Créatúr Tutorial

Amy de Buitléir
Email: amy@nualeargais.ie

April 24, 2015

Contents

Chapter 1

Overview of Créatúr

Créatúr[1] is a software framework for automating experiments with artificial life (ALife). It provides a daemon which ensures that each agent gets its turn to use the CPU. You can use other applications on the computer at the same time without fear of interfering with experiments; they will run normally (although perhaps more slowly). Créatúr also provides a library of modules to help you implement your own ALife species. Even if you aren't using the Créatúr framework, you may find some of these modules useful.

[1] *Créatúr* (pronounced kray-toor) is an irish word meaning animal, creature, or unfortunate person.

Chapter 2

Getting started

2.1 Installing Créatúr and the examples

The instructions below should work on Linux or OSX. (Créatúr uses the `unix` package in order to implement the daemon.)

1. Make sure you've installed the Haskell platform. You can get it from `http://hackage.haskell.org/platform/`.
2. At the command prompt, type `cabal install creatur`. This will install the Créatúr framework.
3. Download `creatur-examples-master.zip` from `https://github.com/mhwombat/creatur-examples/archive/master.zip`. This contains the examples used in this tutorial.
4. Unzip the file you just downloaded (`creatur-examples-master.zip`). This will create a directory called `creatur-examples-master`.
5. At the command prompt, go to the directory you just created. (`cd creatur-examples-master`).
6. Build and install the examples by typing `cabal install`.

2.2 Learning Haskell

If you've never used Haskell before, I recommend the following books. The complete content of both books is freely available online, so try both and see which one best suits your learning style. As suggested by the title, *Real World Haskell* teaches Haskell using a series of practical examples. *Learn You a Haskell* is presented with a great deal of humour, and may give you a deeper understanding of functional programming concepts.

- Learn You a Haskell for Great Good!: A Beginner's Guide
  Miran Lipovaca
  ISBN-13: 978-1593272838
  `http://learnyouahaskell.com/`
- Real World Haskell
  Bryan O'Sullivan, Don Stewart, and John Goerzen
  ISBN-13: 978-0596514983
  `http://book.realworldhaskell.org/`

The Haskell community is very friendly and welcoming to newbies. Two good places to ask questions are the beginners@haskell.org mailing list (see `http://www.haskell.org/haskellwiki/Mailing_lists` for more information) and Stack Overflow `http://stackoverflow.com/questions/tagged/haskell`.

The API documentation for Créatúr is available at `http://hackage.haskell.org/package/creatur`.

2.3 IMPORTANT: GHC version dependencies

You may notice some differences between the snippets of source code presented in this tutorial and the actual source code for the examples. GHC version 7.10 introduced some important changes. For Créatúr users, this means that it is no longer necessary to import `Control.Applicative`. In order to make these examples work with both older and newer versions of GHC, I made some (temporary) changes to the examples. These changes are explained below.

First, I temporarily added a CPP language pragma to allow conditional compilation.

```
{-# LANGUAGE CPP #-}
```

You can remove that line if you're only working with GHC 7.10 or greater, but it will do no harm to leave it in. Finally, I added some code which imports `Control.Applicative` or not, depending on the GHC version used.

```
#if MIN_VERSION_base(4,8,0)
-- Starting with GHC 7.10 (base 4.8), we don't need to import
-- Control.Applicative
#else
import Control.Applicative
#endif
```

You don't need those lines if you're only working with GHC 7.10 or greater, but it will do no harm to leave them in.

Chapter 3

The main daemon loop

The daemon clock is a simple counter used to schedule events. At each tick of the clock, the daemon:

1. Reads the current list of agents, which are stored as separate files in the current directory.
2. Queues the agents in random order.
3. Processes the queue, giving each agent an opportunity to use the CPU by invoking a user-supplied function.
4. Increments the daemon clock.

A different random order is used at each clock tick so that no agent has an unfair advantage. (If agents were always processed in the same order, agents near the end of the list might, for example, find that the best food has already been eaten or that the desirable mating partners are already taken.)

Chapter 4

Overview of the process

The steps for using Créatúr to create and run an ALife experiment are listed below.

1. Create one or more ALife species
2. Generate an initial population
3. Configure a daemon
4. Build and run the universe

Chapter 5

A very simple species

In this part of the tutorial, we create and run a universe with a very simple species. This will introduce you to the basics of the Créatúr framework.

5.1 Create a species

We'll only use one species in this example, and it will be a very simple species called `Rock`. Rocks don't reproduce, so we don't need to worry about genetics.

A complete listing of the source code discussed here is provided on page 7. We'll discuss the main points here. The `DeriveGeneric` pragma activates support for generics (available beginning with GHC 7.2 or the Haskell Platform 2012.4.0.0), which allows some code to be generated automatically for us.

```
{-# LANGUAGE DeriveGeneric #-}
```

`Rocks` have a unique ID, and a counter. We will use the counter to demonstrate that the Créatúr framework persists ("remembers") the state of the rock from one run to the next.

```
data Rock = Rock String Int deriving (Show, Generic)
```

All agents used in the Créatúr framework must be an instance of `Serialize`, which ensures that they can be written to and read from a database or file system. Notice that we do not need to write `put` and `get` functions. Deriving `Generic` instructs Haskell to generate them for us.

```
instance Serialize Rock
```

All species used in the Créatúr framework must be an instance of `ALife.Creatur.Agent`, which requires us to implement two functions: `agentId` and `isAlive`. The function `agentId` returns a unique identifier for this agent. The function `isAlive` indicates whether the agent is currently alive (if it is not alive, it would automatically be archived). Since rocks never die, this function can simply return `True`.

```
instance Agent Rock where
  agentId (Rock name _) = name
  isAlive _ = True
```

Making `Rock` an instance of `ALife.Creatur.Database.Record` allows us to store agents and retrieve them from the file system. A record needs a unique identifier; we can use `agentId` for this purpose.

```
instance Record Rock where key = agentId
```

Finally, we need to write the function that will be invoked by the daemon when it is the agent's turn to use the CPU. This function is passed as a configuration parameter to the daemon, as will be seen in Section 5.3. This example is quite simple; it merely writes a message to the log file and returns a new version of the rock, with an updated counter. However, this is where your agents get a chance to eat, mate, and do whatever else they need to do. We'll see a more typical implementation in Section 7.1.

```
run :: Rock -> StateT (SimpleUniverse Rock) IO Rock
run (Rock name k) = do
  writeToLog $ name ++ "'s turn. counter=" ++ show k
  return $ Rock name (k+1)
```

The type signature of `run` is explained in Section 5.3. The complete code listing is below.

```haskell
{-# LANGUAGE DeriveGeneric #-}

module Tutorial.Chapter5.Rock (Rock(..), run) where

import ALife.Creatur (Agent, agentId, isAlive)
import ALife.Creatur.Database (Record, key)
import ALife.Creatur.Universe (SimpleUniverse, writeToLog)
import Control.Monad.State ( StateT )
import Data.Serialize (Serialize)
import GHC.Generics (Generic)

data Rock = Rock String Int deriving (Show, Generic)

instance Serialize Rock

instance Agent Rock where
  agentId (Rock name _) = name
  isAlive _ = True

instance Record Rock where key = agentId

run :: Rock -> StateT (SimpleUniverse Rock) IO Rock
run (Rock name k) = do
  writeToLog $ name ++ "'s turn. counter=" ++ show k
  return $ Rock name (k+1)
```

5.2 Generate an initial population

We will create our first population in a directory called `Chapter5`. In `main`, we initialise the `chapter5` directory, create two rocks, and write them to the directory. The parameters to the `mkSimpleUniverse` command specify the name of the universe, and the path to the directory that contains it. When a log file reaches the maximum size, it is closed and a new log file is started.

```haskell
import Tutorial.Chapter5.Rock (Rock(..))
import ALife.Creatur.Universe (store, mkSimpleUniverse)
import Control.Monad.State.Lazy (evalStateT)

main :: IO ()
main = do
  let u = mkSimpleUniverse "Chapter5" "chapter5"

  let a = Rock "Rocky" 42
  evalStateT (store a) u

  let b = Rock "Roxie" 99
  evalStateT (store b) u

  return ()
```

The procedure for running this program will be described in Section 5.4.

5.3 Configure a daemon

The module `ALife.Creatur.Universe.Task` provides some tasks that you can use with the daemon. These tasks handle reading and writing agents, which reduces the amount of code you need to write. (It's also easy to write your own tasks, using those in `ALife.Creatur.Universe.Task` as a guide.) The simplest task has the type signature:

```haskell
runNoninteractingAgents
  :: (Universe u, Serialize (Agent u))
    => AgentProgram u -> SummaryProgram u -> StateT u IO ()
```

That signature looks complex, but essentially it means that if you supply an `AgentProgram`, it will run it for you. So what type signature must your `AgentProgram` have? Here's the definition.

```
type AgentProgram u = Agent u -> StateT u IO (Agent u)
```

This is the signature for an agent that doesn't interact with other agents. The input parameter is the agent whose turn it is to use the CPU. The program must return the agent (which may have been modified). The universe will then automatically be updated with these changes. We will use the `run` function in `Tutorial.Chapter5.Rock` (discussed in Section 5.1) as our `AgentProgram`. Its type signature is shown below.

```
run :: Rock -> StateT (SimpleUniverse a) IO Rock
```

`SimpleUniverse a` provides the logging, clock, and agent naming functionality we need. So the type signature of `run` is consistent with `AgentProgram`. The program below configures and launches the daemon.

```
module Main where

import Tutorial.Chapter5.Rock (run)
import ALife.Creatur.Daemon (CreaturDaemon(..), Job(..),
  simpleDaemon, launch)
import ALife.Creatur.Universe (mkSimpleUniverse)
import ALife.Creatur.Task (simpleJob, runNoninteractingAgents,
  doNothing)
import System.Directory (canonicalizePath)

main :: IO ()
main = do
  dir <- canonicalizePath "chapter5" -- Required for daemon
  let u = mkSimpleUniverse "Chapter5" dir
  let j = simpleJob
        { task=runNoninteractingAgents run doNothing doNothing,
          sleepTime=0 }
  launch $ CreaturDaemon (simpleDaemon j u) j
```

5.4 Build and run the example

Here is the listing of `creatur-examples.cabal`.

```
Name:                creatur-examples
Version:             5.9.5
-- keep in sync with créatúr, update tutorial and dependencies below
Stability:           experimental
Description:         Examples demonstrating the use of the \"Créatúr\"
                     framework for artificial life experiments.
Synopsis:            Examples demonstrating the use of \"Créatúr\".
License:             BSD3
License-file:        LICENSE
Homepage:            https://github.com/mhwombat/creatur-example
Bug-reports:         https://github.com/mhwombat/creatur-examples/issues
Copyright:           (c) Amy de Buitléir 2010-2015
Author:              Amy de Buitléir
Maintainer:          amy@nualeargais.ie
Build-Type:          Simple
Cabal-Version:       >=1.8

source-repository head
  type:     git
  location: https://github.com/mhwombat/creatur-examples.git

source-repository this
  type:     git
  location: https://github.com/mhwombat/creatur-examples.git
```

```
  tag:      5.9.5

Executable chapter5-daemon
  Main-Is:          Tutorial/Chapter5/Daemon.hs
  GHC-Options:      -Wall
  Hs-source-dirs:   src
  Build-Depends:    array ==0.5.*,
                    base ==4.*,
                    cereal ==0.4.*,
                    creatur ==5.9.5,
                    directory ==1.2.*,
                    ghc-prim ==0.3.* || ==0.4.*,
                    hdaemonize ==0.5.*,
                    MonadRandom ==0.3.*,
                    mtl ==2.2.*,
                    old-locale ==1.0.*,
                    split ==0.2.*,
                    time ==1.4.* || ==1.5.*,
                    transformers ==0.4.*,
                    unix ==2.7.*,
                    zlib ==0.5.*

Executable chapter5-init
  Main-Is:          Tutorial/Chapter5/GeneratePopulation.hs
  GHC-Options:      -Wall
  Hs-source-dirs:   src
  Build-Depends:    base ==4.*,
                    cereal ==0.4.*,
                    creatur ==5.9.5,
                    directory ==1.2.*,
                    ghc-prim ==0.3.* || ==0.4.*,
                    MonadRandom ==0.3.*,
                    mtl ==2.2.*,
                    old-locale ==1.0.*,
                    time ==1.4.* || ==1.5.*,
                    transformers ==0.4.*,
                    zlib ==0.5.*

Executable chapter7-daemon
  Main-Is:          Tutorial/Chapter7/Daemon.hs
  GHC-Options:      -Wall
  Hs-source-dirs:   src
  Build-Depends:    array ==0.5.*,
                    base ==4.*,
                    cereal ==0.4.*,
                    creatur ==5.9.5,
                    directory ==1.2.*,
                    ghc-prim ==0.3.* || ==0.4.*,
                    hdaemonize ==0.5.*,
                    MonadRandom ==0.3.*,
                    mtl ==2.2.*,
                    old-locale ==1.0.*,
                    split ==0.2.*,
                    time ==1.4.* || ==1.5.*,
                    transformers ==0.4.*,
                    unix ==2.7.*,
                    zlib ==0.5.*

Executable chapter7-init
  Main-Is:          Tutorial/Chapter7/GeneratePopulation.hs
  GHC-Options:      -Wall
  Hs-source-dirs:   src
  Build-Depends:    base ==4.*,
```

```
                    cereal ==0.4.*,
                    creatur ==5.9.5,
                    directory ==1.2.*,
                    ghc-prim ==0.3.* || ==0.4.*,
                    MonadRandom ==0.3.*,
                    mtl ==2.2.*,
                    old-locale ==1.0.*,
                    time ==1.4.* || ==1.5.*,
                    transformers ==0.4.*,
                    zlib ==0.5.*

Executable chapter8-daemon
  Main-Is:          Tutorial/Chapter8/Daemon.hs
  GHC-Options:      -Wall
  Hs-source-dirs:   src
  Build-Depends:    array ==0.5.*,
                    base ==4.*,
                    cereal ==0.4.*,
                    creatur ==5.9.5,
                    directory ==1.2.*,
                    ghc-prim ==0.3.* || ==0.4.*,
                    hdaemonize ==0.5.*,
                    MonadRandom ==0.3.*,
                    mtl ==2.2.*,
                    old-locale ==1.0.*,
                    split ==0.2.*,
                    time ==1.4.* || ==1.5.*,
                    transformers ==0.4.*,
                    unix ==2.7.*,
                    zlib ==0.5.*

Executable chapter8-init
  Main-Is:          Tutorial/Chapter8/GeneratePopulation.hs
  GHC-Options:      -Wall
  Hs-source-dirs:   src
  Build-Depends:    base ==4.*,
                    cereal ==0.4.*,
                    creatur ==5.9.5,
                    directory ==1.2.*,
                    ghc-prim ==0.3.* || ==0.4.*,
                    MonadRandom ==0.3.*,
                    mtl ==2.2.*,
                    old-locale ==1.0.*,
                    time ==1.4.* || ==1.5.*,
                    transformers ==0.4.*,
                    zlib ==0.5.*

Executable chapter9-daemon
  Main-Is:          Tutorial/Chapter9/Daemon.hs
  GHC-Options:      -Wall
  Hs-source-dirs:   src
  Build-Depends:    array ==0.5.*,
                    base ==4.*,
                    cereal ==0.4.*,
                    creatur ==5.9.5,
                    directory ==1.2.*,
                    ghc-prim ==0.3.* || ==0.4.*,
                    hdaemonize ==0.5.*,
                    MonadRandom ==0.3.*,
                    mtl ==2.2.*,
                    old-locale ==1.0.*,
                    split ==0.2.*,
                    time ==1.4.* || ==1.5.*,
```

```
                     transformers ==0.4.*,
                     unix ==2.7.*,
                     zlib ==0.5.*

Executable chapter9-init
  Main-Is:           Tutorial/Chapter9/GeneratePopulation.hs
  GHC-Options:       -Wall
  Hs-source-dirs:    src
  Build-Depends:     base ==4.*,
                     cereal ==0.4.*,
                     creatur ==5.9.5,
                     directory ==1.2.*,
                     ghc-prim ==0.3.* || ==0.4.*,
                     MonadRandom ==0.3.*,
                     mtl ==2.2.*,
                     old-locale ==1.0.*,
                     random ==1.1.*,
                     time ==1.4.* || ==1.5.*,
                     transformers ==0.4.*,
                     zlib ==0.5.*

Executable chapter10-daemon
  Main-Is:           Tutorial/Chapter10/Daemon.hs
  GHC-Options:       -Wall
  Hs-source-dirs:    src
  Build-Depends:     array ==0.5.*,
                     base ==4.*,
                     cereal ==0.4.*,
                     creatur ==5.9.5,
                     directory ==1.2.*,
                     ghc-prim ==0.3.* || ==0.4.*,
                     hdaemonize ==0.5.*,
                     MonadRandom ==0.3.*,
                     mtl ==2.2.*,
                     old-locale ==1.0.*,
                     random ==1.1.*,
                     split ==0.2.*,
                     time ==1.4.* || ==1.5.*,
                     transformers ==0.4.*,
                     unix ==2.7.*,
                     zlib ==0.5.*

Executable chapter10-init
  Main-Is:           Tutorial/Chapter10/GeneratePopulation.hs
  GHC-Options:       -Wall
  Hs-source-dirs:    src
  Build-Depends:     base ==4.*,
                     cereal ==0.4.*,
                     creatur ==5.9.5,
                     directory ==1.2.*,
                     ghc-prim ==0.3.* || ==0.4.*,
                     MonadRandom ==0.3.*,
                     mtl ==2.2.*,
                     old-locale ==1.0.*,
                     random ==1.1.*,
                     time ==1.4.* || ==1.5.*,
                     transformers ==0.4.*,
                     zlib ==0.5.*

Executable chapter11
  Main-Is:           Tutorial/Chapter11/Example.hs
  GHC-Options:       -Wall
  Hs-source-dirs:    src
```

```
Build-Depends:     base ==4.*,
                   cereal ==0.4.*,
                   creatur ==5.9.5,
                   directory ==1.2.*,
                   ghc-prim ==0.3.* || ==0.4.*,
                   MonadRandom ==0.3.*,
                   mtl ==2.2.*,
                   old-locale ==1.0.*,
                   time ==1.4.* || ==1.5.*,
                   transformers ==0.4.*,
                   zlib ==0.5.*
```

1. If you haven't already done so, follow the instructions in Section 2.1.
2. Create the initial population by running `chapter5-init`.
3. Start the daemon with the command `sudo chapter5-daemon start`.
4. Stop the daemon with the command `sudo chapter5-daemon stop`. (Stopping the daemon may take a few seconds.)

Log messages are sent to `chapter5/log/Chapter5.log`. Examine that file and notice that the counter is counting up for both rocks. If you stop the daemon and restart it, it will pick up where it left off [1]

A sample extract from the log file is shown below. The first field is the system clock time. The second field is the daemon clock time. The third field is the log message. Note that in clock tick `0`, `Rocky` gets to use the CPU before `Rocky`, while in clock tick `3`, `Roxie` goes first. This demonstrates the randomisation discussed in section 3.

```
130121121233+0000       Starting
130121121233+0000       0       Rocky's counter is 42
130121121233+0000       0       Roxie's counter is 99
130121121233+0000       1       Rocky's counter is 43
130121121233+0000       1       Roxie's counter is 100
130121121233+0000       2       Roxie's counter is 101
130121121233+0000       2       Rocky's counter is 44
130121121233+0000       3       Roxie's counter is 102
130121121233+0000       3       Rocky's counter is 45
130121121234+0000       4       Rocky's counter is 46
130121121234+0000       4       Roxie's counter is 103
```

5.5 Key points

- Any species used in the Créatúr framework must be an instance of `ALife.Creatur.Agent` and `Serialize`. Normally the species should also be an instance of `ALife.Creatur.Database.Record`.
- It is generally not necessary to write an implementation of `Serialize`. Instead, add `deriving Generic` to the declaration, and declare `instance Serialize MyClass`.
- Consult the haddock documentation (http://hackage.haskell.org/package/creatur) for more detailed information about the API.

[1] When the stop command is received, the daemon will attempt to finish the processing for the current agent. Depending on the processor speed and the complexity of the agent task, it may not finish before the hard kill which is issued three seconds later. The database or file system is not updated until an agent's turn at the CPU is complete, so any partial results are discarded. If the daemon terminates while an agent is running, it "remembers" the state of the queue; that agent's turn will start over when the daemon is restarted.

Chapter 6

Recombination

When agents reproduce, the offspring will inherit a mixture of genetic information from both parents. Here are two scenarios that could be used, although there are other possibilities.

1. Your agents use *asexual* reproduction. Each agent has a *single* sequence of genetic information. When two agents mate, their genes are shuffled to produce two *new* sequences. You can create two children from these sequences, or discard one sequence and create a child with the remaining sequence.

2. Your agents use *sexual* reproduction. Each agent has *two* sequences of genetic information. When two agents mate, each agent contributes *one* sequence to the child. A parent's two sequences are shuffled to produce two *new* sequences. One of the sequences is discarded; the other sequence becomes that parent's contribution to the child's genome. The same process occurs with the other parent's genome. The two sequences (one from each parent) are combined to create the child's genome. This is analogous to the production of a *gamete* (ovum or sperm) in biology.

Both scenarios involve shuffling a pair of sequences to produce two new pairs, and possibly discarding one of the sequences. In addition, you may wish to allow occasional random mutations. The Créatúr framework provides the several operations for this purpose, in the `ALife.Creatur.Genetics.Recombination` package. These operations can be applied (multiple times) with specified probabilities and combined in various ways. Two common operations, *crossover* and *cut-and-splice*, are illustrated below. In crossover, a single crossover point is chosen. All data beyond that point is swapped between strings. In cut-and-splice, two points are chosen, one on each string. This generally results in two strings of unequal length.

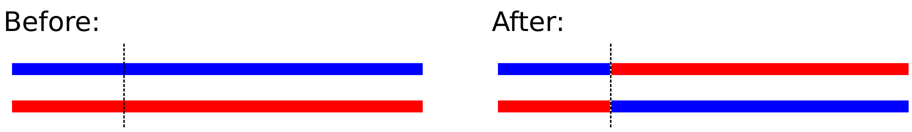

Figure6.1: Crossover

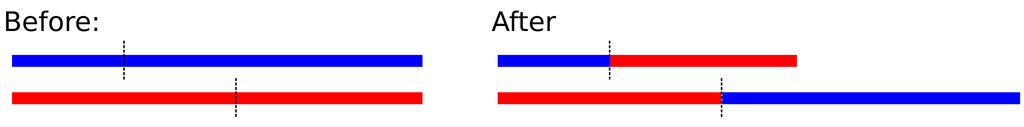

Figure6.2: Cut-and-splice

Here's a sample program that might be used to shuffle two sequences of genetic material.

```
withProbability 0.1 randomCrossover (xs, ys) >>=
withProbability 0.01 randomCutAndSplice >>=
withProbability 0.001 mutatePairedLists >>=
randomOneOfPair
```

To understand how this program works, let's walk through a simple example. Suppose this program acted on the following pair of sequences:

```
([A,A,A,A,A,A,A,A,A,A],[C,C,C,C,C,C,C,C,C,C])
```

The `randomCrossover` function *might* perform a simple crossover, perhaps resulting in:

```
([A,A,A,A,A,A,A,C,C,C],[C,C,C,C,C,C,C,A,A,A])
```

The`randomCutAndSplice` function *might* then perform a cut-and-splice, perhaps resulting in:

```
([A,A,A,A,C,A,A,A],[C,C,C,C,C,C,A,A,A,C,C,C])
```

The `mutatePairedLists` function *might* then mutate one or both sequences, perhaps resulting in

```
([T,A,A,A,C,A,A,A],[C,C,C,C,C,C,A,A,C,C,C,C])
```

The numbers 0.1, 0.01, and 0.001 control the likelihood of each of the three operations occurring. After the first three operations, we have two new sequences. In this example, we only want one of the sequences, so the final line randomly chooses one.

To perform more than one crossover, the operation can simply be repeated as shown below.

```
withProbability 0.1 randomCrossover (xs, ys) >>=
withProbability 0.08 randomCrossover (xs, ys) >>=
. . .
```

Alternatively, we can choose the number of crossover operations at random. The function `repeatWithProbability` performs an operation a random number of times, such that the probability of repeating the operation `n` times is p^n.

```
repeatWithProbability 0.1 randomCrossover (xs, ys) >>=
. . .
```

Other recombination operators are also available. Consult the documentation of `ALife.Creatur.Genetics.Recombination` for more information.

Chapter 7

Simplified sexual reproduction

In this part of the tutorial, we create a species with a greatly simplified form of sexual reproduction that is often used in artificial life and genetic programming

7.1 Create a species

The species used in this example is called `Plant`. Each `Plant` has a unique ID, a flower colour, an energy level, and some genetic information. (A complete listing of the source code discussed here is provided on page 17.)

```
data Plant = Plant
  {
    plantName :: String,
    plantFlowerColour :: FlowerColour,
    plantEnergy :: Int,
    plantGenome :: Sequence
  } deriving (Show, Generic)
```

As with `Rock`s, the type `Plant` will be an instance of the `Serialize`, `Agent` and `Record` classes. Our plants will stay alive until all of their energy is gone.

```
instance Serialize Plant

instance Agent Plant where
  agentId = plantName
  isAlive plant = plantEnergy plant > 0

instance Record Plant where key = agentId
```

We'll have a choice of flower colours.

```
data FlowerColour = Red | Orange | Yellow | Violet | Blue
  deriving (Show, Eq, Generic, Enum, Bounded)
```

In order for `Plant` to be an instance of `Serialize`, any type that it uses must also be an instance. So we make `FlowerColour` be an instance of `Serialize`.

```
instance Serialize FlowerColour
```

We need a way to encode the plant genes into DNA-like sequences that can be shuffled, or even mutated, during reproduction. We'll encode the genes as sequences of `Bool`s. We could write our own coding scheme, but `ALife.Creatur.Genetics.Code.BRGCBool` provides a scheme for us, using a class called `Genetic`. The `Genetic` class provides the method `put`, which encodes a gene and writes it to a sequence, and the method `get`, which reads and decodes the first gene in a sequence. By making `FlowerColour` an instance of `Genetic`, `FlowerColour` will be encoded as a string of boolean values using a *Gray code*. A Gray code maps values to codes in a way that guarantees that the codes for two consecutive values will differ by only one bit. This feature is useful for encoding genes because the result of a crossover operation will be similar to the inputs. This helps to ensure that offspring are similar to their parents, as any radical changes from one generation to the next are the result of mutation alone.

When the genes of an agent have a small set of possible values, it is practical to store their genetic information as a string of `Bool`s. (If an agent has genes with a larger number of possible values, it may be better to store their genetic information as a string of numbers. Créatúr also provides `ALife.Creatur.Genetics.Code.BRGCWord8`, which encodes the genes using a Gray code, but stores them using a string of `Word8`s. Since `BRGCWord8` provides most of the same functions as `BRGCBool`, all we need to do to make `Plant` use `Word8`s is to import `BRGCWord8` instead of `BRGCBool`.) We'll see an example of this in Section 9.1.

```haskell
instance Genetic FlowerColour
```

To support reproduction, we need a way to build a plant from its genome. First, each plant needs a copy of its genome in order to produce offspring; we'll use the `copy` method to obtain this. Next, we determine the colour of the bug. We could use the method `get` in the class `Genetic`, which returns a `Maybe` value containing the next gene in a sequence. However, our sequence of `Bool`s may not be a valid code for a colour, in which case the call to `get` would return `Nothing`. In this example, we will create a plant no matter what errors there are in the genome, so we will use `getWithDefault`, using `Red` as the default value. (Alternatively, we could treat the mutation as non-viable, and not create the offspring. We'll see an example of that in Section 9.1. All plants start life with an energy of `10`.

```haskell
buildPlant :: String -> Reader (Either [String] Plant)
buildPlant name = do
  g <- copy
  colour <- getWithDefault Red
  return . Right $ Plant name colour 10 g
```

We need a way to mate two plants and produce some offspring. We can do this by implementing the `Reproductive` class in `ALife.Creatur.Genetics.Reproduction.SimplifiedSexual`. (In reality, plants use a variety of reproduction methods.) This class requires us to implement the following:

1. A type called `Base`, which specifies the type used to encode genes for this species. Recall that we've used `Bool`s for this purpose.
2. A method called `recombine` which shuffles (and maybe mutates) the parent's genes to produce the offspring.
3. A method called `build` which creates the offspring. We can call the `read` method in the `Reproductive` class, supplying the `buildPlant` method as an argument.

```haskell
instance Reproductive Plant where
  type Strand Plant = Sequence
  recombine a b =
    withProbability 0.1 randomCrossover (plantGenome a, plantGenome b) >>=
    withProbability 0.01 randomCutAndSplice >>=
    withProbability 0.001 mutatePairedLists >>=
    randomOneOfPair
  build name = runReader (buildPlant name)
```

The implementation for `recombine` uses the sample recombination program discussed on page 14. Next, we write the function `run`, which is invoked when it is the agent's turn to use the CPU. Because our plants need to interact in order to mate, when we write the daemon (in Section 7.3) we will use `runInteractingAgents` instead of `runNoninteractingAgents`. This requires a different type signature for `run` than we used for rocks. The type signature we need is

```haskell
type AgentsProgram u = [Agent u] -> StateT u IO [Agent u]
```

The input parameter is a list of agents. The first agent in the list is the agent whose turn it is to use the CPU. The rest of the list contains agents it could interact with. (We only need to use the first two elements of this list.) Finally, the program must return a list of agents that it has modified.

The function `run` is invoked when it is the agent's turn to use the CPU. It takes a list of all agents in the population, with the current agent at the head of the list. It returns a list of agents that have been created or modified during the turn. In this implementation, `run` "mates" two plants and takes away one unit of energy to represent the energy cost of reproduction (otherwise the plants would live forever).

```haskell
run :: [Plant] -> StateT (SimpleUniverse Plant) IO [Plant]
run (me:other:_) = do
  name <- genName
  (Right baby) <- liftIO $ evalRandIO (makeOffspring me other name)
  writeToLog $
    plantName me ++ " and " ++ plantName other ++
      " gave birth to " ++ name ++ ", with " ++
       show (plantFlowerColour baby) ++ " flowers"
  return [deductMatingEnergy me, deductMatingEnergy other, baby]
run _ = return [] -- need two agents to mate
```

The complete code listing is below.

```haskell
{-# LANGUAGE DeriveGeneric, FlexibleContexts, TypeFamilies #-}
module Tutorial.Chapter7.Plant (Plant(..), FlowerColour(..),
  buildPlant, run) where

import ALife.Creatur (Agent, agentId, isAlive)
import ALife.Creatur.Database (Record, key)
import ALife.Creatur.Genetics.BRGCBool (Genetic, Reader, Sequence,
  getWithDefault, runReader, copy)
import ALife.Creatur.Genetics.Recombination (mutatePairedLists,
  randomCrossover, randomCutAndSplice, randomOneOfPair, withProbability)
import ALife.Creatur.Genetics.Reproduction.SimplifiedSexual
  (Reproductive, Strand, recombine, build, makeOffspring)
import ALife.Creatur.Universe (SimpleUniverse, genName, writeToLog)
import Control.Monad.IO.Class (liftIO)
import Control.Monad.Random (evalRandIO)
import Control.Monad.State (StateT)
import Data.Serialize (Serialize)
import GHC.Generics (Generic)

data Plant = Plant
  {
    plantName :: String,
    plantFlowerColour :: FlowerColour,
    plantEnergy :: Int,
    plantGenome :: Sequence
  } deriving (Show, Generic)

instance Serialize Plant

instance Agent Plant where
  agentId = plantName
  isAlive plant = plantEnergy plant > 0

instance Record Plant where key = agentId

data FlowerColour = Red | Orange | Yellow | Violet | Blue
  deriving (Show, Eq, Generic, Enum, Bounded)

instance Serialize FlowerColour
instance Genetic FlowerColour

buildPlant :: String -> Reader (Either [String] Plant)
buildPlant name = do
  g <- copy
  colour <- getWithDefault Red
  return . Right $ Plant name colour 10 g

instance Reproductive Plant where
  type Strand Plant = Sequence
  recombine a b =
    withProbability 0.1 randomCrossover (plantGenome a, plantGenome b) >>=
    withProbability 0.01 randomCutAndSplice >>=
    withProbability 0.001 mutatePairedLists >>=
    randomOneOfPair
  build name = runReader (buildPlant name)

run :: [Plant] -> StateT (SimpleUniverse Plant) IO [Plant]
run (me:other:_) = do
  name <- genName
  (Right baby) <- liftIO $ evalRandIO (makeOffspring me other name)
  writeToLog $
    plantName me ++ " and " ++ plantName other ++
```

```
      " gave birth to " ++ name ++ ", with " ++
       show (plantFlowerColour baby) ++ " flowers"
  writeToLog $ "Me: " ++ show me
  writeToLog $ "Mate: " ++ show other
  writeToLog $ "Baby: " ++ show baby
  return [deductMatingEnergy me, deductMatingEnergy other, baby]
run _ = return [] -- need two agents to mate

deductMatingEnergy :: Plant -> Plant
deductMatingEnergy p = p {plantEnergy=plantEnergy p - 1}
```

7.2 Generate an initial population

We will create the next population in a directory called `chapter7`. In `main`, we initialise the `chapter7` directory, create three `Plant`s, and write them to the directory. The initial population only contains plants with red, yellow, or violet flowers. Eventually, plants with orange or blue flowers might appear, but only as the result of mutation. (In Section 9.2, we introduce a different technique for creating an initial population, using random genetic sequences.) The complete code listing is below.

```
import Tutorial.Chapter7.Plant (FlowerColour(..), buildPlant)
import ALife.Creatur.Universe (store, mkSimpleUniverse)
import ALife.Creatur.Genetics.BRGCBool (write, runReader)
import Control.Monad.State.Lazy (evalStateT)

main :: IO ()
main = do
  let u = mkSimpleUniverse "Chapter7" "chapter7"

  -- Create some plants and save them in the population directory.
  let g1 = write Red
  let (Right p1) = runReader (buildPlant "Rose") g1
  evalStateT (store p1) u

  let g2 = write Yellow
  let (Right p2) = runReader (buildPlant "Sunny") g2
  evalStateT (store p2) u

  let g3 = write Violet
  let (Right p3) = runReader (buildPlant "Vi") g3
  evalStateT (store p3) u

  return ()
```

7.3 Configure a daemon

The program below configures and launches the daemon. Note that this time we call `runInteractingAgents` instead of `runNoninteractingAgents`. The argument `2` indicates that we need at least two agents in the population (for mating).

```
module Main where

import Tutorial.Chapter7.Plant (run)
import ALife.Creatur.Daemon (CreaturDaemon(..), Job(..),
  simpleDaemon, launch)
import ALife.Creatur.Universe (mkSimpleUniverse)
import ALife.Creatur.Task (simpleJob, runInteractingAgents,
  doNothing)
import System.Directory (canonicalizePath)
```

```haskell
main :: IO ()
main = do
  dir <- canonicalizePath "chapter7" -- Required for daemon
  let u = mkSimpleUniverse "Chapter7" dir
  let j = simpleJob
        { task=runInteractingAgents run doNothing doNothing,
          sleepTime=0 }
  launch $ CreaturDaemon (simpleDaemon j u) j
```

7.4 Build and run the example

1. If you haven't already done so, follow the instructions in Section 2.1.
2. Create the initial population by running `chapter7-init`.
3. Start/stop/restart the daemon with the command `sudo chapter7-daemon start|stop|restart`. (Stopping the daemon may take a few seconds.)

Log messages are sent to `chapter7/log/Chapter7.log`.

A sample extract from the log file is shown below. To conserve space, the timestamps have been omitted. From this, you can see that the first mating (between `Rose` and `Sunny`) occurs at time 0. The offspring, `Chapter7_1`, gets its first CPU turn at time 1.

```
Starting
0       Vi and Rose gave birth to Chapter7_1, with Red flowers
0       Rose and Vi gave birth to Chapter7_2, with Violet flowers
0       Sunny and Rose gave birth to Chapter7_3, with Red flowers
1       Chapter7_1 and Sunny gave birth to Chapter7_4, with Yellow flowers
1       Rose and Vi gave birth to Chapter7_5, with Violet flowers
1       Chapter7_3 and Chapter7_2 gave birth to Chapter7_6, with Red flowers
1       Chapter7_2 and Sunny gave birth to Chapter7_7, with Violet flowers
1       Vi and Chapter7_7 gave birth to Chapter7_8, with Violet flowers
1       Sunny and Chapter7_8 gave birth to Chapter7_9, with Yellow flowers
2       Chapter7_4 and Rose gave birth to Chapter7_10, with Yellow flowers
2       Chapter7_7 and Chapter7_6 gave birth to Chapter7_11, with Violet flowers
2       Chapter7_6 and Rose gave birth to Chapter7_12, with Red flowers
2       Chapter7_5 and Chapter7_10 gave birth to Chapter7_13, with Violet flowers
2       Chapter7_3 and Chapter7_7 gave birth to Chapter7_14, with Red flowers
2       Chapter7_8 and Chapter7_14 gave birth to Chapter7_15, with Red flowers
2       Rose and Chapter7_14 gave birth to Chapter7_16, with Red flowers
2       Sunny and Chapter7_12 gave birth to Chapter7_17, with Yellow flowers
2       Vi and Chapter7_5 gave birth to Chapter7_18, with Violet flowers
```

7.5 Key points

- A species that reproduces should be an instance of the `Genetic` class in one of the `ALife.Creatur.Genetics.Code.*` modules, e.g., `ALife.Creatur.Genetics.Code.BRGCBool`.
- It is generally not necessary to write an implementation of `Genetic`. Instead, add `deriving Generic` to the declaration, and declare `instance Genetic MyClass`.
- In order for a type to use a generic implementation for a class such as `Serialize` or `Genetic`, any type that it "contains" must also be an instance of that class.
- Encoding genetic information using a Gray code helps to ensure that offspring are similar to their parents (unless a mutation occurs).
- The module `ALife.Creatur.Genetics.Recombination` provides functions for controlling how the genes of the parents are shuffled to produce the child's genome.

Chapter 8

Sexual reproduction

In this part of the tutorial, we create a species with the ability to reproduce sexually.

8.1 Create a species

We create a type to represent the `Bug` species. Each bug has a unique ID, a colour, sex, an energy level, and some genetic information. Because our bugs will reproduce *sexually*, they have *two* sets of genes, one inherited from each parent. Thus, the genome will be stored as a `DiploidSequence` instead of a `Sequence`.

```
data Bug = Bug
  {
    bugName :: String,
    bugColour :: BugColour,
    bugSex :: Sex,
    bugEnergy :: Int,
    bugGenome :: DiploidSequence
  } deriving (Show, Generic)

instance Serialize Bug
```

We make `Bug` implement the `Agent` and `Record` classes. Our bugs will stay alive until their energy reaches zero.

```
instance Agent Bug where
  agentId = bugName
  isAlive bug = bugEnergy bug > 0

instance Record Bug where key = agentId
```

We create the genes for colour and sex.

```
data BugColour = Green | Purple
  deriving (Show, Eq, Enum, Bounded, Generic)
instance Serialize BugColour
instance Genetic BugColour

data Sex = Male | Female
  deriving (Show, Eq, Enum, Bounded, Generic)
instance Serialize Sex
instance Genetic Sex
```

Recall that our bugs have *two* sets of genes. These genes may not be identical, so we need a way to determine the resulting colour of the bug from its genes. The `Diploid` class contains a method called `generic`, which, given two possible forms of a gene, takes into account any dominance relationship, and returns a gene representing the result.

We could write an implementation of `express`, but the `Diploid` class provides a generic implementation. The generic implementation of `express` chooses the ”smaller” of the two values. For numeric values, this simply means taking the minimum of the two values, so a gene with a value of `3.5` is dominant over one with a value of `7.6`. For types with multiple constructors, the constructors that appear earlier in the definition are dominant over those that appear later, so a `Male` gene will be dominant over a `Female` gene. In other words, a bug with two `Female` genes is female, but a bug with at least one `Male` gene is male. This is loosely based on the XY-chromosome system used by humans and some other animals. (To avoid giving the males too much power, we'll let the females initiate mating!)

```haskell
instance Diploid BugColour
instance Diploid Sex
```

To support reproduction, we need a way to build a bug from its genome. Like the plants we created earlier, each bug needs a copy of its genome in order to produce offspring. For plants, we used the method `copy` to get the unread genetic information. However, bugs have two sets of genes, so we use `copy2` instead. Next, we determine the sex and colour of the bug from its genome. We could use the method `getAndExpress` in the module `BRGCBool`, which returns a `Maybe` value containing a tuple with the first gene in a sequence, and the rest of the paired sequences. It may happen that neither sequence of `Bool`s begins with a valid code for a colour or for the sex, in which case the call to `getAndExpress` would return `Nothing`. For convenience, we'll use the `getAndExpressWithDefault` method instead, supplying a default sex and colour. All bugs start life with 10 units of energy.

```haskell
buildBug :: String -> DiploidReader (Either [String] Bug)
buildBug name = do
  g <- copy2
  sex <- getAndExpressWithDefault Female
  colour <- getAndExpressWithDefault Green
  return . Right $ Bug name colour sex 10 g
```

Next, we need a way to mate two bugs and produce some offspring. We can do this by implementing the `Reproductive` class in `ALife.Creatur.Genetics.Reproduction.Sexual`. This class requires us to implement the following:

1. A type called `Strand`, which specifies the type used for encoded genes for this species. Recall that we've used `Bool`s for this purpose.
2. A method called `produceGamete` which shuffles (and maybe mutates) the two sequences of genes from one parent, and then produces a *single* sequence that will become part of the child's genome. (This is analogous to creating either a single sperm or ova.)
3. A method called `build` which creates the offspring. We can use the `buildBug` method that we've already created, and pass it to `runReader`.

```haskell
instance Reproductive Bug where
  type Strand Bug = Sequence
  produceGamete a =
    repeatWithProbability 0.1 randomCrossover (bugGenome a) >>=
    withProbability 0.01 randomCutAndSplice >>=
    withProbability 0.001 mutatePairedLists >>=
    randomOneOfPair
  build name = runDiploidReader (buildBug name)
```

Although the implementation of `produceGamete` is similar to that of `recombine` for the `Plant` class, these two functions have different uses. Asexual reproduction uses `recombine` to mix the genetic information from two parents; the resulting sequence becomes the entire genome of the child. Sexual reproduction uses `produceGamete` to mix the genetic information from *one* parent; the resulting sequence becomes *half* of the genome of the child. The other half of the child's genome comes from the other parent, also generated using `produceGamete`.

The function `run` is invoked when it is the agent's turn to use the CPU. It takes a list of all agents in the population, with the current agent at the head of the list. It returns a list of agents that have been created or modified during the turn. Let's let the females initiate mating. If this bug is female, and the second bug is male, then mating occurs. If mating occurs, we deduct one unit of energy. If mating does not occur, then no agents have been modified so we return an empty list.

```haskell
run :: [Bug] -> StateT (SimpleUniverse Bug) IO [Bug]
run (me:other:_) = do
  writeToLog $ agentId me ++ "'s turn"
  if bugSex me == Female && bugSex other == Male
    then do
      name <- genName
      (Right baby) <- liftIO $ evalRandIO (makeOffspring me other name)
      writeToLog $
        bugName me ++ " and " ++ bugName other ++
          " gave birth to " ++ name ++ ", a " ++
          show (bugColour baby) ++ " " ++ show (bugSex baby) ++ " bug"
      return [deductMatingEnergy me, deductMatingEnergy other, baby]
    else return []
run _ = return [] -- need two agents to mate

deductMatingEnergy :: Bug -> Bug
deductMatingEnergy bug = bug {bugEnergy=bugEnergy bug - 1}
```

The complete code listing is below.

```haskell
{-# LANGUAGE DeriveGeneric, FlexibleContexts, TypeFamilies #-}
module Tutorial.Chapter8.Bug (Bug(..), Sex(..), BugColour(..),
  buildBug, run) where

import ALife.Creatur (Agent, agentId, isAlive)
import ALife.Creatur.Database (Record, key)
import ALife.Creatur.Genetics.BRGCBool (Genetic, Sequence,
  DiploidSequence, DiploidReader, getAndExpressWithDefault,
  runDiploidReader, copy2)
import ALife.Creatur.Genetics.Diploid (Diploid)
import ALife.Creatur.Genetics.Recombination (mutatePairedLists,
  randomCrossover, randomCutAndSplice, randomOneOfPair,
  repeatWithProbability, withProbability)
import ALife.Creatur.Genetics.Reproduction.Sexual (Reproductive, Strand,
  produceGamete, build, makeOffspring)
import ALife.Creatur.Universe (SimpleUniverse, genName, writeToLog)
import Control.Monad.IO.Class (liftIO)
import Control.Monad.Random (evalRandIO)
import Control.Monad.State (StateT)
import Data.Serialize (Serialize)
import GHC.Generics (Generic)

data Bug = Bug
  {
    bugName :: String,
    bugColour :: BugColour,
    bugSex :: Sex,
    bugEnergy :: Int,
    bugGenome :: DiploidSequence
  } deriving (Show, Generic)

instance Serialize Bug

instance Agent Bug where
  agentId = bugName
  isAlive bug = bugEnergy bug > 0

instance Record Bug where key = agentId

data BugColour = Green | Purple
  deriving (Show, Eq, Enum, Bounded, Generic)
instance Serialize BugColour
instance Genetic BugColour
```

```haskell
instance Diploid BugColour

data Sex = Male | Female
  deriving (Show, Eq, Enum, Bounded, Generic)
instance Serialize Sex
instance Genetic Sex
instance Diploid Sex

buildBug :: String -> DiploidReader (Either [String] Bug)
buildBug name = do
  g <- copy2
  sex <- getAndExpressWithDefault Female
  colour <- getAndExpressWithDefault Green
  return . Right $ Bug name colour sex 10 g

instance Reproductive Bug where
  type Strand Bug = Sequence
  produceGamete a =
    repeatWithProbability 0.1 randomCrossover (bugGenome a) >>=
    withProbability 0.01 randomCutAndSplice >>=
    withProbability 0.001 mutatePairedLists >>=
    randomOneOfPair
  build name = runDiploidReader (buildBug name)

run :: [Bug] -> StateT (SimpleUniverse Bug) IO [Bug]
run (me:other:_) = do
  writeToLog $ agentId me ++ "'s turn"
  if bugSex me == Female && bugSex other == Male
    then do
      name <- genName
      (Right baby) <- liftIO $ evalRandIO (makeOffspring me other name)
      writeToLog $
        bugName me ++ " and " ++ bugName other ++
          " gave birth to " ++ name ++ ", a " ++
          show (bugColour baby) ++ " " ++ show (bugSex baby) ++ " bug"
      writeToLog $ "Mother: " ++ show me
      writeToLog $ "Father: " ++ show other
      writeToLog $ "Baby: " ++ show baby
      return [deductMatingEnergy me, deductMatingEnergy other, baby]
    else return []
run _ = return [] -- need two agents to mate

deductMatingEnergy :: Bug -> Bug
deductMatingEnergy bug = bug {bugEnergy=bugEnergy bug - 1}
```

8.2 Generate an initial population

We will create the next population in a directory called Chapter8. In main, we initialise the chapter8 directory, create three Bugs, and write them to the directory. The bugs in the initial population will each have two identical sequences of genetic material. (In Section 9.2, we introduce a different technique for creating an initial population, using random genetic sequences.) The complete code listing is below.

```haskell
import Tutorial.Chapter8.Bug (Sex(..), BugColour(..), buildBug)
import ALife.Creatur.Universe (store, mkSimpleUniverse)
import ALife.Creatur.Genetics.BRGCBool (put, runWriter,
  runDiploidReader)
import Control.Monad.State.Lazy (evalStateT)

main :: IO ()
main = do
  let u = mkSimpleUniverse "Chapter8" "chapter8"
```

```haskell
  -- Create some Bugs and save them in the population directory.
  let g1 = runWriter (put Male >> put Green)
  let (Right b1) = runDiploidReader (buildBug "Bugsy") (g1,g1)
  evalStateT (store b1) u

  let g2 = runWriter (put Male >> put Purple)
  let (Right b2) = runDiploidReader (buildBug "Mel") (g2,g2)
  evalStateT (store b2) u

  let g3 = runWriter (put Female >> put Green)
  let (Right b3) = runDiploidReader (buildBug "Flo") (g3, g3)
  evalStateT (store b3) u

  let g4 = runWriter (put Male >> put Purple)
  let (Right b4) = runDiploidReader (buildBug "Buzz") (g4, g4)
  evalStateT (store b4) u
```

8.3 Configure a daemon

The program below configures and launches the daemon.

```haskell
module Main where

import Tutorial.Chapter8.Bug (run)
import ALife.Creatur.Daemon (CreaturDaemon(..), Job(..),
  simpleDaemon, launch)
import ALife.Creatur.Universe (mkSimpleUniverse)
import ALife.Creatur.Task (simpleJob, runInteractingAgents,
  doNothing)
import System.Directory (canonicalizePath)

main :: IO ()
main = do
  dir <- canonicalizePath "chapter8" -- Required for daemon
  let u = mkSimpleUniverse "Chapter8" dir
  let j = simpleJob
        { task=runInteractingAgents run doNothing doNothing,
          sleepTime=0 }
  launch $ CreaturDaemon (simpleDaemon j u) j
```

8.4 Build and run the example

1. If you haven't already done so, follow the instructions in Section 2.1.
2. Create the initial population by running `chapter8-init`.
3. Start/stop/restart the daemon with the command `sudo chapter8-daemon start|stop|restart`. (Stopping the daemon may take a few seconds.)

Log messages are sent to `chapter8/log/Chapter8.log`.
A sample extract from the log file is shown below.

```
150416183914+0100       0       Starting
150416183914+0100       1       Beginning of round
150416183914+0100       1       Bugsy's turn
150416183914+0100       1       Mel's turn
150416183914+0100       1       Buzz's turn
150416183914+0100       1       Flo's turn
150416183914+0100       1       Flo and Bugsy gave birth to Chapter8_1, a Green Male bug
150416183914+0100       1       Mother: Bug {bugName = "Flo", bugColour = Green, bugSex = Female,
```

```
bugEnergy = 10, bugGenome = ([False,True],[False,True])}
150416183914+0100       1       Father: Bug {bugName = "Bugsy", bugColour = Green, bugSex = Male,
bugEnergy = 10, bugGenome = ([True,True],[True,True])}
150416183914+0100      1      Baby: Bug {bugName = "Chapter8_1", bugColour = Green, bugSex = Male,
bugEnergy = 10, bugGenome = ([False,True],[True,True])}
150416183914+0100       1       Flo returned to population
150416183914+0100       1       Bugsy returned to population
150416183914+0100       1       Chapter8_1 added to population
150416183914+0100       1       End of round
150416183914+0100       2       Beginning of round
150416183914+0100       2       Chapter8_1's turn
150416183914+0100       2       Bugsy's turn
150416183914+0100       2       Flo's turn
150416183914+0100       2       Flo and Buzz gave birth to Chapter8_2, a Green Male bug
150416183914+0100       2       Mother: Bug {bugName = "Flo", bugColour = Green, bugSex = Female,
bugEnergy = 9, bugGenome = ([False,True],[False,True])}
150416183914+0100       2       Father: Bug {bugName = "Buzz", bugColour = Purple, bugSex = Male,
bugEnergy = 10, bugGenome = ([True,False],[True,False])}
150416183914+0100      2      Baby: Bug {bugName = "Chapter8_2", bugColour = Green, bugSex = Male,
bugEnergy = 10, bugGenome = ([False,True],[True,False])}
150416183914+0100       2       Flo returned to population
150416183914+0100       2       Buzz returned to population
150416183914+0100       2       Chapter8_2 added to population
150416183914+0100       2       Buzz's turn
150416183914+0100       2       Mel's turn
150416183914+0100       2       End of round
```

8.5 Key points

- A species that reproduces *sexually* should be an instance of `Diploid` (as well as `Genetic`).
- It is generally not necessary to write an implementation of `Diploid`. Instead, add `deriving Generic` to the declaration, and declare `instance Diploid MyClass`.
- The generic implementation of `Diploid` chooses the "smaller" of two values. For types with multiple constructors, this means that constructors that appear earlier in the definition are dominant over those that appear later.
- The list of agents returned by the "run" method only needs to contain agents that have been modified.

Chapter 9

Generating a random initial population

In this part of the tutorial, we generate a random initial population.

9.1 Create a species

Let's make or bugs more interesting by adding spots.

```
data Bug = Bug
  {
    bugName :: String,
    bugColour :: BugColour,
    bugSpots :: [BugColour],
    bugSex :: Sex,
    bugEnergy :: Int,
    bugGenome :: DiploidSequence
  } deriving (Show, Generic)
```

We'll also allow a broader range of colours.

```
data BugColour = Green | Purple | Red | Brown | Orange | Pink | Blue
  deriving (Show, Eq, Enum, Bounded, Generic)
```

Creating a random sequence of genes is not difficult. But how long should the string be? We'd have to calculate the number of bits required to represent the colours we've allowed. Furthermore, the length of the sequence depends on how many spots the bug has. So we need to know the at least part of the decoded value of the sequence in order to determine the length required. We could just create a random gene sequence that is longer than we expect to need; the extra genes won't do any harm, and might eventually become useful as the result of recombination. But that is somewhat wasteful.

It might be better to have the initial population start with a "clean" genome where the entire sequence is used. Recombination will eventually make some sequences longer, and others shorter, but on average we would expect the sequences to be a reasonable length.

Fortunately, there is an easy way to do this. When creating our initial population, we can pass `buildBug` an infinite gene sequence, but instruct it to keep only as much of the sequence as it needs to build a complete bug. We add a flag to the `buildBug` function to tell it whether it should truncate the sequence (which is what we want when creating the initial population), or keep the entire gene sequence (which is what we want during normal operation). Another change is that we used `getAndExpress` instead of `getAndExpressWithDefault`. If the genome is invalid, `buildBug` will return `Nothing`.

```
buildBug :: Bool -> String -> DiploidReader (Either [String] Bug)
buildBug truncateGenome name = do
  sex <- getAndExpress
  colour <- getAndExpress
  spots <- getAndExpress
  g <- if truncateGenome then consumed2 else copy2
  return $ Bug name <$> sex <*> colour <*> spots <*> pure 10 <*> pure g
```

The complete code listing is below.

```
{-# LANGUAGE DeriveGeneric, FlexibleContexts, TypeFamilies #-}
{-# LANGUAGE CPP #-}
```

```haskell
module Tutorial.Chapter9.Bug (Bug(..), Sex(..), BugColour(..),
  buildBug, run) where

import ALife.Creatur (Agent, agentId, isAlive)
import ALife.Creatur.Database (Record, key)
import ALife.Creatur.Genetics.BRGCBool (Genetic, Sequence,
  DiploidSequence, DiploidReader, getAndExpress,
  runDiploidReader, copy2, consumed2)
import ALife.Creatur.Genetics.Diploid (Diploid)
import ALife.Creatur.Genetics.Recombination (mutatePairedLists,
  randomCrossover, randomCutAndSplice, randomOneOfPair,
  repeatWithProbability, withProbability)
import ALife.Creatur.Genetics.Reproduction.Sexual (Reproductive, Strand,
  produceGamete, build, makeOffspring)
import ALife.Creatur.Universe (SimpleUniverse, genName, writeToLog)
import Control.Monad.IO.Class (liftIO)
import Control.Monad.Random (evalRandIO)
import Control.Monad.State (StateT)
import Data.Serialize (Serialize)
import GHC.Generics (Generic)

#if MIN_VERSION_base(4,8,0)
-- Starting with GHC 7.10 (base 4.8), we don't need to import
-- Control.Applicative
#else
import Control.Applicative
#endif

data Bug = Bug
  {
    bugName :: String,
    bugColour :: BugColour,
    bugSpots :: [BugColour],
    bugSex :: Sex,
    bugEnergy :: Int,
    bugGenome :: DiploidSequence
  } deriving (Show, Generic)

instance Serialize Bug

instance Agent Bug where
  agentId = bugName
  isAlive bug = bugEnergy bug > 0

instance Record Bug where key = agentId

data BugColour = Green | Purple | Red | Brown | Orange | Pink | Blue
  deriving (Show, Eq, Enum, Bounded, Generic)
instance Serialize BugColour
instance Genetic BugColour
instance Diploid BugColour

data Sex = Male | Female
  deriving (Show, Eq, Enum, Bounded, Generic)
instance Serialize Sex
instance Genetic Sex
instance Diploid Sex

buildBug :: Bool -> String -> DiploidReader (Either [String] Bug)
buildBug truncateGenome name = do
  sex <- getAndExpress
  colour <- getAndExpress
  spots <- getAndExpress
```

```
  g <- if truncateGenome then consumed2 else copy2
  return $ Bug name <$> sex <*> colour <*> spots <*> pure 10 <*> pure g

instance Reproductive Bug where
  type Strand Bug = Sequence
  produceGamete a =
    repeatWithProbability 0.1 randomCrossover (bugGenome a) >>=
    withProbability 0.01 randomCutAndSplice >>=
    withProbability 0.001 mutatePairedLists >>=
    randomOneOfPair
  build name = runDiploidReader (buildBug False name)

run :: [Bug] -> StateT (SimpleUniverse Bug) IO [Bug]
run (me:other:_) = do
  writeToLog $ agentId me ++ "'s turn"
  if bugSex me == Female && bugSex other == Male
    then do
      name <- genName
      (Right baby) <- liftIO $ evalRandIO (makeOffspring me other name)
      writeToLog $
        bugName me ++ " and " ++ bugName other ++
          " gave birth to " ++ name ++ ", a " ++
          show (bugColour baby) ++ " " ++ show (bugSex baby) ++ " bug"
      writeToLog $ "Mother: " ++ show me
      writeToLog $ "Father: " ++ show other
      writeToLog $ "Baby: " ++ show baby
      return [deductMatingEnergy me, deductMatingEnergy other, baby]
    else return []
run _ = return [] -- need two agents to mate

deductMatingEnergy :: Bug -> Bug
deductMatingEnergy bug = bug {bugEnergy=bugEnergy bug - 1}
```

9.2 Generate an initial population

We will create the next population in a directory called `Chapter9`. In `main`, we initialise the `chapter9` directory, create two infinite gene sequences, use those sequences to create some Bugs, and write them to the directory. In the previous example, the bugs in the initial population each had two identical sequences of genetic material. This time, however, each bug is created from two random sequences, which will usually differ. The complete code listing is below.

```
import Tutorial.Chapter9.Bug (Bug, buildBug)
import ALife.Creatur.Universe (store, mkSimpleUniverse)
import ALife.Creatur.Genetics.BRGCBool (DiploidReader,
  runDiploidReader)
import Control.Monad.State.Lazy (evalStateT)
import Data.Either (rights)
import System.Random (randoms, getStdGen, newStdGen)

buildBugs :: [String] -> DiploidReader [Bug]
buildBugs names = do
  bugs <- mapM (buildBug True) names
  return $ rights bugs

main :: IO ()
main = do
  let u = mkSimpleUniverse "Chapter9" "chapter9"

  -- Create some Bugs and save them in the population directory.
  let names = ["Bugsy", "Mel", "Flo", "Buzz"]

  r1 <- newStdGen -- source of random genes
```

```
  r2 <- getStdGen -- source of random genes

  let g1 = randoms r1
  let g2 = randoms r2

  let agents = runDiploidReader (buildBugs names) (g1, g2)
  mapM_ (\b -> evalStateT (store b) u) agents
```

9.3 Configure a daemon

The program below configures and launches the daemon.

```
module Main where

import Tutorial.Chapter9.Bug (run)
import ALife.Creatur.Daemon (CreaturDaemon(..), Job(..),
  simpleDaemon, launch)
import ALife.Creatur.Universe (mkSimpleUniverse)
import ALife.Creatur.Task (simpleJob, runInteractingAgents,
  doNothing)
import System.Directory (canonicalizePath)

main :: IO ()
main = do
  dir <- canonicalizePath "chapter9" -- Required for daemon
  let u = mkSimpleUniverse "Chapter9" dir
  let j = simpleJob
        { task=runInteractingAgents run doNothing doNothing,
          sleepTime=0 }
  launch $ CreaturDaemon (simpleDaemon j u) j
```

9.4 Build and run the example

1. If you haven't already done so, follow the instructions in Section 2.1.
2. Create the initial population by running `chapter9-init`.
3. Start/stop/restart the daemon with the command `sudo chapter9-daemon start|stop|restart`. (Stopping the daemon may take a few seconds.)

Log messages are sent to `chapter9/log/Chapter10.log`.

9.5 Key points

- To create a species for which the initial population can be randomly generated, add a parameter to the "build" method that controls whether the gene sequence is truncated.
- When creating the initial population, use truncation. Thereafter, turn truncation off.

Chapter 10

Working with multiple species

In this part of the tutorial, we create a universe with multiple species, where each species is a different Haskell type. (They will be wrapped up in a single type via different constructors.)

10.1 Create a species

We'll have a Martian landscape with rocks, plants, and bugs.

```
data Martian = FromRock Rock | FromPlant Plant | FromBug Bug
  deriving (Show, Generic)
```

All agents will use the same `run` function, which prints a log message and then gives the agent an opportunity to mate.

```
run :: [Martian] -> StateT (SimpleUniverse Martian) IO [Martian]
run xs@(me:_) = do
  writeToLog $ agentId me ++ "'s turn"
  tryMating xs
run [] = error "empty agent list"
```

If the first two agents on the list are the same species, and aren't rocks, then we call that agent's custom implementation of `tryMating`.

```
tryMating :: [Martian] -> StateT (SimpleUniverse Martian) IO [Martian]
tryMating (FromPlant me:FromPlant other:_) = do
    xs <- P.tryMating [me, other]
    return $ map FromPlant xs
tryMating (FromBug me:FromBug other:_) = do
    xs <- B.tryMating [me, other]
    return $ map FromBug xs
tryMating xs = return xs -- can't mate rocks or mismatched species
```

The complete code listing is below.

```
{-# LANGUAGE DeriveGeneric #-}
module Tutorial.Chapter10.Martian (Martian(..), run) where

import Tutorial.Chapter10.Rock (Rock)
import Tutorial.Chapter10.Plant (Plant)
import qualified Tutorial.Chapter10.Plant as P (tryMating)
import Tutorial.Chapter10.Bug (Bug)
import qualified Tutorial.Chapter10.Bug as B (tryMating)
import ALife.Creatur (Agent, agentId, isAlive)
import ALife.Creatur.Database (Record, key)
import ALife.Creatur.Universe (SimpleUniverse, writeToLog)
import Control.Monad.State (StateT)
import Data.Serialize (Serialize)
import GHC.Generics (Generic)

data Martian = FromRock Rock | FromPlant Plant | FromBug Bug
```

```haskell
  deriving (Show, Generic)

instance Serialize Martian

instance Agent Martian where
  agentId (FromRock x) = agentId x
  agentId (FromPlant x) = agentId x
  agentId (FromBug x) = agentId x
  isAlive (FromRock x) = isAlive x
  isAlive (FromPlant x) = isAlive x
  isAlive (FromBug x) = isAlive x

instance Record Martian where
  key = agentId

run :: [Martian] -> StateT (SimpleUniverse Martian) IO [Martian]
run xs@(me:_) = do
  writeToLog $ agentId me ++ "'s turn"
  tryMating xs
run [] = error "empty agent list"

tryMating :: [Martian] -> StateT (SimpleUniverse Martian) IO [Martian]
tryMating (FromPlant me:FromPlant other:_) = do
    xs <- P.tryMating [me, other]
    return $ map FromPlant xs
tryMating (FromBug me:FromBug other:_) = do
    xs <- B.tryMating [me, other]
    return $ map FromBug xs
tryMating xs = return xs -- can't mate rocks or mismatched species
```

We can re-use the implementation of `Bug` from Section 8, but without the `run` method. All agents will use the `run` method in `Tutorial.Chapter10.Martian`. The complete code listing is below.

```haskell
{-# LANGUAGE DeriveGeneric #-}

module Tutorial.Chapter10.Rock (Rock(..)) where

import ALife.Creatur (Agent, agentId, isAlive)
import ALife.Creatur.Database (Record, key)
import Data.Serialize (Serialize)
import GHC.Generics (Generic)

data Rock = Rock String Int deriving (Show, Generic)

instance Serialize Rock

instance Agent Rock where
  agentId (Rock name _) = name
  isAlive _ = True

instance Record Rock where key = agentId
```

We can re-use the implementation of `Plant` from Section 7, with a few changes. The code that implements mating has been moved into a custom `tryMating` function. Again, we can drop the `run` method because all agents will use the `run` method in `Tutorial.Chapter10.Martian`. For a little variety, we'll store the genetic information as `[Word8]` instead of `[Bool]`. To make that change, we only need to modify the import to:

```haskell
import ALife.Creatur.Genetics.BRGCWord8 . . .
```

The complete code listing is below.

```haskell
{-# LANGUAGE DeriveGeneric, FlexibleContexts, TypeFamilies #-}
module Tutorial.Chapter10.Plant (Plant(..), FlowerColour(..),
```

```haskell
  buildPlant, tryMating) where

import ALife.Creatur (Agent, agentId, isAlive)
import ALife.Creatur.Database (Record, key)
import ALife.Creatur.Genetics.BRGCWord8 (Genetic, Reader, Sequence,
  getWithDefault, runReader, copy, consumed)
import ALife.Creatur.Genetics.Recombination (mutatePairedLists,
  randomCrossover, randomCutAndSplice, randomOneOfPair, withProbability)
import ALife.Creatur.Genetics.Reproduction.SimplifiedSexual
  (Reproductive, Strand, recombine, build, makeOffspring)
import ALife.Creatur.Universe (SimpleUniverse, genName, writeToLog)
import Control.Monad.IO.Class (liftIO)
import Control.Monad.Random (evalRandIO)
import Control.Monad.State (StateT)
import Data.Serialize (Serialize)
import GHC.Generics (Generic)

data Plant = Plant
  {
    plantName :: String,
    plantFlowerColour :: FlowerColour,
    plantEnergy :: Int,
    plantGenome :: Sequence
  } deriving (Show, Generic)

instance Serialize Plant

instance Agent Plant where
  agentId = plantName
  isAlive plant = plantEnergy plant > 0

instance Record Plant where key = agentId

data FlowerColour = Red | Orange | Yellow | Violet | Blue
  deriving (Show, Eq, Generic, Enum, Bounded)
instance Serialize FlowerColour
instance Genetic FlowerColour

buildPlant :: Bool -> String -> Reader (Either [String] Plant)
buildPlant truncateGenome name = do
  colour <- getWithDefault Red
  g <- if truncateGenome then consumed else copy
  return . Right $ Plant name colour 10 g

instance Reproductive Plant where
  type Strand Plant = Sequence
  recombine a b =
    withProbability 0.1 randomCrossover (plantGenome a, plantGenome b) >>=
    withProbability 0.01 randomCutAndSplice >>=
    withProbability 0.001 mutatePairedLists >>=
    randomOneOfPair
  build name = runReader (buildPlant False name)

tryMating
  :: (Agent a, Serialize a)
    => [Plant] -> StateT (SimpleUniverse a) IO [Plant]
tryMating (me:other:_) = do
  name <- genName
  (Right baby) <- liftIO $ evalRandIO (makeOffspring me other name)
  writeToLog $
    plantName me ++ " and " ++ plantName other ++
      " gave birth to " ++ name ++ ", with " ++
        show (plantFlowerColour baby) ++ " flowers"
```

```
  writeToLog $ "Me: " ++ show me
  writeToLog $ "Mate: " ++ show other
  writeToLog $ "Baby: " ++ show baby
  return [deductMatingEnergy me, deductMatingEnergy other, baby]
tryMating x = return x -- need two agents to mate

deductMatingEnergy :: Plant -> Plant
deductMatingEnergy p = p {plantEnergy=plantEnergy p - 1}
```

We can re-use the implementation of Bug from Section 8, with a few changes. The code that implements mating has been moved into a custom tryMating function. And again we can drop the run method and switch to Word8 encoding for the genetic information. The complete code listing is below.

```
{-# LANGUAGE DeriveGeneric, FlexibleContexts, TypeFamilies #-}
{-# LANGUAGE CPP #-}
module Tutorial.Chapter10.Bug (Bug(..), Sex(..), BugColour(..),
  buildBug, tryMating) where

import ALife.Creatur (Agent, agentId, isAlive)
import ALife.Creatur.Database (Record, key)
import ALife.Creatur.Genetics.BRGCWord8 (Genetic, Sequence,
  DiploidSequence, DiploidReader, getAndExpress, runDiploidReader,
  copy2, consumed2)
import ALife.Creatur.Genetics.Diploid (Diploid)
import ALife.Creatur.Genetics.Recombination (mutatePairedLists,
  randomCrossover, randomCutAndSplice, randomOneOfPair,
  repeatWithProbability, withProbability)
import ALife.Creatur.Genetics.Reproduction.Sexual (Reproductive, Strand,
  produceGamete, build, makeOffspring)
import ALife.Creatur.Universe (SimpleUniverse, genName, writeToLog)
import Control.Monad.IO.Class (liftIO)
import Control.Monad.Random (evalRandIO)
import Control.Monad.State (StateT)
import Data.Serialize (Serialize)
import GHC.Generics (Generic)

#if MIN_VERSION_base(4,8,0)
-- Starting with GHC 7.10 (base 4.8), we don't need to import
-- Control.Applicative
#else
import Control.Applicative
#endif

data Bug = Bug
  {
    bugName :: String,
    bugColour :: BugColour,
    bugSpots :: [BugColour],
    bugSex :: Sex,
    bugEnergy :: Int,
    bugGenome :: DiploidSequence
  } deriving (Show, Generic)

instance Serialize Bug

instance Agent Bug where
  agentId = bugName
  isAlive bug = bugEnergy bug > 0

instance Record Bug where key = agentId

data BugColour = Green | Purple | Red | Brown | Orange | Pink | Blue
```

```haskell
  deriving (Show, Eq, Enum, Bounded, Generic)
instance Serialize BugColour
instance Genetic BugColour
instance Diploid BugColour

data Sex = Male | Female
  deriving (Show, Eq, Enum, Bounded, Generic)
instance Serialize Sex
instance Genetic Sex
instance Diploid Sex

buildBug :: Bool -> String -> DiploidReader (Either [String] Bug)
buildBug truncateGenome name = do
  sex <- getAndExpress
  colour <- getAndExpress
  spots <- getAndExpress
  g <- if truncateGenome then consumed2 else copy2
  return $ Bug name <$> sex <*> colour <*> spots <*> pure 10 <*> pure g

instance Reproductive Bug where
  type Strand Bug = Sequence
  produceGamete a =
    repeatWithProbability 0.1 randomCrossover (bugGenome a) >>=
    withProbability 0.01 randomCutAndSplice >>=
    withProbability 0.001 mutatePairedLists >>=
    randomOneOfPair
  build name = runDiploidReader (buildBug False name)

tryMating
  :: (Agent a, Serialize a)
    => [Bug] -> StateT (SimpleUniverse a) IO [Bug]
tryMating (me:other:_) = do
  writeToLog $ bugName me ++ ", a " ++ show (bugSex me) ++ " bug, sees "
    ++ bugName other ++ ", a " ++ show (bugSex other)
  if bugSex me == Female && bugSex other == Male
    then do
      name <- genName
      (Right baby) <- liftIO $ evalRandIO (makeOffspring me other name)
      writeToLog $
        bugName me ++ " and " ++ bugName other ++
          " gave birth to " ++ name ++ ", a " ++
          show (bugColour baby) ++ " " ++ show (bugSex baby) ++ " bug"
      writeToLog $ "Mother: " ++ show me
      writeToLog $ "Father: " ++ show other
      writeToLog $ "Baby: " ++ show baby
      return [deductMatingEnergy me, deductMatingEnergy other, baby]
    else do
      writeToLog $ bugName me ++ " is not interested in "
        ++ bugName other
      return []
tryMating _ = return [] -- need two agents to mate

deductMatingEnergy :: Bug -> Bug
deductMatingEnergy bug = bug {bugEnergy=bugEnergy bug - 1}
```

10.2 Generate an initial population

We will create the next population in a directory called `Chapter10`. In `main`, we initialise the `chapter10` directory, create some rocks, bugs, and plants, and then write them to the directory. Since we're using `[Word8]` to encode the genome, we need to import `BRGCWord8` instead of `BRGCBool`.

```haskell
import Tutorial.Chapter10.Rock (Rock(..))
import Tutorial.Chapter10.Plant (buildPlant)
import Tutorial.Chapter10.Bug (buildBug)
import Tutorial.Chapter10.Martian (Martian(..))
import ALife.Creatur.Genetics.BRGCWord8 (Reader, DiploidReader,
  runReader, runDiploidReader)
import ALife.Creatur.Universe (store, mkSimpleUniverse)
import Data.Either (rights)
import Control.Monad.State.Lazy (evalStateT)
import System.Random (getStdGen, newStdGen, randoms)

buildPlants :: [String] -> Reader [Martian]
buildPlants names = do
  xs <- mapM (buildPlant True) names
  return . map FromPlant . rights $ xs

buildBugs :: [String] -> DiploidReader [Martian]
buildBugs names = do
  xs <- mapM (buildBug True) names
  return . map FromBug . rights $ xs


main :: IO ()
main = do
  let u = mkSimpleUniverse "Chapter10" "chapter10"

  -- Create some rocks and save them in the population directory.
  let rock1 = FromRock $ Rock "Rocky" 0
  evalStateT (store rock1) u

  let rock2 = FromRock $ Rock "Roxie" 0
  evalStateT (store rock2) u

  -- Create some plants and save them in the population directory.
  let plantNames = ["Rose", "Sunny", "Vi"]

  r <- newStdGen -- source of random genes
  let g = randoms r

  let plants = runReader (buildPlants plantNames) g
  mapM_ (\b -> evalStateT (store b) u) plants

  -- Note: The next part "hangs" for me. It didn't used to. And the
  -- same code works in Chapter 9. I haven't had a chance to debug
  -- the problem yet.

  -- Create some Bugs and save them in the population directory.
  let names = ["Bugsy", "Mel", "Flo", "Buzz"]

  r1 <- newStdGen -- source of random genes
  r2 <- getStdGen -- source of random genes

  let g1 = randoms r1
  let g2 = randoms r2

  let agents = runDiploidReader (buildBugs names) (g1, g2)
  mapM_ (\b -> evalStateT (store b) u) agents
```

10.3 Configure a daemon

The daemon implementation should be familiar by now.

```
module Main where

import Tutorial.Chapter10.Martian (run)
import ALife.Creatur.Daemon (CreaturDaemon(..), Job(..),
  simpleDaemon, launch)
import ALife.Creatur.Universe (mkSimpleUniverse)
import ALife.Creatur.Task (simpleJob, runInteractingAgents,
  doNothing)
import System.Directory (canonicalizePath)

main :: IO ()
main = do
  dir <- canonicalizePath "chapter10" -- Required for daemon
  let u = mkSimpleUniverse "Chapter10" dir
  let j = simpleJob
        { task=runInteractingAgents run doNothing doNothing,
          sleepTime=0 }
  launch $ CreaturDaemon (simpleDaemon j u) j
```

10.4 Build and run the example

1. If you haven't already done so, follow the instructions in Section 2.1.
2. Create the initial population by running `chapter10-init`.
3. Start/stop/restart the daemon with the command `sudo chapter10-daemon start|stop|restart`. (Stopping the daemon may take a few seconds.)

Log messages are sent to `chapter10/log/Chapter10.log`.

10.5 Key points

- To work with multiple species, where each species is a different Haskell type, unify the types with a "container" type (e.g., the `Martian` class).

Chapter 11

Advanced genomes

.

In the previous chapters, we developed agents with very simple genes. However, genes can be arbitrarily complex. The default implementation of `Genetic` is usually sufficient, as in the following example.

```
data ComplexGene = A | B Colour | C Word8 | D Bool Char | E [ComplexGene]
  deriving (Show, Eq, Generic)

instance Genetic ComplexGene
```

You are not restricted to using the default implementation of `Genetic`. In the following example, we write a custom implementation. You can mix custom and default implementations; the implementation of `Genetic` for `CustomGene` uses the default implementation of `Genetic` for `Colour`.

```
data CustomGene = F Colour | G Bool
  deriving (Show, Eq, Generic)

instance Genetic CustomGene where
  put (F c) = putRawWord8 7 >> put c
  put (G b) = putRawWord8 8 >> put b
  get = do
    x <- getRawWord8
    case x of
      (Right 7) -> do
        c <- get :: Reader (Either [String] Colour)
        return . fmap F $ c
      (Right 8) -> do
        b <- get :: Reader (Either [String] Bool)
        return . fmap G $ b
      _       -> return $ Left ["Invalid gene sequence"]
```

One reason you might want to write a custom implementation of `Genetic` is for efficiency. In this example, we store three boolean values in a Word8 value to reduce the amount of storage required.

```
data CustomGene2 = H Bool Bool Bool
  deriving (Show, Eq, Generic)

instance Genetic CustomGene2 where
  put (H x y z) = putRawWord8 (x' + y' + z')
    where x' = (4 *) . fromIntegral . fromEnum $ x :: Word8
          y' = (2 *) . fromIntegral . fromEnum $ y :: Word8
          z' = fromIntegral . fromEnum $ z :: Word8
  get = do
    w <- getRawWord8 :: Reader (Either [String] Word8)
    let x = fmap (flip testBit 2) w :: Either [String] Bool
    let y = fmap (flip testBit 1) w :: Either [String] Bool
    let z = fmap (flip testBit 0) w :: Either [String] Bool
    return $ H <$> x <*> y <*> z
```

You will find these examples in the code listing below. To run the example, type `chapter11`.

```haskell
{-# LANGUAGE DeriveGeneric, FlexibleContexts, FlexibleInstances,
    TypeFamilies #-}
{-# LANGUAGE CPP #-}
import Prelude hiding (read)
import ALife.Creatur.Genetics.BRGCWord8 (Genetic, Reader, put, get,
  putRawWord8, getRawWord8, write, read)
import Data.Bits
import Data.Word (Word8)
import GHC.Generics (Generic)

#if MIN_VERSION_base(4,8,0)
-- Starting with GHC 7.10 (base 4.8), we don't need to import
-- Control.Applicative
#else
import Control.Applicative
#endif

--
-- This example shows how the default implementation of Genetic is
-- usually sufficient, even for a complex data structure.
--
data Colour = Green | Purple
  deriving (Show, Eq, Enum, Bounded, Generic)
instance Genetic Colour

data ComplexGene = A | B Colour | C Word8 | D Bool Char | E [ComplexGene]
  deriving (Show, Eq, Generic)

instance Genetic ComplexGene

--
-- This is an example of a custom implementation of Genetic. This
-- implementation uses the default implementations of Genetic for Colour
-- and Bool, showing that it is possible to mix and match.
--

data CustomGene = F Colour | G Bool
  deriving (Show, Eq, Generic)

instance Genetic CustomGene where
  put (F c) = putRawWord8 7 >> put c
  put (G b) = putRawWord8 8 >> put b
  get = do
    x <- getRawWord8
    case x of
      (Right 7) -> do
        c <- get :: Reader (Either [String] Colour)
        return . fmap F $ c
      (Right 8) -> do
        b <- get :: Reader (Either [String] Bool)
        return . fmap G $ b
      _       -> return $ Left ["Invalid gene sequence"]

--
-- In this example, we store three boolean values in a Word8 value
-- to reduce the amount of storage required.
--

data CustomGene2 = H Bool Bool Bool
  deriving (Show, Eq, Generic)

instance Genetic CustomGene2 where
```

```haskell
  put (H x y z) = putRawWord8 (x' + y' + z')
    where x' = (4 *) . fromIntegral . fromEnum $ x :: Word8
          y' = (2 *) . fromIntegral . fromEnum $ y :: Word8
          z' = fromIntegral . fromEnum $ z :: Word8
  get = do
    w <- getRawWord8 :: Reader (Either [String] Word8)
    let x = fmap (flip testBit 2) w :: Either [String] Bool
    let y = fmap (flip testBit 1) w :: Either [String] Bool
    let z = fmap (flip testBit 0) w :: Either [String] Bool
    return $ H <$> x <*> y <*> z

test :: (Eq x, Show x, Genetic x) => x -> IO ()
test x = do
  putStrLn $ "wrote gene: " ++ show x
  let dna = write x
  putStrLn $ "dna=" ++ show dna
  let x2 = read dna
  putStrLn $ "read: " ++ show x2
  if x2 == Right x
     then putStrLn "SUCCESS"
     else putStrLn "FAILURE"

main :: IO ()
main = do
  putStrLn "Example of complex gene"
  test (E [ A, B Purple, C 7, D True 'a', E []])

  putStrLn "\nExample of a custom implementation of Genetic"
  test (F Green)

  putStrLn "\nAnother example of a custom implementation of Genetic"
  test (H True False True)
```

Chapter 12

FAQ

Frequently Anticipated Questions

Q: What is the structure of the universe directory?

A: See the example below.

```
chapter9
chapter9/db
chapter9/db/Flo             -- contains the agent named ``Flo''
chapter9/db/Bugsy           -- contains the agent named ``Bugsy''
chapter9/db/Chapter9_16     -- contains the agent named ``Chapter9_16''
. . .
chapter9/namer              -- contains a counter for generating names
chapter9/log
chapter9/log/Chapter9.log   -- the current log file
                            -- there may be older log files as well
chapter9/log/Chapter9.exp   -- contains a counter for log rotation
chapter9/clock              -- contains the current ``universe'' time
```

Q: How can I write tools (apart from the daemon) that operate on the agents.

A: See `Tutorial/Chapter9/Examine.hs` for an example of a simple analysis tool. That program uses `agentIds` and `getAgent` to read the agents from the population, without modifying them. You can also work with agents that have "died" and been archived, using `archivedAgentIds` and `getAgentFromArchive`.

Alternatively, you can read the agent files directly, as shown below. Be careful not to modify a file while the daemon is running.

```
ghci> :l Tutorial.Chapter9.Bug
[1 of 1] Compiling Tutorial.Chapter9.Bug ( src/Tutorial/Chapter9/Bug.hs, interpreted )
Ok, modules loaded: Tutorial.Chapter9.Bug.
ghci> import Data.Serialize
ghci> import Data.ByteString
ghci> x <- BS.readFile "chapter9/db/Bugsy"
ghci> let b = decode x :: Either String Bug
ghci> b
Right (Bug {bugName = "Bugsy", bugColour = Green, bugSpots = [], bugSex = Female,
bugEnergy = 6, bugGenome = ([False,True,False,True,False],[True,True,True,False])})
```

Q: What causes this (or a similar) error message?

```
No instance for (ALife.Creatur.Genetics.Code.BRGCWord8.GGene
                   (GHC.Generics.Rep ClassifierGene))
  arising from a use of `ALife.Creatur.Genetics.Code.BRGCWord8.$gdmput'
```

A: Did you remember to add `deriving Generic` to your gene type?

Chapter 13

TO DO

Some things I'd like to do to enhance this tutorial...

1. Explain that if you don't want the child to be immediately mature and able to interact with other agents and mate, you could keep it as a field in the mother's implementation until it's mature. Include an example.

2. Show how to collect statistics.

3. Show how to keep agents and logs in a database rather than as separate files. I'm thinking of providing an ”agent interface” to MongoDB and any ODBC-compliant DB.

4. Show how to get a list of agents meeting certain criteria (e.g. all agents within a certain distance of a particular agent). This will require a different DB.

5. Show how to add other state data to the universe (in addition to the clock, logger, agent namer, and database).