program.

The simulation is of a Predator/Prey system, in which there are a number of `Predator` Objects and a number of `Prey` Objects. These Objects should *both* be derived from the superclass `Creature`.

The Predator/Prey system will be modelled in a two-dimensional array of cells arranged in a grid. Each Creature has a *position* in the grid given by its x and y coordinates, and there may be more than one Creature in a given cell in the grid.

The creatures, both Predator and Prey, each have a *status* variable, which describes what their current action is. That state will be one of "moving," "hunting," "resting," and "breeding" (perhaps more if you extend). The descriptions of each of the action for these states follow.

The state of the creature determines its action, and each action can take different amounts of time. In order to model different times for actions efficiently, we simply start a count-down when going into a state (such as resting) and at each turn decrement that count until it reaches zero. When the count reaches 0, that action is considered finished and the creature is free to change to a different state.

For instance you might write a control program for the Predator that determines from the evironment and its instance variables that it should *rest* for 10 hours; in that case you would set the state to "resting" and the countdown to 10. Every time the simulation program (your Java code) checks to see what this creature is doing for the next turn, it checks to see if the countdown has reached zero; if it has then it can choose another action by changing the "intention" variable, otherwise it must decrement the countdown and remain in the current state for the next turn.

The creatures can observe aspects of the environment (the 2-D array of cells) and can access their own instance variables, so the control program will base the decision on whether to rest or hunt etc. on that information. E.g., Predator creatures can see Prey if they are hunting for food or breeding, but not if they (the Prey) are resting.

In order to keep the control programs relatively simple the Creatures should *not* be able to change the state or instance variables of other Creatures. (Such an added complication is very difficult to implement with the tools you have from this unit!)

### 2.1.1 Predators and Prey

This project is a simulation using *agents*.

The simulation has both predators (e.g., lions) and prey (e.g., tourists). The Predators eat the Prey and the Prey graze. Both are Creatures, and both need Energy to survive, to hunt, and to breed – even to move. Essentially this simulation should mimic, albeit simplistically, the complex dynamic processes that we see in real biological systems.

Creatures need a minimum energy to be able to breed. Both kinds of creatures begin with an Energy of 10. The energy must always be positive: if an action reduce the energy to zero or below, the creature dies and is removed from the grid.

Thus you might construct a Creature class like this:

```
abstract class Creature {
   protected CreatureState currentState; // CreatureState is an enumerated type
   protected Direction currentDir // Direction could be an enumerated type too
   protected int energy;
   protected int xCoordinate;
   protected int yCoordinate;
   protected Prog behaviour;
   protected int dailyRest;
   public Creature(int x, int y, Prog control) {
      energy = 50;
      xCoordinate = x;
      yCoordinate = y;
      currentState = CreatureState.Resting;
      behaviour = control;
      dailyRest = 0;
   }
}
```

. . . and you would construct the constructors of Predator and Prey calling `super(...)`, in addition to other instructions.

The simulation will run over many time steps, which should be equivalent to hours.

Your program should do something like the following:

```
Main
   Initialise the environment (grid + Predators + Prey)
   Loop through time steps:
   For each time step
      For each Creature
         If it is continuing a previous action then
            Do so;
         else
            Determine the next action (from control prog.)
            Execute the action.
```

At the minimum, each Creature must have the following instance variables:

- *energy* summarising the amount of resources available to the creature;
- *x*, *y* the x- and y- coordinates;
- *state* the behaviour that the Creature is attempting to perform;
- *countdown* the amount of time remaining in the current state before the creature can change.

You could of course call any or all of these variables different things, but the names should be sensible.

## 2.2 Important notes

This section contains some notes of clarification, in the hope that we can keep you on target for a good outcome.

#### 2.2.1 GUIs

GUIs are *not* part of the core requirements for a passing grade in this task. For credit or better grades you may create a GUI, but there are other components that also could give you credit level marks so please do not become obsessed with the GUI! You may get full marks for a good *interface*, but not all interfaces are GUIs.

#### 2.2.2 Parser

*If* you write a parser, you should write it *last*.

Although reading input may seem like a good thing to do first, the parser is the hardest part and it is better to get everything else under control first. The seminar topics are arranged this way too.

You can test things without a parser by constructing small Composite objects through the BlueJ (or other IDE) workbench.

Note that a parser is required for *credit-level* marks or better; it is *not* a Core Requirement.

#### 2.2.3 Interactions

Some actions are dependent on the states of other creatures in the same cell. For instance if a Predator enters a cell while hunting, it will only be able to find Prey creatures that are themselves visible (not resting). A very poor scalability solution would be to test each Prey creature to see if it were in that cell and then see if it were visible (hunting or breeding); you should think of a better way!

## 3 Sim$^2$

This section is on the artificial modelling language you will use to describe behaviours.

### 3.1 The Grammar

The EBNF based grammar used here is read as:

| Statement | Meaning |
|---|---|
| `A ::= B C` | A is a B followed by a C |
| `A ::= [ B ]` | A is an optional B |
| `A ::= B \| C` | A is a choice of B or C |
| `A ::= { B }` | A is an arbitrary number of Bs |

Sim$^2$ is a very simple simulation language that you will use to describe behaviour of individuals in a dynamic environment.

Sim$^2$ has the following grammar:

```
<prog> ::= "sim" <statement> "end"
<statement> :: ( "block" { <statement> } "end")
      | ( <ifthen> | <assign> | <agentaction> )
<ifthen> ::= "if" <cond> "then" <statement> [ "else" <statement> ] "end"
<ident> ::= <var> | <int> | "rand"
<assign> :: <var> "=" <expr>
<expr>::= <ident> | <arith> | <cond>
<arith> ::= "add" <ident> <ident> | "subtract" <ident> <ident>
<cond> ::= "(" <ident> ( "<" | ">" | "==") <ident> ")"
<agentaction> ::= <move> | <hunt> | <rest> | <breed>
<move> ::= "move" <dir>
<hunt> ::= "hunt" <dir>
<rest> ::= "rest" <int>
```

```
<breed> ::= "breed" <dir>
<dir> ::= "N" | "NE" | "E" | "SE" | "S" | "SW" | "W" | "NW"
```

The *terminals* are: `sim`, `end`, `block`, `if`, `then`, `else`, `rand`, $=$, $<$, $>$, $==$, (, ), `move`, `hunt`, `rest`, `breed`, `N`, `NE`, `E`, `SE`, `S`, `SW`, `W`, `NW`, integers and variable names.

The *non-terminals* are: `<prog>`, `<statement>`, `<ifthen>`, `<ident>`, `<assign>`, `<cond>`, `<agentaction>`, `<move>`, `<hunt>`, `<rest>`, `<breed>`, and `<dir>`. In general it is the LHS that maps to classes.

### 3.1.1 `<prog>`

This just represents the program: all Sim$^2$ programs have this general format.

### 3.1.2 `<statement>`

Valid statements are `<ifthen>`, `<assign>`, `<agentaction>`, and blocks of statements.

### 3.1.3 `<ifthen>`

The first control flow statement of Sim$^2$. Note that the `"else" <statement>` is optional!

### 3.1.4 `<ident>`

`<ident>` is an identifier, and may be a variable, an integer value, or the keyword “rand.”

### 3.1.5 `<cond>`

`<cond>` returns a “true” or “false” value depending on whether the test within the parentheses is true or false. The grammar does not specify how “true” is coded but it is an integer value; traditionally you would have “true” for 1 and “false” for 0. The allowed operators in a ¡cond¿ are < (less than), > (greater than), and == (equals).

### 3.1.6 `"rand"`

The only built-in mathematical function in Sim$^2$ is `rand`, which returns a pseudo-random integer, uniformaly distributed in the range [0,999].

### 3.1.7 `<agentaction>`

The agent actions are much more complex and are defined in the following sections.

### 3.1.8 `<assign>`

This is the simple assignment statement, setting the value of `<var>` to that of the `<expr>`. Note that you can define other variable names by using them in an `<assign>`, thus

```
count = 3
```

creates an integer variable count and assigns its value to be 3. Newly declared variables should be initialized to zero.

**Note** that an assign statement where the left hand side is a predefined variable like energy has no effect. It is up to the simulation program, not the control program, to change such variables.

## 4 Actions Summary

The following sections describe the way the simulation system should act, for each of the possible actions a creature might perform, in more detail.

## 4.1 Creatures

Some actions are common to both kinds of creatures, both Predator and Prey.

### 4.1.1 `<move>`

Any creature moving a single space incurs an energy cost of 1.

The `<move>` action just moves the Creature one cell in any of the 8 points of the compass, N, NE, E, etc. **Note** that more than one creature can exist in the same cell at any time.

### 4.1.2 `<rest>`

While resting, both kinds of Creature are invisible to any others, even of the same type. That also means that resting creatures cannot be chosen as mates by a breeder.

Creatures must each rest at least 8 hours in every day, but may rest longer. At the end of each day the Creature's energy is adjusted as follows:

- If the Creature spent the entire day resting, it loses 1 energy.
- If the Creature spent less than 8 hours resting, it loses 5 Energy.
- Otherwise, there is no effect on Energy due to resting.

The Creature class must therefore have a mechanism for keeping track of how long in a given day it has rested (perhaps `public int dailyRest`).

In your Sim$^2$ code you would write something like `rest 10` which means "for the next 10 turns, the state should be resting." The simulation program should check the state of the creature at each turn.

### 4.1.3 `<breed>`

Both Creature types have the same rules for breeding. This takes lots of energy (25) and another (visible) creature of the same type (but we are ignoring the creatures' sex here: it's a simple simulation after all!). To breed, a creature first moves to a new cell and then looks for a possible mate; if there is no visible creature of the same type and with sufficient energy in the cell, then don't do anything more; otherwise breed!

The creature instigating the breeding, the 'mum', sets its (her) state to be something like "breeding" for the next 6 turns, after which two new offspring are created in the same cell, and the Energy value of 'mum' is reduced by 25. (Note that while breeding 'mum' is *invisible*.) Thus the simulation program should check on the state of the Creatures and, if they are breeding, reduce the remaining time by 1 each time step. When the count gets to zero, the new offspring are produced and the mum is no longer "breeding."

As a possible extension you may want to consider more complex and realistic rules for breeding, e.g., the 'dad' must also choose breeding asits next action, and it too loses energy.

## 4.2 Predators: `<hunt>`

The Sim$^2$ code might be written `hunt NE`, which would set the action to be performed in this time step.

In the Simulation program you should implement the `<hunt>` action as follows: Move one space in the direction given (North 'N', North-East 'NE', etc.) and see if there are Prey there; if there are any *visible* (see later), then chase, to try to catch one! Just MOVING to a new cell costs 1 Energy, and actually CHASING a prey creature costs Energy of 10, whether the hunt is successful or not. Note that Prey cannot be caught without a chase.

As there are Prey creatures *visible* in the square, then the chasing predator has a good chance of catching one. It can catch at most one at a time. If it catches a prey creature then it stays to eat it for THREE (3) hours and gets 20 Energy. You would set the state to "eating" or some equivalent and set a variable to show that there were 3 turns remaining until the eating is done.

Here a "good chance" is worked out as a percentage equal to the current energy, up to a maximum of 75%. Thus a creature with low Energy is much less likely to be able to catch more to eat!

For example, a Predator with 61 Energy might `<hunt> NE`; on moving to that cell (and losing 1 Energy to go to 60) it sees there is some Prey, and it tries to catch one. If it succeeds (with probability 0.60) it will dine on Wildebeest for the next three hours and go up to 80 Energy.

### 4.3 Prey: `<hunt>`

This means "hunt for food" and costs 1 in Energy. As with the Predator `<hunt>`, the creature moves one space in the direction given. Because this is considered to be a rich environment, they gain 2 Energy from the food they find there. They cannot eat from the same cell twice in succession: they must move in order to eat. While hunting for food in this way, Prey are visible to the Predators.

## 5 The meaning of a Sim$^2$ Program

In some timesteps the action taken by a creature is already determined by its previous activity; for example, after `rest 10`, the next ten turns involve the rest action. If no action is predetermined, then the control program is executed: each statement in turn is processed until an action is chosen. Once an action is chosen, subsequent statents are ignored.

### 5.1 Example

An example of a (very simple) Predator behaviour might be

```
sim
   if ( Energy > 0 ) then
      if ( Energy < 100 ) then
         if (rand < 500) then
            hunt E
         else
            hunt N
         end
      else
         breed S
      end
   end
end
```

This program would have the following operation:

**1** if the Energy level is positive

- **1.1** if Energy is less than 100
  - **1.1.1** if (a random number is less than 500) then
    - **1.1.1.1** hunt to the East
  - **1.1.2** else
    - **1.1.2.1** hunt to the North
- **1.2** else
  - **1.2.1** go looking South for a Mate

**2** end

This would result in the Predator choosing one of several actions by setting the current state to something like "hunting" and the current direction to North.

The simulation program would then work out what were the consequences of this decision, such as whether there was anything in the area to prey on, and if so, whether that Predator creature got some lunch.

### 5.2 Notes

You should be aware of the *boundary conditions* in this simulation. The grid is finite!

Keep an eye out for interesting dynamics! If nothing interesting is happening, perhaps you need to change the behaviour.

## 6 Assessment

This Task can gain a pass mark with just the core requirements, and no fancy stuff. If you want to get Credit or higher grades then you should think about interesting ways in which you can extend the project. We have supplied some possible extensions below, which are by no means exhaustive. If you have other ideas discuss them with your tutor.

### 6.1 Core Requirements

For *pass-level* marks you *must*

- create a program that is capable of simulating a collection of Predators and Prey in the 100x100 grid;
- properly initialise the creatures (but the initial positions may be anywhere you like);
- express in the Sim$^2$ language the behaviour of both Predator and Prey creatures;
- run the simulation program through arbitrary time steps, keeping track of the numbers of predators and prey.

### 6.2 Extensions

For *credit-level* marks you must satisfy all the core requirements and extend your program.

The extension *must* include writing a decorated parser to read in arbitrary programs written in correct Sim$^2$ code, and turn that Sim$^2$ code into a executable program object as part of a Creature's behaviour.

The extension *may* incorporate such things as follow (and possibly others):

These are starred with how challenging *I* think they might be: you may disagree! The number of stars doesn't mean you will necessarily get any more marks.

If you want *any* marks for extensions the first extension must be parsing an input file in Sim$^2$ as mentioned above.

$\star$ Allow different behaviour among individuals of the same species. *Let them compete!*

$\star$ Include the time of day in your simulation: at night time conditions should be different from those during the day or at dawn and dusk. *Make it realistic!*

$\star$ Write a GUI and simulate the movements of Creatures in your grid. *Show it all graphically!*

$\star\star$ Extend the Sim$^2$ language to permit more complex `statements` and/or `agentactions`. *Let them plot and plan!*

$\star\star$ Introduce a memory to the creatures: where was the good supply of prey items (or conversely where were there no predators) last week? *Let them learn!*

$\star\star\star$ Introduction of inheritance with variation: the foundation of evolutionary theory. When creatures breed, merge with variation the behaviours of the parents. *Evolve new creatures!*

There are many more ways you might extend this Task: if you have an idea that you think you would like to use, but are unsure of whether it would count, discuss it with your tutor or one of the lecturers: we will be able to advise you.

## 7 Marking Scheme

**This is an outline – more soon**

### 7.1 PROBATION

If you are on probation the most you can get for the technical parts (recursion, simulation framework, and basic actions) is ZERO (0).

### 7.2 First Stage

At this point you won't have been taught about parsing, so you will not be able to do that extension: we will therefore not be considering any extensions at this point. Focus on getting the core requirements done for the first stage!

Marks will be awarded according to the following guidelines:

**Composite Design:** 4

0: No evidence of a reasonable attempt at composite design for the simulation program.

2: Must have an appropriate superclass to represent the control program, subclasses for the various atomic constructs (such as move and hunt), and other subclasses for the composed structures (such as conditionals or blocks). Each subclass should have the appropriate instance variables and constructor.

3: As above, plus: flawed attempt at correct evaluation of the control program to work out the next action, or at correctly calling the appropriate method for that action.

4: Correct access throughout of information about creatures and the environment by the simulation program. The program should correctly evaluate the control program to work out what the next action is, and based on that action call the appropriate method in the Predator or Prey class.

**Recursion on the composite:** 2

0: No recursion in evaluating the intention.

1: Flawed but reasonable attempt at recursion in the evaluation of the next intention, e.g., some composites not accounted for, or some base cases wrong, not ending a block when an intention is assigned, and so on.

2: This is for correctly working out what the intention is, when the control program is non-trivial: that is, a composite such as a conditional or a block.

**Correct implementation of base actions (e.g. hunt):** 2

**Simulation Framework:** 2

**Control programs in Sim$^2$:** 2

**Presentation:** 1

**Total: 15**

### 7.3 Second Stage

**Individual:** 0-4 for Parser; 0-6 for contribution assessed by tutor

**Group:** Inspection 2; Demo 2; Parser and other extensions 0, 2-6.