ESL Reference Guide

Tony Clark

Aston University, Birmingham, UK

Abstract

ESL is an actor-based functional language that aims to provide abstractions over applications that require a high degree of concurrency. ESL is statically typed which means that programs can be verified for type safety before they run. ESL provides extensions over the standard Actor Model, including shared state and access to the underlying Java platform, for practical application development. ESL compiles to pure Java and runs against a small Java library. The ESL compiler is written in ESL. ESL is supported by a development environment called EDB that includes libraries for data visualisation in terms of HTML, SVG and GraphViz. This report provides a reference for ESL and EDB using examples.

Contents

1. Introduction

This document describes the language ESL and its implementation. ESL is an actor-based language that has been designed to offer a convenient way to build applications that benefit from a large number of independent concurrent processes that communicate in terms of structured data. ESL compiles directly to Java source code and can therefore easily integrate with Java applications. The benefit of using ESL compared to Java is the abstraction that it provides over the construction and management of processes and the data that is passed between them. In addition, ESL enforces a separation of concerns since communication is by asynchronous message which reduces the risks associated with large scale concurrency due to race conditions and shared data. Having said that, whilst ESL actors are encapsulated, it is unreasonable to build a large scale system without some degree of shared data, so ESL provides mechanisms for data sharing where needed. ESL aims to support system verification through an expressive statically checked type system.

The Actor Model of Computation is characterised in figure 1 which shows three actors. Each actor has a local thread of control which selects a message from the head of the mailbox and processes it. When the current message is processed, the thread inspects the mailbox for the next message; if no message is available then the actor becomes idle until a message is received. Each actor has a local state that can be inspected and updated by the methods that handle the messages.

An actor can send messages to any actor that it knows about. An actor will know about an actor if it created the actor or it received the actor as part of a message. A message may contain basic data items, collections of data item, actors, and functions. When a source actor passes a function to a target actor, the source may provide access to its internal state when the target calls the function on its own thread.

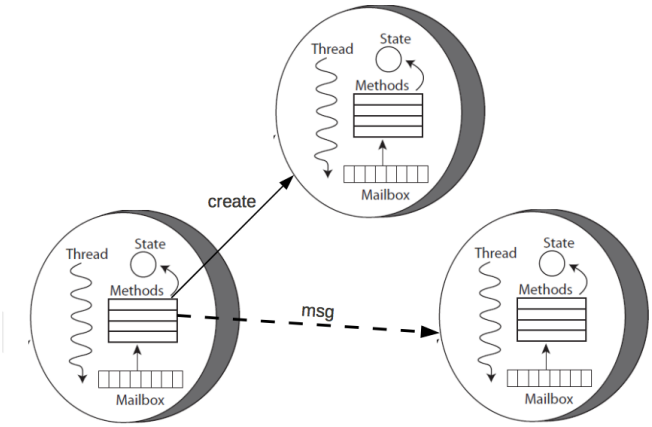


Figure 1: Actor Model of Computation [1]

2. Data and Types

ESL provides basic data types: integers, floats, booleans and strings. For example the following listing declares identifiers along with their values. Comments follow the Java format:

```
1 /* The parameters of the program are defined below.
2    Change the starting position to get a different
3    configuration. */
4
5 x::Int          = 100;     // Starting x position.
6 y::Int          = 200;     // Starting y position.
7 pi::Float       = 3.14;    // Used for position calculation.
8 isActive::Bool  = true;    // True when moving.
9 name::Str       = 'Wilma'; // Used as a unique identifier.
```

Characters such as `#a` and `#+` are represented as integers ascii codes and are therefore of type `Int`. Special characters are introduced using `#\` followed by their name: `#\space`, `#\newline` and `#\backslash`.

ESL supports homogeneous arrays, for example the following is a one-dimensional array of strings with 100 elements indexed from 0:

```
1 names::Array[Str] = new Array[Str](100);
2 names[0] := 'Wilma';
3 print[Str]('value at 0 = ' + names[0]);
```

Arrays may be nested, for example to create a two-dimensional array of booleans:

```
1 points::Array[Array[Bool]] = new Array[Array[Bool]](100);
2 for x::Int in 0..100 do {
3   points[x] := new Array[Bool](200);
4   for y::Int in 0..200 do {
5     points[x][y] := true;
6   }
7 }
```

Note that array elements can be updated and passed between actors, therefore they are a basis for race conditions and other concurrency problems. The intention is that arrays are used locally within an actor as efficient indexable storage. If arrays are shared then access should be protected using locks.

ESL provides homogeneous lists whose elements cannot be modified by side effect and are therefore safe to pass between actors. A list is constructed using `::` and `[ ... ]` and can be decomposed using the operations `head` and `tail` in addition to pattern matching (see below). For example:

```
1 nums::[Int] = [1,2,3,4,5,6,7,8,9,0];
2 print[Int](head[Int](nums));
3 print[[Int](tail[Int](nums))
4 print[[Int]](head[Int](nums) : tail[Int](tail[Int](nums));
```

which produces `1`, `[2,3,4,5,6,7,8,9,0]` and `[1,3,4,5,6,7,8,9,0]` respectively. Note that the operations `head` and `tail` are supplied with the type of the list element; this is because there operations are *polymorphic* as described in section 13. The definition of `nums` above is equivalent to:

```
1 nums::[Int] = 1:2:3:4:5:6:7:8:9:0:[];
```

Sets and bags are like lists except the elements are not ordered. Adding an element to a set or bag produces a new set or bag respectively. Adding an element to a set has no effect if the set already contains the element, otherwise both the set and bag are extended:

```
1 setOfNames::Set{Str} = Set{'Fred','Wilma'};
2 bagOfNames::Bag{Str} = Bag{'Fred','Wilma'};
3 print[Set{Str}](Set{'Fred' | setOfNames});
4 print[Bag{Str}](Bag{'Fred' | setOfNames});
```

produces `Set{'Fred','Wilma'}` and `Bag{'Fred','Wilma','Fred'}` respectively. Note that extraction from sets and bags occurs via pattern matching is described below. Sets require a definition of equality for ESL values which is provided in figure 2.

ESL hash tables map keys to values. A hash table provides operations for adding a mapping, accessing the value for a key, testing whether a mapping exists. A hash table uses field reference to get the list of keys and the list of values:

```
1 ages::Hash[Str,Int] = new Hash[Str,Int];
2 ages.put('Fred',35);
3 ages.put('Wilma',32);
4 print[Bool](ages.hasKey('Fred'));
5 print[Int](ages.get('Wilma'));
6 print[[Str]](ages.keys);
7 print[[Int]](ages.vals);
```

prints `true`, `32`, `[Fred,Wilma]`, and `[35,32]`. Note that because hash tables are updated by side effect, they are not thread safe and should be used locally within an actor, or access should be protected using locks.

ESL functions are first-class values and may be passed between actors within data structures. The rules for identifier binding within functions are defined in section 8. A simple example of a function is:

```
1 add1::(Int)→Int = fun(x::Int)::Int x + x;
2 print[Int](add1(100));
```

which prints `101`.

Records map names to values. Unlike hash tables, the domain of a record is fixed. New type names can be defined that refer to existing types, so the following code implements an two-dimensional point object:

Type	Equality for Values of Type
i1,i2::Int	i1 and i2 are the same number
f1,f2::Float	f1 and f2 are the same number
s1,s2::Str	s1 and s2 have the same characters in order
b1,b2::Bool	b1 and b2 are the same boolean values
h1:l1,h2:l2::[T]	h1 is equal to h2, and l1 is equal to l2
[]::[T]	equal to an empty list of the same type.
set{e1::T | s1::Set{T}} set{e2::T | s2::Set{T}}	e1 and e2 are equal, and s1 and s2 are equal
set{}::Set{T}	is only equal to an empty set of the same type.
bag{e1::T | b1::Bag{T}} bag{e2::T | b2::Bag{T}}	e1 and e2 are equal, and b1 and b2 are equal
bag{}::Bag{T}	is only equal to an empty bag of the same type.
a1,a2::Array[T]	a1 and a2 are the same array
t1,t2::Hash[K,V]	t1 and t2 are the same table
a1,a2::Act {...}	a1 and a2 are the same actor
f1,f2::(T ...) → T	f1 and f2 are the same function
r1,r2::{ n→T ... }	r1.n is equal to r2.n for all names
C(n1,...), C(m2,...)	equal when corresponding ni and mi are equal
null[T]	only equal to the value null

Figure 2: Data Equality

```
type Point = { x::Int; y::Int }
mkPoint(n::Int,m::Int)::Point = {x→n; y→m}
print[Point](mkPoint(100,200));
print[Int](mkPoint(100,200).x);
```

prints {x=100,y=200} and 100. Records cannot be updated by side effect.

The value null can be used as an undefined value for any data type. It is polymorphic and is followed by the type for which it acts as undefined. For example:

```
x::Int = null[Int];
p::Point = null[Point];
```

The following operations are builtin to ESL:

```
_ + _        :: (Int,Int)        → Int      // addition
_ + _        :: (Float,Float)    → Float    // addition
_ + _        :: (Str,T)          → Str      // concatenation
_ + _        :: (T,Str)          → Str      // concatenation
_ + _        :: ([T],[T])        → [T]      // concatenation
_ + _        :: (Set{T},Set{T})  → Set{T}   // union
_ - _        :: (Int,Int)        → Int      // subtraction
_ - _        :: (Float,Float)    → Float    // subtraction
_ * _        :: (Int,Int)        → Int      // multiplication
_ * _        :: (Float,Float)    → Float    // multiplication
_ / _        :: (Int,Int)        → Int      // division
_ / _        :: (Float,Float)    → Float    // division
_ % _        :: (Int,Int)        → Int      // modulo
_ = _        :: (T,T)            → Bool     // equal to
_ <> _       :: (T,T)            → Bool     // not equal to
_ and _      :: (Bool,Bool)      → Bool     // boolean and
_ or _       :: (Bool,Bool)      → Bool     // boolean or
not          :: (Bool)           → Bool     // boolean not
_ .. _       :: (Int,Int)        → [Int]    // integer range
_ @ _        :: (T,T)            → T        // parallel
print[T]     :: (T)              → Void     // print
random       :: (Int)            → Int      // random value
wait         :: (Int)            → Void     // thread delay
isqrt        :: (Int).           → Float    // square root
intToFloat   :: (Int)            → Float    // conversion
round        :: (Float)          → Int      // conversion
stopAll      :: ()               → Void     // halt execution
```

When an ESL identifier is defined its type must be declared. Types are defined as follows:

Type	Description
Int	integers
Float	floats
Bool	booleans
Str	strings
Void	nothing
[T]	lists
(T,...) → T	functions
Array[T]	arrays
Set{T}	sets
Bag{T}	bags
Hash[T,T]	hash tables
{i:T;...}	records
Forall[V,...] T	polymorphic values
rec V. T	recursive types
union { C(T,...);... }	union types

A type may be named and then the name can be used as a shorthand. For example:

```
type Ints    = [Int];
type AddInts = (Ints,Ints) → Ints;

addInts::AddInts = fun(l1::Ints,l2::Ints)::Ints
  case l1,l2 {
    [],l → [];
    l,[] → [];
    x:l1,y:l2 → (x+y):addInts(l1,l2);
  }
```

A function binding can be defined in-line so that the example above can be expressed as follows:

```
addInts(l1::Ints,l2::Ints)::Ints =
  case l1,l2 {
    [],l → [];
    l,[] → [];
```

```
    x:l1,y:l2 → (x+y):addInts(l1,l2);
  }
```

Types can be recursive, for example the following implements two dimensional points with function updates:

```
type Point2D = {
  getX::()→Int;
  getY::()→Int;
  setX::(Int)→Point2D;
  setY::(Int)→Point2D
};

mkPoint(x::Int,y::Int)::Point2D = {
  getX → fun()::Int x;
  getY → fun()::Int y;
  setX → fun(x::Int)::Point2D mkPoint(x,y);
  setY → fun(y::Int)::Point2D mkPoint(x,y)
};

let p::Point2D = mkPoint(100,200);
in {
  print[Int](p.getX());
  print[Int](p.setX(p.getX()+1).getX());
}
```

which prints `100` then `101`.

3. Blocks and Commands

The type `Void`, like Java, represents an evaluation that does not return a result. The operation `print[T](t)` takes a value `t::T` and 'returns' `Void`. A *block* is a sequence of *commands* enclosed in curly brackets. A command is just any expression, but syntactically, if the expression does not end with a closing `}`, then the command must be terminated with a semicolon. The return value of a block is the return value of the last command. For example:

```
print3(a::Int,b::Int,c::Int)::Void = {
  print[Int]('first = ' + a);
  print[Int]('second = ' + b);
  print[Int]('third = ' + b);
}
```

The special case of the block `{}` is of type `Void`.

4. List Comprehensions and List Operations

Lists are a very useful way of organising collections of elements. They can easily be processed using recursion because of their structure: a list is either empty `[]` or is a list `l` with a head element `e` 'consed' to the tail: `e:l`.

A *list comprehension* is an expression that transforms lists. It has the form: `[ e | q ... ]` where `e` is an expression and `q` is a *qualifier*. The idea is that a list is constructed whose elements are formed by evaluating `e` after performing each of the qualifiers in turn. There are two types of qualifier: *binding* and *predicate*. A binding qualifier has the form `p ← e` where `p` is a pattern and `e` is an expression whose value is a list. When it is evaluated, a binding qualifier repeatedly matches the elements from the list against the pattern. A predicate qualifier has the form `?e` where `e` is a boolean valued expression. When it is evaluated, if the expression in a predicate qualifier produces `true` then execution continues, otherwise it ignores the element most recently selected by a binding qualifier and selects the next element. For example:

```
[ n | n ← 0..100, ?(n % 2 = 0) ]
```

produces a list of even numbers between `0` and `99` inclusive. Multiple binding qualifiers may be used:

```
[ mkPoint(x,y) | x ← 0..100, y ← 0..200, ?(x <> y) ]
```

Lists can be processed using recursive functions and pattern matching. Typically a case-expression is used to match against a base case (often `[]`) and a recursion case (often `h:t`). For example the following operation adds up the distances of all points to the origin:

```
addDists(ps::[Point])::Float =
  case ps {
    [] → 0.0;
    h::t → distanceToOrigin(h) + addDists(t);
  }
```

The following defines Quicksort in ESL:

```
qsort(l::[Int])::[Int] =
  case l {
    [] →[];
    x:l →
      qsort([ n | n ← l, ?(n<x) ])
      + [x] +
      qsort([ n | n ← l, ?(n>x) ]);
  }
```

The collection of list operations that are supported by ESL are shown in C.

5. Algebraic Data Types and Pattern Matching

ESL supports algebraic data types, also known as *union types*. For example, the directions of the compass can be represented using different integers or strings, but it is better to define a new type `Direction` and to have 4 unique values of that type. This can be defined as follows:

```
data Direction =
  North
| South
| East
| West;
```

Now any identifier defined to have type `Direction` can hold one of the 4 values and the special value `null[Direction]`:

```
d::Direction = North;
```

Pattern matching can be used with such a data type:

```
moveRight(dir::Direction)::Direction =
  case dir {
    North → East;
    East  → South;
    South → West;
    West  → North;
  }
```

Given a list of such directions, they can be mapped:

```
dirs::[Direction] = [ North, South, East, West ];
[ moveRight(d) | d ← dirs ]
```

The element types of a union can have components. For example a tree of integers:

```
data TreeOfInt =
  Branch(TreeOfInt,TreeOfInt)
| Leaf(Int);
```

Note that a branch term contains two integer trees and a leaf contains an integer. The integers in the tree can be added together:

```
addTreeOfInt(t::TreeOfInt)::Int =
  case t {
    Branch(l,r) → addTreeOfInt(l) + addTreeOfInt(r);
    Leaf(n)     → n;
  }
print[Int](addTreeOfInt(Branch(Leaf(100),Leaf(200))));
```

Prints `300`. When pattern matching, the case arms are tried in turn, so we can define a short-cut version:

```
addTreeOfInt(t::TreeOfInt)::Int =
  case t {
    Branch(Leaf(0),t) → addTreeOfInt(t);
    Branch(t,Leaf(0)) → addTreeOfInt(t);
    Branch(l,r)       → addTreeOfInt(l) + addTreeOfInt(r);
    Leaf(n)           → n;
  }
```

Case arms may contain conditions. For example, adding up all those elements of a tree that satisfy a predicate:

```
addIf(p::(Int)→Bool,t::TreeOfInt)::Int =
  case t {
    Branch(l,r)       → addIf(p,l) + addIf(p,r);
    Leaf(n) when p(n) → n;
    Leaf(n)           → 0;
  }
addIf(fun(n::Int)::Bool n > 100,Branch(Leaf(100),Leaf(200)));
```

Prints `200`.

6. Sets and Collection Matching

ESL supports homogenous sets: the empty set `set{}`; a set constructed from an element `e` and a set `s`, `set{e | s}`; a set union `s1 + s2`. Both sets and lists can be processed using pattern matching. For example:

```
setContainsInt(s::Set{Int},x::Int)::Bool =
  case s {
    set{y | z} when x = y → true;
    s                     → false;
  }
```

The operation `setContainsInt` uses pattern matching to select the element `y` which is equal to the argument `x`. Note that sets are unordered, so the pattern `set{y | z}` is *non-deterministic*. When the supplied value `s` matches the pattern, one of the elements in the set is matched against `y`; if the pattern fails to match before the corresponding → then a different element `y` will be chosen. In the example, the condition `x = y` is within the scope fo the choice, this causes elements from the set `s` to be tried un turn until one that matches `x` is found or the set is exhausted. If all the values have been tried from the set `s` then the case-arm fails and the next case-arm is tried.

Non-deterministic choice may occur more than once in the same pattern. For example, the following returns a duplicate entry in two sets:

```
duplicate(s1::Set{Int},s2::Set{Int},notFound::Int)::Int =
  case s1,s2 {
    set{x | p}, set{y | q} when x = y → x;
    s1,s2                             → notFound;
  }
```

The first case-arm in `duplicate` selects an element `x` from `s1`, then selects `y` from `s2` such that they are equal elements. If there is no element in common then the `notFound` element is returned.

Non-deterministic pattern-based choice can be used over lists using the pattern `p1 + [p2] + p3` where `p1` matches a prefix of a list, `p2` matches some element, and `p3` matches a suffix. The pattern fails in the case that the list if empty. For example, testing whether a given element occurs in a list:

```
intMember(n::Int,l::[Int])::Bool =
  case l {
    l1 + [x] + l2 when x = n → true;
    l                        → false;
  }
```

7. Actors

Actors are implemented in terms of *behaviour types* and *behaviours*. A behaviour type provides an interface definition for a set of behaviours. An interface consists of message types and exported identifiers and their types. Consider a simple behaviour type called `Sink` that accepts a single type of message `M(Int)`:

```
Act Sink {
  M(Int);
}
```

Any behaviour of type `Sink` must implement the message `M`:

```
act sink::Sink {
  M(n::Int) → print[Str]('got: ' + n);
}
```

There can be any number of behaviours with the same type:

```
act sink2::Sink {
  M(n::Int) → {} // Ignore the message.
}
```

An actor is created with a specific behaviour:

```
s::Sink = new sink;
```

Once created, the actor can receive messages. A message is asynchronous: `s ← M(100)` and is placed on the input queue of the target actor. Each actor is processing at most one message at any time, and is idle if there are no messages on its queue and it is not currently handling a message. Since all actors run concurrently, there is no guarantee that of message ordering other

than messages sent from the same actor will retain the ordering in which they are sent.

Any number of behaviours can be defined to conform to a behaviour type. The following shows three different implementations of the type `A`. The first, `a`, just loops indefinitely which an actor can do because all message handling occurs concurrently with other actors. The second, `forward` is supplied with a second actor of type `A` to which it forwards messages. Finally, `broadcast` is supplied with a collection of actors of type `A` and will send any message to all of them:

```
Act A {
  M(Int,Bool);
}

act a::A {
  M(n::Int,b::Bool) →
   self ← M(n,not(b));
}

actor::A = new a;

act forward(other::A)::A {
  M(n::Int,b::Bool) →
    other ← M(n,b);
}

proxy::A = new forward(actor);

act broadcast(as::[A])::A {
  toAll(l::[A],n::Int,b::Bool)::Void =
    case l {
      [] → {};
      h::A:t::[A] → {
        h ← M(n,b);
        toAll(t,n,b);
      }
    }
  M(n::Int,b::Bool) →
    toAll(as,n,b);
  }

spray::A = new broadcast([new forward(new a) | i ← 0.1000])
    ;
```

Often we want to do some initialisation when an actor is created. The special message handler → is run each time a new actor is created before any messages are processed:

```
act sink3::Sink {
  → print[Str]('I am initialising');
  M(n::Int) → print[Str]('got: ' + n);
}
```

Actors can refer to other actors. The following pair of actors will bounce messages back and forth until the counter runs down:

```
Act Ping {
  Ping(Int,Pong);
}

Act Pong {
  Pong(Int,Ping);
}

act ping::Ping {
  Ping(0,p::Pong)       → print[Str]('stop');
  Ping(n::Int,p::Pong)  →
    p ← Pong(n-1,self);
}

act pong::Pong {
  Pong(0,p::Ping)       → print[Str]('stop');
  Pong(n::Int,p::Ping)  →
    p ← Ping(n-1,self);
}

p1::Ping = new ping;
p2::Pong = new pong;
p1 ← ping(100,p2);
```

Behaviour definitions can be parameterised so that new actors can initialise the behaviour. For example, in order to provide an identifier to each actor. In the following example, we create a collection of machines and then broadcast jobs. A machine can process a job if the identifier matches:

```
Act Machine {
  Process(Int);
}

act machine(id::Int)::Machine {
  Process(m::Int) when m = id → {
    print[Str]('handling job');
  }
  Process(m::Int) → {} // Ignore job if not target machine.
}

noOfMs::Int          = 100;
noOfJs::Int          = 10000;
machines::[Machine] = [ new machine(n) | n ← 0..noOfMs ];

for job::Int in 0..noOfJs do {
  let targetMachine::Int = random(noOfMs);
  in {
    for m::Machine in machines do {
      m ← Perform(targetMachine);
    }
  }
}
```

An actor may perform `become` to change behaviour. The replacement behaviour must implement the same behaviour type. Subsequent messages are handled by the new behaviour. For example, a resource manager allocates resource on request and becomes pending resource manager for the period of time when resources are unavialable:

```
type Resource = ...;
Act Consumer {
  Receive(Resource);
}

Act ResourceManager {
  Allocate(Consumer);
  Free(Resource);
}

act manager(rs::[Resource])::ResourceManager {
  Allocate(c::Consumer) when rs = [] → {
    become pending;
    self ← Allocate(c);
  }
  Allocate(c::Consumer) → {
    become manager(tail[Resource](rs));
    c ← Receive(head[Resource](rs);
```

```
  }
  Free(r::Resource) → become manager(r:rs);
}

act pending::ResourceManager {
  Allocate(c::Customer) → self ← Allocate(c);
  Free(r::Resource) → become manager([r]);
}
```

8. Binding and Scope

ESL is a statically scoped language meaning that identifiers can be referenced within the textual binding block in which they are defined. Identifiers can be introduced as: function arguments; let-bindings; letrec-bindings; behaviour definitions; behaviour arguments (equivalent to function arguments); top-level definitions; pattern variables; for-loop controls. This section addresses each of these categories in turn.

Let-expressions introduce local bindings for the scope of the body of the let. A let-expression may have multiple bindings in which case they are performed in parallel:

```
let dx::Int = x1-x2;
    dy::Int = y1-y2;
in isqrt((dx*dx)+(dy*dy));
```

A let-binding can be used to define a local function. Special syntax allows a function to be defined without using the keyword `fun`:

```
let
  distance(x1::Int,y1::Int,x2::Int,y2::Int)::Float =
    let dx::Int = x1-x2;
        dy::Int = y1-y2;
    in isqrt((dx*dx)+(dy*dy));
in distance(100,200,50,70);
```

A letrec-expression can be used to create local recursive functions:

```
letrec
  contains(l::[Int],n::Int)::Bool =
    case l {
      [] → false;
      h:t when h=n → true;
      h:t → contains(t,n);
    }
in contains([1,2,3,4,5,6,7,8,9],5);
```

A letrec-expression can be used to create mutually recursive function definitions:

```
letrec
  isEvan(n::Int)::Bool = if n = 0 then true else isOdd(n-1);
  isOdd(n::Int)::Bool = if n = 0 then false else isEven(n-1);
in isEven(101);
```

Letrec-bindings that do not establish functions are performed in sequence.

Behaviour definitions contain a collection of bindings that are established as a letrec and scoped over the message handlers of the behaviour. In addition, the names of local bindings can be exported and referenced using the `_._` operation. An an example, support we have a post office actor that delivers letters to people by matching up the address in the message:

```
Act Person {
  export getAddress::()→Str;
}

act person(address::Str)::Person {
  export getAddress;
  getAddress()::Str = address;
  Open(Letter);
}

data Letter = Letter(Str,Str);

Act PostOffice {
  Deliver(Str);
}

act postOffice(people::[Person])::PostOffice {
  findPerson(address::Str,people::[Person])::Person =
    case people {
      [] → throw[Person]('cannot deliver to ' + address);
      p:ps when p.getAddress() = address → p;
      p:ps → findPerson(address,people);
    }
  Deliver(Letter(name,address)) →
    findPerson(address) ← Open(Letter(name,address));
}
```

9. Modules

labelsec:modules

ESL is a compiled language. The units of compilation are called modules and are usually contained in text files. A module contains a collection of mutually recursive bindings for types, behaviours, functions and values. A module `m` may export some of its bindings so that they can be used by any other module that imports `m`. For example a module defined in a file `a/b/c.esl` defines two names and exports one of them:

```
export f;

f(n::Int)::Int = g(n) + 1;
g(n::Int)::Int = n + 100;
```

A second module can then import `c` and use the exported name:

```
export main;

import 'a/b/c.esl';

g(n::Int)::Int = n = 200;

act Main::Act{} {
  → print[Int](f(10));
}
```

prints 111. Note that the second module above defines a behaviour names `main`. When building an application there should be a root module that defines `main` which is the entry point for the application. When ESL starts to run a module it creates a single actor with the behaviour `main`. There is no restriction on the behaviour type of `main`.

10. Time

ESL applications can be driven by a system clock. This can be useful when the application is driven by click ticks or when the application should run for a specific length of time. In order to receive clock ticks, a behaviour must implement at least one message handler of type `Time(Int)`. Any actor with such a behaviour will receive clock ticks *when the actor is idle*. If an actor does not define a handler for `ntTimeI` then there is no overhead for handling time.

The value of `n` in `Time(n)` is the time in milliseconds since the start of the application. The frequency of the ticks is undefined and therefore there is no guarantee that an actor will receive a tick with a specific value of `n`, and it may be the case the an actor receives multiple messages with the same value of `n`.

An actor may call `wait(n)` which causes the actor's thread to wait for `n` milliseconds. Since all actors have concurrent behaviours, a call of `wait` will not affect any other actor.

The following definition shows a typical pattern involving click ticks: two message handlers are defined, the first to detect whether a limit has been achieved and stops the application, the second just ignores the click:

```
1 Time(n::Int) when n > limit → stopAll();
2 Time(n::Int)                → {}
```

11. State

A value binding establishes an association between an identifier and a value. The association has a scope that defines the ESL code where the identifier can be referenced. ESL uses *lexical scoping* where the association is established in a construct (module, behaviour, let, letrec, function, case-arm, message handler) whose textual definition is the scope.

The original definition of the actor model of computation does not support side-effects. This restriction provides a rather austere application development platform where certain obvious implementation approaches become more unwieldy than they might otherwise be. Therefore, ESL provides side-effects on value bindings. These must be used with care since the actor model was originally designed to avoid issues such as race conditions that occur because of shared state.

Side-effects may be used to provide actors with mutable state:

```
1 Act BankAccount {
2   Deposit(Int);
3 }
4
5 act account::BankAccount {
6   funds::Int = 0;
7   Deposit(n::Int) → funds := funds + n;
8 }
```

An actor may encapsulate its state. Since an actor is singly-threaded, this means that there can be no doubt regarding whether an update has occurred or not when the value of a state variable is used, *i.e.* the state is not used by multiple threads. However, it is often useful for an actor to provide access to its internal state via its interface. This can be achieved in one of two key ways: message passing or an interface operation.

If access is provided by message passing then reference to the state remains singly threaded, however the message must contain the requesting actor who receives the state value via a return message:

```
1  Act BankAccount {
2    Deposit(Int);
3    Withdraw(Int,A);
4  }
5
6  act account::BankAccount {
7    funds::Int = 0;
8    Deposit(n::Int) → funds := funds + n;
9    Withdraw(n::Int,a::A) when n >= funds → {
10     funds := funds - n;
11     A ← Withdrawn;
12   }
13   Withdraw(n::Int,a::A) →
14     a ← WithdrawFailed;
15 }
```

Alternatively, access may be provided via an interface operation. In this case the state can be accessed using multiple threads raising the possibility of race conditions. Consider the following version of the bank account:

```
1  Act BankAccount {
2    export
3      getFunds::() → Int;
4      widthdraw::(Int) → Void;
5      deposit::(Int) → Void;
6  }
7
8  act account::BankAccount {
9    export getFunds,withdraw,deposit;
10   funds::Int = 0;
11   getFunds()::Int = funds;
12   deposit(n::Int)::Void = funds := funds + n;
13   withdraw(n::Int)::Void =
14     if n >= funds
15     then funds := funds - n;
16     else {}
17 }
```

Two different actors may share access to a bank account. The first actor performs the following:

```
1 a.deposit(100);
2 a.withdraw(10);
3 print[Str]('funds = ' + a.getFunds());
```

what value will be printed? We might expect the funds to be increased by `90`. However, the answer depends on whether the second actor has performed a deposit or withdrawal in between line `1` and `2` or between line `2` and `3`.

In order to be sure, such transactional blocks must be protected. All access to shared state must use locks to gain exclusive access. Each actor that uses the exported operations must do so within a `grab(lock)` block. Any value may be used as the lock providing the same value is used by all threads that share the state. In this case it makes sense to use the account as the lock:

```
grab(a) {
  a.deposit(100);
  a.withdraw(10);
  print[Str]('funds = ' + a.getFunds());
}
```

Now providing that all actors that share the account use a similar `grab(a)` to wrap any transactions, the actor above can be sure that the funds will have been increased by 90.

The scheme above relies on the client actors being well behaved in their use of the bank account by wrapping transactions in a `grab(a)`. A better way is for the bank account to enforce the protection of the state whilst still offering an interface. This can be achieved by passing the account a transaction function as follows:

```
Act BankAccount {
  export
    transaction::(()→Int,(Int)→Void,(Int)→Void)→Void
}

act account::BankAccount {
  export transaction;
  funds::Int = 0;
  getFunds()::Int = funds;
  deposit(n::Int)::Void = funds := funds + n;
  withdraw(n::Int)::Void =
    if n >= funds
    then funds := funds - n;
    else {}
  transaction(action::(()     → Int,
                       (Int) → Void,
                       (Int) → Void) → Void)::Void =
    grab(self) {
      action(getFunds,deposit,withdraw);
    }
}
```

Any client actor must provide the account with an action which is performed on the client's thread and is guaranteed to occur within the scope of the lock:

```
a.transaction(fun(getFunds::()     → Int,
                  deposit::(Int)   → Void,
                  withdraw::(Int) → Int)::Void {
  deposit(100);
  withdraw(10);
  print[Str]('funds = ' + getFunds());
});
```

12. Concurrency

ESL actors run concurrently. An actor is created using `new b` or `new b(x,...)` which immediately returns a handle to the newly created actor and schedules the thread of the newly created actor. The behaviour `b` may have an initialisation clause (a message handler starting with →) that is the first action performed on the new thread. After initialisation, the new thread enters a loop that inspects the actor's message queue and dispatches to a message handler in `b` if a message exists, otherwise the thread waits until a message is received. If `b` defines a handler for `Time(Int)` then the ESL system sends the actor a message providing the queue is empty.

```
Act Main       {                   }
Act Searcher   { Time(Int);        }
Act Controller { Time(Int); Found; }

size::Int                = 1000;
numOfSearchers::Int      = 10;
board::Array[Array[Str]] = new Array[Array[Str]](size);
max(a::Int,b::Int)::Int  = if a > b then a; else b;

act controller::Controller {
  count::Int = 0;
  Time(n::Int) → count := count + 1;
  Found → {
    print[Str]('Found in ' + count + ' steps');
    stopAll();
  }
}

act searcher(control::Controller)::Searcher {
  x::Int = random(size);
  y::Int = random(size);
  delta(n::Int)::Int = (n + max(random(3) - 1,0)) % size;
  Time(n::Int) →
    if board[x][y] = '*'
    then control ← Found;
    else { x := delta(x); y := delta(y); }
}

act main::Main {
  → {
    for x::Int in 0..size do {
      board[x] := new Array[Str](size);
      for y::Int in 0..size do
        board[x][y] := '';
    }
    board[random(size)][random(size)] := '*';
    let control::Controller = new controller;
    in {
      for i::Int in 0..numOfSearchers do
        new searcher(control);
    }
  }
}
```

Figure 3: Searching an Array

ESL is designed to make the creation and interaction of concurrent actors very lightweight. As described in `sec:state`, actors can share state and, if so, must make appropriate use of locks. However, in general the state of an actor is encapsulated and is modified by third party actors using message passing.

The use of large scale concurrency and asynchronous message passing changes the conventional approach to system design and implementation which is based on task decomposition and sequential ordering. This section provides a number of examples that show how actors can be used to implement tasks that take advantage of the concurrent computational model.

12.1. Search

Actors make the implementation of applications involving brute-force search simple because a number of actors can be created that concurrently navigate the search space. The amount of search space explored at any given time is proportional to the

number of actors created (and the amount of processing power available).

Consider a square two-dimensional string array that contains a single occurrence of `'*'` at an unknown location. A sequential program might start at `(0,0)` and move through the locations until it finds `'*'`. The worst case execution will inspect all the locations in the array.

A simple actor-based solution creates several actors that all search for the `'*'` concurrently. Each actor can start at a random location and move around at random. This provides a way of comparing the benefits of scaling up the number of actors in a brute-force search.

Figure 3 shows the implementation of the simple search application. The worst-case results of several runs are as follows:

numOfSearchers	count
1	2155079
10	40255
100	30331
1000	656

12.2. Quicksort

Sequential quicksort was defined in section 4. The algorithm clearly lends itself to concurrency since it splits on an element such that the lists of elements below and above the chosen number can be sorted independently. Figure 4 shows the definition of concurrent quicksort in ESL. The actor `qsorter` selects and element `x` and creates two independent child `qsorter` actors to sort the list of elements below `x` and above `x` respectively before changing behaviour to a `qwaiter`.

The `qwaiter` actor implements a typical scenario: waiting for two independent calculations to terminate before continuing. It achieves this using `Left` and `Right` tokens in the two children and waiting until it receives both tokens before sending the sorted list to its own parent.

Figure 5 shows a diagram representation of a simple quicksort in terms of the actors, their links and the messages that are sent. Note the dashed arrow showing `qsorter` actors becoming `qwaiter` actors.

12.3. Termites

Actor based systems can be used to exhibit *emergent behaviour* where system-level behaviour can be observed as a result of many simple individual behaviours even though each individual behaviour has no knowledge of the whole. An example is *termites* shown in figure 6 where a collection of twigs (black squares) and moved by a collection of termites (red squares) to form piles even though the individual termites do not have any knowledge of pile-formation.

The behaviour of a termite is simple: walk around at random until a twig is found that is not next to other twigs. The termite then picks up the lone-twig and walks at random until it finds another twig that is next to other twigs. The termite then drops the twig and then starts over again.

The implementation of termites in ESL is shown in figure 7. The implementation shows a typical scenario in actor-based systems where there is a single world-state and many different

```
1  data Direction = Left | Right | Final;
2
3  Act QSort { Sorted([Int],Direction); }
4
5  nums::[Int] = [ random(50) | n::Int ← 0..25 ];
6
7  act qmain(l::[Int])::QSort {
8    → new qsorter(self,l,Final);
9    Sorted(l::[Int],Final) → {
10     print[[Int]](l);
11     stopAll();
12   }
13 }
14
15 act qwaiter(parent::QSort,n::Int,dir::Direction)::QSort {
16   left::[Int] = null[[Int]];
17   right::[Int] = null[[Int]];
18   check()::Void =
19     if left <> null[[Int]] and right <> null[[Int]]
20     then parent ← Sorted(left+[n]+right,dir);
21     else {}
22   Sorted(l::[Int],Left)  → { left := l;  check(); }
23   Sorted(l::[Int],Right) → { right := l; check(); }
24 }
25
26 act qsorter(parent::QSort,l::[Int],dir::Direction)::QSort {
27   → case l {
28       [] → parent ← Sorted(l,dir);
29       x::Int:l::[Int] → {
30         new qsorter(self,[ n | n::Int ← l, ?(n<x) ],Left);
31         new qsorter(self,[ n | n::Int ← l, ?(n>x) ],Right)
     ;
32         become qwaiter(parent,x,dir);
33         {}
34       }
35     }
36   Sorted(l::[Int],d::Direction) → throw[Void] 'error!';
37 }
38
39 act main::Act{} { → new qmain(nums); }
```

Figure 4: Concurrent Quicksort

actors that share the state. Each individual needs the state in order to determine its next move which involves an update to the world state. Given that all the actors are operating concurrently, this dependency on a single world state can cause problems if access and updates are not protected. Conventional programming languages provide *data locks* to deal with shared state.

ESL provides locks, but often they are not required because actors operate in terms of asynchronous messages and queues. The diagram below shows how this works:

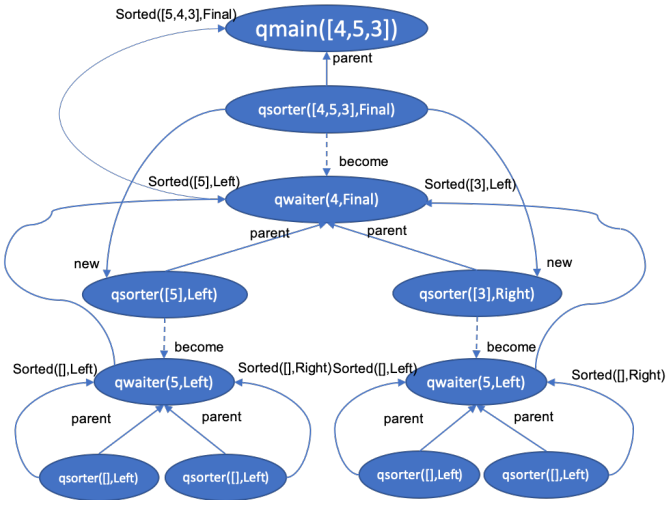


Figure 5: Example Quicksort

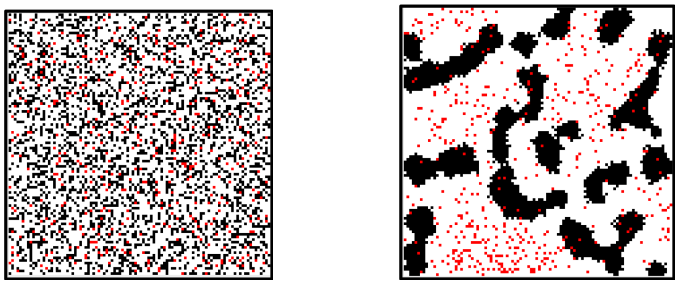


Figure 6: Termites Before and After

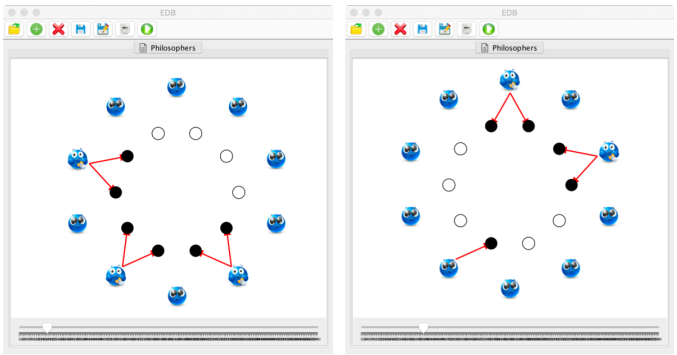
The world at the centre of the figure represents the shared state. Four independent worker actors are shown, although any number can be involved. The shared state offers an interface that supports the operations required by the worker actors which, in turn, provide an interface that represents the different outcomes from performing the world operations.

From the perspective of a single worker, it looks like they get dedicated access to the shared world state because a message sent to the world is queued until the world becomes free. Each worker message must include the worker actor so that the shared state can reply to it. When a worker sends a message to the shared state the worker becomes idle and will be woken up by the reply from the shared state. When the shared state processes a worker message, the worker has exclusive access to the data, the data can be changed and then the message is sent back to the worker which is woken up.

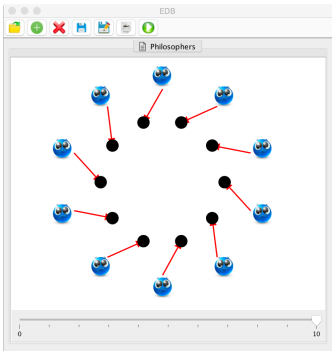
12.4. Dining Philosophers

Dining Philosophers is a standard scenario that is used to exemplify the issues of shared access to resources. Several concurrent processes (philosophers) are vying for shared resources (chopsticks) in order to eat. Each process needs exactly two resources to each and each resource is shared by exactly two processes. When a process acquires a pair of resources, it performs activity (eats) and then releases the resources. Access to resources is assumed to be fair. If a process acquires resources sequentially, then a situation can arise where all resources have acquired one resource and is awaiting on a resource that has been acquired by another process (starvation). In order to avoid starvation, the processes need to wait until they can individually acquire a pair of resources as an atomic action.

A situation where the philosophers are happily eating is shown below:



In the case where chopsticks are grabbed one at a time, the situation rapidly deadlocks:



The implementation of deadlock-free dining philosophers is shown below:

```
1  numberOfPhilosophers::Int = 10;
2  eatTime::Int              = 50;
3  thinkTime::Int            = 50;
4
5  Act Main { Time(Int); }
6  Act Chopstick { }
7  Act Philosopher { Time(Int); }
8
9  eat()::Void   = wait(eatTime);
10 think()::Void = wait(thinkTime);
```

```
Act Main { Time(Int); }
Act Grid { SetColour(Int,Int,Str); TermiteAt(Int,Int,Int);}

size::Int          = 120;
limit::Int         = 40000;
numOfTermites::Int = 600;
twig::Str          = 'black';
background::Str    = 'white';
grid::Grid         = new 'esl.grid.Grid'[Grid](size,size,6);

isLegal(x::Int,y::Int)::Bool =
  (x >= 0 and x < size) and (y >= 0 and y < size);

Act Termite {  Search; Drop; FindSpace; GetAway(Int); }

act termite(id::Int,w::World)::Termite {

  x::Int  = random(size);
  y::Int  = random(size);
  dx::Int = random(3) - 1;
  dy::Int = random(3) - 1;

  randomDir()::Void = {
    dx := random(3)-1;
    dy := random(3)-1;
  }
  move()::Void = {
    x := (x + dx); y := (y + dy);
    if x < 0
    then { dx := 1; x := 0; move();
    } else if x > (size - 1)
    then { dx := -1; x := (size - 1); move();
    } else if y < 0
    then { dy := 1; y := 0; move();
    } else if y > (size - 1)
    then { dy := 0 -1; y := (size - 1); move();
    } else grid ← TermiteAt(id,x,y);
  }
  moveRandom()::Void = {
    randomDir();
    move();
  }

  → self ← Search;

  Search → {
    moveRandom();
    w ← TryPickup(x,y,self);
  }
  Drop → {
    moveRandom();
    w ← FindPile(x,y,self);
  }
  FindSpace → {
    moveRandom();
    w ← TryDrop(x,y,self);
  }
  GetAway(0) → {
    self ← Search;
  }
  GetAway(n::Int) → {
    move();
    self ← GetAway(n-1);
  }
}


Act World {
  TryPickup(Int,Int,Termite);
  FindPile(Int,Int,Termite);
  TryDrop(Int,Int,Termite);
}

act world::World {
  locations::Array[Array[Str]] =
    let a::Array[Array[Str]] = new Array[Array[Str]](size);
    in {
      for x::Int in 0..size do {
        a[x] := new Array[Str](size);
        for y::Int in 0..size do {
          a[x][y] := if random(100) < 30
                     then twig
                     else background;
          grid ← SetColour(x,y,a[x][y]);
        }
      }
      edb.display[Grid]('Termites',grid);
      a;
    }
  termites::[Termite] =
    [ new termite(n,self) | n::Int ← 0..numOfTermites ];
  foundSingleton(x::Int,y::Int)::Bool =
    locations[x][y] = twig and twigCount(x,y) < 5;
  foundPile(x::Int,y::Int)::Bool =
    locations[x][y] = twig and twigCount(x,y) > 4;
  isTwig(x::Int,y::Int)::Bool =
    if isLegal(x,y)
    then locations[x][y] = twig;
    else false;
  twigCount(x::Int,y::Int)::Int =
    sum([ if isTwig(x+dx,y+dy) then 1 else 0 |
          dx ← [-1,0,1],
          dy ← [-1,0,1],
          ?(x<>0 or y<>0) ]);

  TryPickup(x::Int,y::Int,t::Termite) → {
    if foundSingleton(x,y)
    then {
      locations[x][y] := background;
      grid ← SetColour(x,y,background);
      t ← Drop;
    } else t ← Search;
  }
  FindPile(x::Int,y::Int,t::Termite) → {
    if foundPile(x,y)
    then t ← FindSpace;
    else t ← Drop;
  }
  TryDrop(x::Int,y::Int,t::Termite) → {
    if locations[x][y] = background
    then {
      locations[x][y] := twig;
      grid ← SetColour(x,y,twig);
      t ← GetAway(20);
    } else t ← FindSpace;
  }
}

w::World = new world;

act main::Main {
  Time(n::Int) when n > limit → stopAll();
  Time(n::Int) → { }
}
```

Figure 7: Termites

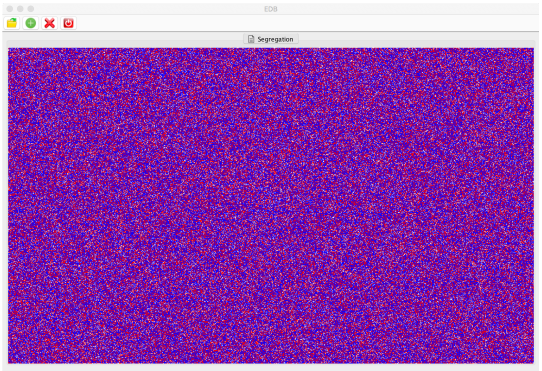
```
11
12 act philosopher(i::Int,
13                 left::Chopstick,
14                 right::Chopstick)::Philosopher {
15   Time(n::Int) → {
16     think();
17     grab(left,right) {
18       eat();
19     }
20   }
21 }
22
23 act chopstick::Chopstick {}
24
25 chopsticks::[Chopstick] =
26   [ new chopstick | i::Int ← 0..numberOfPhilosophers ];
27 chop(i::Int)::Chopstick =
28   nth[Chopstick](chopsticks,i%numberOfPhilosophers);
29
30 philosophers::[Philosopher] =
31   [ new philosopher(i,chop(i),chop(i+1))
32   | i::Int ← 0..numberOfPhilosophers ];
```

Deadlock is avoided because the `grab` block acquires the locks on `left` and `right` at the same time. If the locks are not available then the `grab` block will wait until they are both free and acquire them simultaneously. Changing the definition of `philosopher` to the following leads to deadlock:

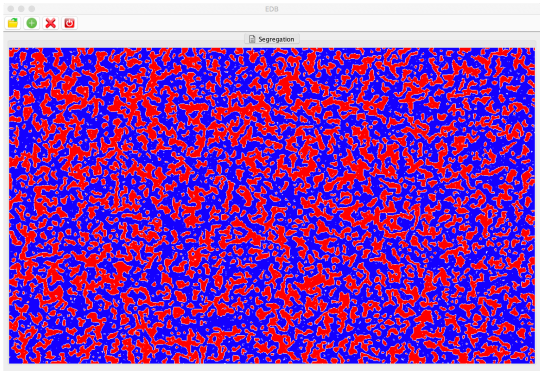
```
1  act philosopher(i::Int,
2                  left::Chopstick,
3                  right::Chopstick)::Philosopher {
4    Time(n::Int) → {
5      think();
6      grab(left) {
7        grab(right) {
8          eat();
9        }
10     }
11   }
12 }
```

12.5. Segregation

Thomas Schelling proposed an agent-based model of segregation in *Thomas C Schelling. Models of Segregation. American Economic Review, 59(2):488–493, 1969.* which demonstrates that relatively mild preferences for neighbours of a similar race can lead to high levels of segregation. The Schelling model shows how a completely random mix of two types of independent individuals:



can lead to a structured outcome:



when none of the individuals would claim to have designed the outcome. This is another example of emergent behaviour, where relative simple individual behaviours can lead to a combined outcome that none of the individuals have planned for.

The data types used in the ESL implementation of segregation are shown below:

```
1  type Main = Act { }
2
3  type Agent = Act {
4    export getX::()→Int;
5           getY::()→Int;
6           setX::(Int)→Void;
7           setY::(Int)→Void;
8  }
9
10 type Grid = Act {
11   SetColour(Int,Int,Str);
12   Swap(Int,Int,Int,Int);
13 }
14
15 type Population = Act {
16   Move(Agent);
17 }
18
19 data Loc = Coord(Int,Int);
20 coordX(p::Loc)::Int = case p { Coord(x::Int,y::Int) → x; }
21 coordY(p::Loc)::Int = case p { Coord(x::Int,y::Int) → y; }
```

As we see below, an agent is just a passive structure with state. Each agent is singly threaded through the shared population state by a series of messages. A grid is a foreign actor implemented in Java which draws a colour at an `(x,y)` position and can swap the colours between two different locations. A population controls the grid.

An agent manages its state and initiates a sequence of messages to a population:

```
1  act agent(x::Int,y::Int)::Agent {
2
3    export getX,getY,setX,setY;
4
5    getX()::Int = x;
6    getY()::Int = y;
7    setX(x0::Int)::Void = x := x0;
8    setY(y0::Int)::Void = y := y0;
9
10   → population ← Move(self);
11
12 }
```

The following parameters and constants control the segregation application:

```
diffLimit::Float = 25.00;  // % comfort limit.
width::Int       = 1000;   // with of population.
height::Int      = 600;    // height of population.
redpc::Int       = 40;     // % of population red.
emptypc::Int     = 10;     // % of population empty.
empty::Int       = 0;      // constant representing empty.
red::Int         = 1;      // constant representing red.
blue::Int        = 2;      // constant representing blue.
vLength::Int     = 0;      // number of vacancies.

opp(c::Int)::Int = if c = red then blue; else red;
colour(c::Int)::Str =
  if c = red then 'red';
  else if c = empty then 'white';
  else 'blue';

legalx(x::Int)::Bool = (x >= 0) and (x < width);
legaly(y::Int)::Bool = (y >= 0) and (y < height);
```

The main work of the application is performed by the population actor:

```
act pop::Population {

  createAgent(x::Int,y::Int)::Agent = {
    grid ← SetColour(x,y,colour(population[x][y]));
    new agent(x,y);
  }

  createVacancy(x::Int,y::Int)::Loc = {
    grid ← SetColour(x,y,colour(empty));
    Coord(x,y);
  }

  population::Array[Array[Int]] =
    // Create the basic population...
    let a::Array[Array[Int]] = new Array[Array[Int]](width);
    in {
      for w::Int in 0..width do {
        a[w] := new Array[Int](height);
        for h::Int in 0..height do {
          a[w][h] :=
            probably(100-emptypc)::Int {
              probably(redpc)::Int
                red;
              else blue;
            } else empty;
        }
      }
      a;
    }

  agents::Array[Array[Agent]] =
    // Create the agents...
    let a::Array[Array[Agent]] =
          new Array[Array[Agent]](width);
    in {
      for x::Int in 0..width do {
        a[x] := new Array[Agent](height);
        for y::Int in 0..height do {
          a[x][y] := if population[x][y] <> empty
                     then createAgent(x,y)
                     else null[Agent];
        }
      }
      a;
    }

  createVacancies()::Array[Loc] =
    // create an array of vacancies for easy access...
    let vacancies::[Loc] = [ createVacancy(x,y) |
                             x::Int ← 0..width,
                             y::Int ← 0..height,
                             ?(population[x][y]=empty) ];
    in {
      vLength := length[Loc](vacancies);
      let v::Array[Loc] = new Array[Loc](vLength);
      in {
        for i::Int in 0..vLength do {
          v[i] := nth[Loc](vacancies,i);
        }
        v;
      }
    }

  vacancies::Array[Loc] = createVacancies();

  popSet(x::Int,y::Int,c::Int)::Void =  population[x][y] := c;

  diffCellCount(x::Int,y::Int,c::Int)::Int =  length[Int]([ 1 |
    dx ← [-1,0,1],
    dy ← [-1,0,1],
    ?not(dx=0 and dy=0),
    ?legalx(x+dx),
    ?legaly(y+dy),
    ?population[x+dx][y+dy] = c]);

  diffpc(x::Int,y::Int)::Float =
    (intToFloat(diffCellCount(x,y,opp(population[x][y])))
     /8.0)*100.0;

  Move(a::Agent) → {
    let x::Int = a.getX(); y::Int = a.getY(); in
    if diffpc(x,y) > diffLimit
    then
      let i::Int = random(vLength); in
      let p::Loc = vacancies[i]; in
      let x0::Int = coordX(p); y0::Int = coordY(p);
      in {
        vacancies[i] := Coord(x,y);
        popSet(x0,y0,population[x][y]);
        popSet(x,y,empty);
        grid ← Swap(x,y,x0,y0);
        a.setX(x0);
        a.setY(y0);
        self ← Move(a);
      }
    else self ← Move(a);
  }
}
```

A `pop` actor receives a message `Move(a)`. Since the receiver is singly threaded, this provides `a` with exclusive access to the arrays `population` and `vacancies`. If each agent shared access to these resources, it is likely that movements could interleave and the state could become inconsistent. Once the move has occurred, the `pop` actor sends itself a `Move(a)` message which is queued with all other move requests. Since messages are delivered fairly, no agent is starved of movement.

13. Polymorphism

When a function is defined, its type signature defines the types of its arguments and its return type. Consider the function `idInt`:

```
idInt(x::Int)::Int = x;
```

which is similar to the function `idBool`:

```
idBool(x::Bool)::Bool = x
```

which is similar to many other functions such as `idFloat`, `idStr` and `idListOfInt`. In all cases the particular type of the argument is not relevant since the function always does the same thing: return the argument. It is important that the return type of the function is the same as the type of the argument. ESL allows a polymorphic function to be defined as follows:

```
id[T](x::T)::T = x;
```

When such a definition is used, the particular types must be supplied:

```
n::Int = id[Int](100);
b::Bool = id[Bool](true);
```

Multiple type arguments may be used. For example, the following is a function that applies a function `f` to all elements of a list in order to transform them from type `Source` to type `Target`:

```
map[Source,Target](l::[Source],f::(Source)→Target)::[Target]=
  case l {
    [] → [];
    h:t → f(h) : map[Source,Target](t,f);
  }
```

Type functions provide abstractions over type definitions. The type function `ListOf` maps a type `T` to the type `[T]`:

```
type ListOf[T] = [T];
```

Once defined, `ListOf[T]`, for some specific type expression `T` can be used as a type.

Data types can be defined parametrically. For example the following is a tree type for any type `T`:

```
data Tree[T] =
  Branch(Tree[T],Tree[T])
| Leaf(T);
```

Since the constructors `Branch` and `Leaf` are defined in the context of a type function, type arguments must be supplied when the constructors are used in patterns or to construct data values. For example:

```
treeTrans[T1,T2](map::(T1) → T2,t::Tree[T1])::Tree[T2] =
  case t {
    Branch[T1](l,r) → Branch[T2](treeTrans[T1,T2](map,l),
     treeTrans[T1,T2](map,r));
    Leaf[T1](v)     → Leaf[T2](map(v));
  }
```

and the two data values:

```
t1::Tree[Int]  =
  Branch[Int](Leaf[Int](100),Leaf[Int](200));
t2::Tree[Bool] =
  treeTrans[Int,Bool](fun(i::Int)::Bool i > 100,t1);
```

The type mechanisms are hidden behind the language structures. If these are unpacked then the reason for the type arguments should be clear:

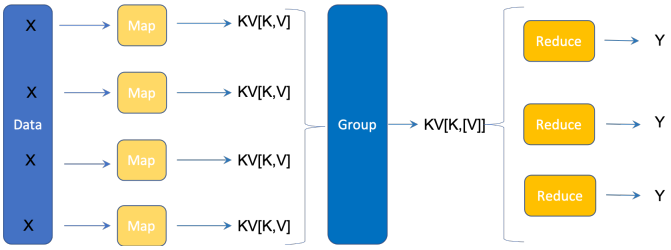
```
type Tree = rec X. Fun[Y]
  union {
    Branch(X[Y],X[Y]),
    Leaf(Y)
  }
```

The constructor types in a union-type definition are viewed as function that inject into the union-type. Since the union-type is defined as a type function, the constructors must be supplied with a type whenever they are used.

13.1. Map/Reduce

Map/reduce is an approach to processing large amounts of data that involves transforming the data and grouping with respect to keys. It is a pattern that is particularly suited to large amounts of concurrent processing because the mapping can occur independently and the grouping can be divided between different processing units. The approach has been used to help process internet data, such as counting the number of occurrences of text.

The diagram below shows how map/reduce works:



The data is of type `X` and is mapped to produce a collection of key-value pairs where the keys are of type `K` and the value of type `V`. Many different pairs may contain the same key; the pairs are grouped so that all the values associated with the same key are located together. Finally, each collection of values for the same key are reduced to produce a value of type `Y`.

The map/reduce process is independent of the type of data `X`, the type of the keys `K` and the type of the target values `Y`. It is therefore a suitable candidate for polymorphism. The example will use a collection of words:

```
import 'esl/strings.esl';

words::[Str] = splitBy(#\space,'Sed ut perspiciatis unde
    omnis iste natus error sit voluptatem accusantium
    doloremque laudantium, totam rem aperiam, eaque ipsa
    quae ab illo inventore veritatis et quasi architecto
    beatae vitae dicta sunt explicabo. Nemo enim ipsam
    voluptatem quia voluptas sit aspernatur aut odit aut
    fugit, sed quia consequuntur magni dolores eos qui
    ratione voluptatem sequi nesciunt. Neque porro
    quisquam est, qui dolorem ipsum quia dolor sit amet,
    consectetur, adipisci velit, sed quia non numquam eius
     modi tempora incidunt ut labore et dolore magnam
    aliquam quaerat voluptatem. Ut enim ad minima veniam,
    quis nostrum exercitationem ullam corporis suscipit
    laboriosam, nisi ut aliquid ex ea commodi consequatur?
     Quis autem vel eum iure reprehenderit qui in ea
    voluptate velit esse quam nihil molestiae consequatur,
     vel illum qui dolorem eum fugiat quo voluptas nulla
    pariatur?');
```

We will define a map/reduce implementation that uses polymorphic ESL actors to perform concurrent maps. The single

implementation will be used in two ways: (1) to produce the number of words of each length; (2) to produce the number of occurrences of each word.

```
act Main {}

data KV[Key,Value] = Key(Key,Value);

type Map[In]           = Act { Perform(In);              }
type Group[Key,Value]  = Act { Store(KV[Key,Value]);     }
type Reduce[Key,Value] = Act { Process(KV[Key,[Value]]); }
```

The data type `KV` is polymorphic with respect to the type of the key and the value. The type `Map` is a behaviour that it independent of the type of data items it processes via the `Perform` message. The type `Group` stores key-value pairs independent of they specific types, and the type `Reduce` processes key-value pairs after they have been grouped.

A map actor is created with the behaviour `mkMap`:

```
act mkMap[In,Key,Value](f::(In)→KV[Key,Value],
                        g::Group[Key,Value])::Map[In]  {
  Perform(x::In) →
    g ← Store(f(x));
}
```

The `mkMap` function takes a mapping function `f` and a grouping actor `g`. When a map actor is sent a `Perform(x)` message it uses `f` to transform `x` to a key-value pair which is supplied to the grouping actor `g`.

A group actor has the following behaviour:

```
act mkGroup[K,V](n::Int,reduce::Reduce[K,V])::Group[K,V] {
  results :: [KV[K,[V]]] = [];
  add(x::KV[K,V],rs::[KV[K,[V]]])::[KV[K,[V]]] =
    case x {
      Key[K,V](i::K,v::V) →
        case rs {
          [] → [Key[K,[V]](i,[v])];
          Key[K,[V]](j,vs):rs when i = j →
            Key[K,[V]](i,v:vs):rs;
          r:rs  → r:add(x,rs);
        }
    }
  Store(x::KV[K,V]) when n = 1 → {
    results := add(x,results);
    for r::KV[K,[V]] in results do {
      reduce ← Process(r);
    }
  }
  Store(x::KV[K,V]) → {
    results := add(x,results);
    n := n - 1;
  }
}
```

A group actor is supplied with the number of independent mappings `n` and a reducer actor `reduce`. The grouped results are maintained in the list `results`. Note that the type of `results` is `[KV[K,[V]]]`; the idea is that each key is associated with all the values supplied to the grouping agent.

A grouping agent is supplied with key-value pairs in the message `Store(KV(k,v))`. If this not the last pair, then it is added to the results using `add` which ensures that all the values associated with the same key are grouped together. When the last key-value pair is received, the results are passed to the reducing actor.

Map/reduce is performed by the following function:

```
mapReduce[In,Key,Value](l::[In],
                        f::(In)→KV[Key,Value],
                        g::Group[Key,Value])::Void =
  for x::In in l do {
    (new (mkMap[In,Key,Value])(f,g)) ← Perform(x);
  }
```

13.1.1. All Words of a Given Length

```
act reduceWordLen::Reduce[Int,Str] {
  Process(r::KV[Int,[Str]]) →
    case r {
      Key[Int,[Str]](i::Int,s::[Str]) →
        print[Str](i + ' ' + s);
    }
}

strLen(s::Str)::KV[Int,Str] =
  Key[Int,Str](length[Int](s.explode),s);

mapReduceWordLength()::Void =
  let r::Reduce[Int,Str] = new reduceWordLen;
      n::Int = length[Str](words); in
  let g::Group[Int,Str] = new (mkGroup[Int,Str])(n,r);
  in mapReduce[Str,Int,Str](words,strLen,g);
```

13.1.2. Occurrences of a Word

```
act occurs::Reduce[Str,Int] {
  Process(r::KV[Str,[Int]]) →
    case r {
      Key[Str,[Int]](s::Str,is::[Int]) →
        print[Str](s + ' ocurrences = ' + length[Int](is));
    }
}

strOccurs(s::Str)::KV[Str,Int] = Key[Str,Int](s,1);

mapReduceOccurrences()::Void =
  let r::Reduce[Str,Int] = new occurs;
      n::Int = length[Str](words); in
  let g::Group[Str,Int] = new (mkGroup[Str,Int])(n,r);
  in mapReduce[Str,Str,Int](words,strOccurs,g);
```

13.2. Cached Functions

Single-argument functions without side effects can be cached so that the computation needed to produce the result is performed once and then added to a table. Subsequent applications of the function to the same argument will just look the value up in the table. Although the type of the argument and the return value differs between functions, the caching activity is the same in all cases. polymorphism can be used to abstract the type of the argument and return value:

```
cache[Arg,Value](f::(Arg) → Value)::(Arg) → Value  =
  let table::Hash[Arg,Value] = new Hash[Arg,Value];
  in {
    fun(a::Arg)::Value
      if table.hasKey(a)
      then table.get(a);
      else
        let v::Value = f(a);
```

```
9         in {
10          table.put(a,v);
11          v;
12        }
13    }
```

Suppose we construct lists of the form `0..n` repeatedly for a range of values `n` where the same `n` may occur frequently. It is wasteful to construct the same list multiple times, especially if the value of `n` is high. It is much more efficient to use `cache`:

```
1 getList::(Int) → [Int] = cache[Int,[Int]](fun(n::Int) 0..n);
```

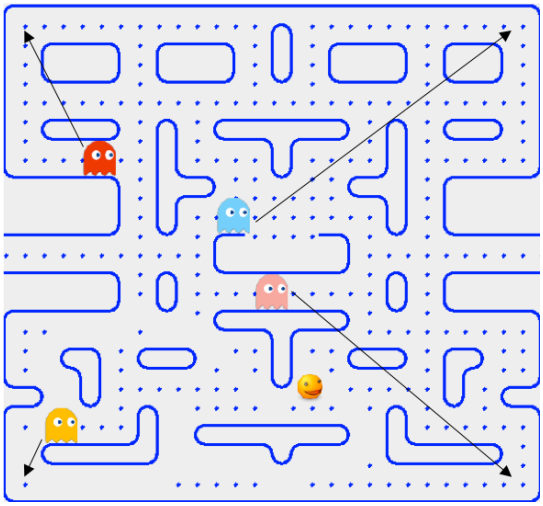
14. Object Orientation

ESL supports subtypes and inheritance. Behaviour types can be defined as an extension of an existing behaviour type in which case all the message definitions and export declarations from the parent are included in the child. Whenever an identifier is declared to be of a particular behaviour type, any behaviour that conforms to an extension of the declared type can be supplied.

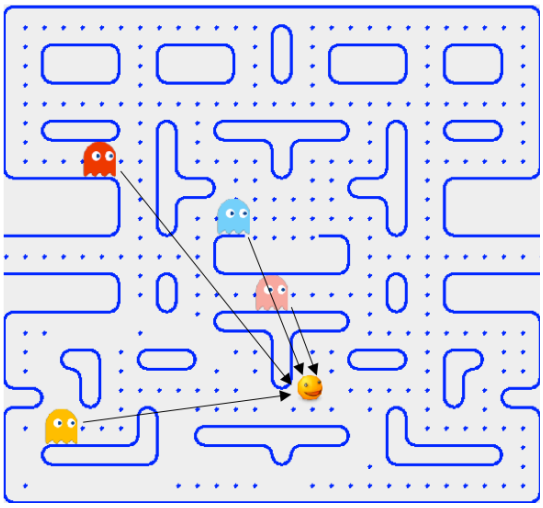
Behaviour definitions can extend a parent definition in which case the message handlers from the parent are all included in the child (in the same way that Java methods are inherited). Where the message signatures overlap, the child handlers will take precedence, but the parent definitions will be used where none of the child definition match. Parent local definitions are available in the child without qualification using the keyword `super`. This protects the integrity of the parent by making access to parent state and behaviour explicit.

To demonstrate ESL object-oriented features we will develop a Pacman game where inheritance allows position information to be abstracted into a behaviour that is then reused across players and ghosts. Sub-types are used to produce different types of ghosts and players.

The game consists of a maze containing 4 ghosts, some food and a Pacman. The Pacman moves around eating the food and the ghosts move around trying to catch the Pacman. The ghosts can only move horizontally and vertically whilst the Pacman can also move diagonally. No ghost or the Pacman can move through walls. At any given time, the ghosts are in one of two modes: `Scatter` and `Chase`. When in `Scatter` mode, the ghosts try to reach their home corner; each ghost has a different home:



When in `Case` mode, the ghosts head towards the Pacman and try to occupy the same location in which case the Pacman loses:



The user interface is defined as an actor that is supplied with messages to update the Pacman maze:

```
1  data GhostInfo = Point(Ghost,Int,Int,Int); // Point(g,type,
       x,y)
2
3  data GhostState =
4    Scatter
5  | Chase;
6
7  Act GUI {
8    export mouseX::Int; mouseY::Int;
9    State([GhostInfo],Array[Array[Int]]);
10   Player(Int,Int);
11 }
```

The Pacman maze is implemented as an actor with the behaviour type `Maze`. The maze provides the shared world state and ensures that actors move in turn each time the `Move` message is processed:

```
1 Act Maze {
2   export isWall::(Int,Int)→Bool;
3          atHome::(Int,Int)→Bool;
4          getPacman::() → Pacman;
5          getGhosts::() → [Ghost];
```

```
        legalPos::(Int,Int)→Bool;
        eat::(Int,Int)→Void;

  RegisterGhost(Ghost);
  RegisterPacman(Pacman);
  Move;
}
```

All the actors in the maze will be sub-types of `Player` which provides basic features for location and starting to play:

```
Act Player {
 export
    getX::() → Int;
    getY::() → Int;
    setX::(Int) → Void;
    setY::(Int) → Void;
    getMaze::() → Maze;
    moveTo::(Int,Int)→Void;
    move::()→Void;
    canMove::(Int,Int)→Bool;
    plan::(Int,
           (Plan,Plan)→Bool,
           (Plans)→Plan,
           (Plan)→Bool,
           (Plan)→[Loc],
           Plans)
          → Plan;
    moves::(Plan)→[Loc];
    deltaOK::(Int,Int)→Bool;
    initPlans::()→Plans;
    deltas::[Int];

  Start;
}
```

The operation `move` is called to make a single-step move in the game. A player uses `plan` to plan out the move. The arguments of plan are as follows:

```
plan(max,compare,goal,ok,moves,start)
```

where `max` determines the maximum length of the plan, `compare` is a predicate that compares plans such that `compare(p1,p2)` is true when `p1` is a better plan than `p2`, `goal` maps a collection of plans to a single plan that achieves a goal or returns `null`, `ok` returns true when a plan is legal, `moves` maps a plan to a sequence of locations that are legal next moves; `start` is a collection of starting plans (usually the current position of a player).

The `plan` operation is used to automatically generate next moves for players and is implemented differently for ghosts and the Pacman. Some of the player operations can be implemented and then reused across ghosts and players:

```
ct player(x::Int,y::Int,maze::Maze)::Player {

  // A basic implementation of the player interface...

  export getX,getY,setX,setY,moveTo,getMaze,move,canMove,
         moves,plan,deltaOK,initPlans,deltas;

  getX()::Int = x;
  getY()::Int = y;
  setX(v::Int)::Void = x := v;
  setY(v::Int)::Void = y := v;
  moveTo(v1::Int,v2::Int)::Void = { x := v1; y := v2; }
  getMaze()::Maze = maze;
  move()::Void = throw[Void]('abstract operation move');
  canMove(x::Int,y::Int)::Bool =
    maze.legalPos(x,y) or maze.atHome(x,y);
  deltas::[Int] =  [-1,0,1];
  deltaOK(dx::Int,dy::Int)::Bool = not(dx = 0 and dy = 0);
  initPlans()::Plans = [[Loc(self.getX(),self.getY())]];

  getBestPlan(better::(Plan,Plan)→Bool,plans::Plans)::Plan =
    select1[Plan](plans,null[Plan],fun(p1::Plan)::Bool
      not(exists[Plan](fun(p2::Plan)::Bool
        p1 <> p2 and better(p2,p1),plans)));

  plan(max::Int, better::(Plan,Plan)→Bool,
       goal::(Plans)→Plan,pred::(Plan)→Bool,
       moves::(Plan)→[Loc],plans::Plans)::Plan =
    if plans = []
    then null[Plan];
    else if max < 0
    then getBestPlan(better,plans);
    else
      let p::Plan = goal(plans);
          plans::Plans =
            [p | p ← addMoves(plans,moves), ?pred(p)];
      in if p = null[Plan]
         then plan(max-1,better,goal,pred,moves,plans);
         else p;

  hasMove(m::Loc)::(Plan)→Bool =
    fun(p::Plan)::Bool member[Loc](m,p);

  addMoves(plans::Plans,moves::(Plan)→[Loc])::Plans =
    [ m:p | p ← plans,
            m ← moves(p),
            ?(not(exists[Plan](hasMove(m),plans))) ];

  moves::(Plan) → [Loc] =
    cache[Plan,Loc,[Loc]](head[Loc],fun(p::Plan)::[Loc]
      case head[Loc](p) {
        Loc(x::Int,y::Int) →
          [ Loc(x+dx,y+dy) |
              dx ← deltas,
              dy ← deltas,
              ?(self.deltaOK(dx,dy)),
              ?(canMove(x+dx,y+dy)) ];
      });

  Start → throw[Void]('abstract message Start');
}
```

In `player` the implementation of `plan` uses the supplied arguments to map a collection of plans to a single best plan. A plan is a sequence of locations, in reverse order, that are a legal sequence of moves from some starting position. The operation `addMoves` is used by `plan` to add a new move to the end of each plan. A check is made in `addMoves` to ensure that no plan contains a cycle.

****

In Pacman there are four types of ghost. Each type has a different behaviour which will be implemented as different sub-behaviours of a common parent behaviour `ghost`. The maze is drawn by a Swing-based Java application that will draw the ghosts in different colours, supplied to the Java as integer codes that are the value of `ghostType`:

```
Act Ghost extends Player { export ghostType::Int; }
```

We create an empty behaviour type as placeholder in the hierarchy for players:

```
act Pacman extends Player { }
```

Figure 8 defines an abstract ghost behaviour that inherits the behaviour `player`. A ghost exists in one of two states: `Chase` and `Scatter` which controls whether it is trying to catch the Pacman or trying to reach its home position. It achieves this by creating a plan using the local function `plan`. A plan is created by extending a collection of current plans with all the possible legal moves calculated by the function `moves`. The length of the plan is controlled by `maxPlanLength`; when this is reached, the best plan is selected amongst all the available plans. The best plan is that which minimises the distance to the goal location.

Having defined an abstract ghost behaviour we can extend it to produce four different concrete ghost behaviours. In the real game, the ghosts differ by strategy to catch the Pacman, however we will limit the differences to the home locations:

```
act ghost1(x::Int,y::Int,m::Maze)::Ghost
  extends ghost(x,y,1,1,0,m) { }
act ghost2(x::Int,y::Int,m::Maze)::Ghost
  extends ghost(x,y,1,maxHeight-2,1,m) { }
act ghost3(x::Int,y::Int,m::Maze)::Ghost
  extends ghost(x,y,maxWidth-2,1,2,m) { }
act ghost4(x::Int,y::Int,m::Maze)::Ghost
  extends ghost(x,y,maxWidth-2,maxHeight-2,3,m) { }
```

An abstract Pacman is a player that registers itself with the maze:

```
act pacman(x::Int,y::Int,m::Maze)::Pacman

  extends player(x,y,m) {

  Start →
    m ← RegisterPacman(self);
}
```

There will be two different concrete Pacman behaviours. The first allows a human player to control the Pacman using the mouse. This will be achieved by querying the current mouse position:

```
act playerPacman(x::Int,y::Int,m::Maze)::Pacman

  extends pacman(x,y,m) {

  export move;

  move()::Void = {
    let mouseX::Int = gui.mouseX;
        mouseY::Int = gui.mouseY;
        x::Int = self.getX(); y::Int = self.getY(); in
    let dx::Int = mouseX - x;
        dy::Int = mouseY - y; in
    let mx::Int =if dx>0 then 1;else if dx<0 then -1;else 0;
        my::Int =if dy>0 then 1;else if dy<0 then -1;else 0;
    in {
      if (mx <> 0) and m.legalPos((x+mx)%maxWidth,y)
      then self.setX((x+mx)%maxWidth);
      else
        if (my <> 0) and
           m.legalPos(x,y+my)
        then self.setY(y+my);
        else
          if m.legalPos((x+mx)%maxWidth,y+my)
          then self.moveTo((x+mx)%maxWidth,y+my);
          else
            if x + mx = 0 and m.legalPos(0,y)
            then self.setX(maxWidth -1);
            else {}
      gui ← Player(x,y);
      m.eat(x,y);
    }
  }
}
```

An alternative implementation of the player provides a machine implemented behaviour. Like ghosts, the automatic movement is defined by a simple planner that tries to maximise the amount of food that is eaten whilst avoiding ghosts.

The following variables control the game:

```
cellWidth::Int          = 20;    // Used to convert the (
    mouseX,mouseY) positions.
cellHeight::Int         = 20;    // Used to convert the (
    mouseX,mouseY) positions.
maxWidth::Int           = 29;    // Max number of horizontal
    locations in the maze.
maxHeight::Int          = 27;    // Max number of vertical
    locations in the maze.
delay::Int              = 100;   // Used to slow down the
    game.
cellEmpty::Int          = 0;     // Code for an empty cell.
legalCell::Int          = 2;     // No occupation above this
    code.
homeCell::Int           = 3;     // Code for ghost home.
hWall::Int              = 4;     // Code for a horizontal
    wall.
vWall::Int              = 5;     // Code for a vertical wall.
topLeftCorner::Int      = 6;     // Code for a top-left
    corner.
topRightCorner::Int     = 7;     // Code for a top-right
    corner.
bottomLeftCorner::Int   = 8;     // Code for a bottom-left
    corner.
bottomRightCorner::Int = 9;      // Code for a bottom-right
    corner.
timeLimit::Int          = 70000; // How long to run the game.
```

15. EDB

EDB is an editor-based development environment for ESL that integrates syntax checking, type checking, compilation, application execution and run-time debugging. EDB relies on access to a Java compiler as provided by:

`javax.tools.ToolProvider.getSystemJavaCompiler()`

At startup, EDB is supplied with the following run-time Java arguments in order:

1. A path to the directory that contains the ESL source code. From the root installation directory, this will be `esl`.

2. A path to a directory containing EDB data that is saved between executions. This includes information about the current state of EDB when it is shut down so that it can be recreated.

EDB consists of a collection of tabs. Each tab has a particular type:

```
act ghost(x::Int,y::Int,homeX::Int,homeY::Int,gType::Int,m::Maze)::Ghost extends player(x,y,m) {
  export ghostType, move;

  ghostType::Int     = gType;        // The type of the ghost.
  state::GhostState  = Chase;        // Current state, used for planning.
  changeMode::Int    = 15;           // Change the state after this number of moves.
  modeCount::Int     = changeMode;   // Current countdown to change state.
  maxPlanLength::Int = 8;            // Select the best plan after this number of plan steps.

  move()::Void =
    letrec
      selectMin(r::(Plan,Plan)→Bool,plans::[Plan])::Plan =
        select1[Plan](plans,null[Plan],fun(p1::Plan)::Bool
          forall[Plan](fun(p2::Plan)::Bool if p1=p2 then true else r(p1,p2),plans));
      planDistance(p::Plan,x::Int,y::Int)::Float = case head[Loc](p) { Loc(x0,y0) → distance(x,y,x0,y0); }
      plan(goalX::Int,goalY::Int,plans::Plans)::Plan =
        // Create a plan that takes the ghost nearest to the goal position...
        if length[Loc](head[Plan](plans)) > maxPlanLength
        then
          selectMin(fun(p1::Plan,p2::Plan)::Bool planDistance(p1,goalX,goalY) <= planDistance(p2,goalX,goalY),plans);
        else
          let p::Plan = select1[Plan](plans,null[Plan],fun(p::Plan)::Bool head[Loc](p) = Loc(goalX,goalY));
          in if p = null[Plan]
             then plan(goalX,goalY,[ m:p |
               p ← plans,m ← moves(p),?(not(exists[Plan](fun(p::Plan)::Bool member[Loc](m,p),plans))) ]);
             else p;
      getMove(goalX::Int,goalY::Int)::Loc =
        // Get a move towards the goal...
        case plan(goalX,goalY,[[Loc(self.getX(),self.getY())]]) {
          [Loc(x,y)] → Loc(x,y);           // Nothing available, no move.
          p → last[Loc](butlast[Loc](p));  // Plan is in reverse and contains current location.
        }
      moves::(Plan) → [Loc] = cache[Plan,Loc,[Loc]](head[Loc],fun(p::Plan)::[Loc]
        // Get the available moves based on the location at the head of the supplied plan...
        case head[Loc](p) {
          Loc(x::Int,y::Int) →
            [ Loc(x+dx,y+dy) | dx ← [-1,0,1], dy ← [-1,0,1], ?(not(x=0 and y=0)), ?(self.canMove(x+dx,y+dy)) ];
        });
    in {
      case state {
        Scatter when modeCount > 0 →  // Scattering heads towards the home location...
          case getMove(homeX,homeY) {
            Loc(x::Int,y::Int) → {
              self.moveTo(x,y);
              modeCount := modeCount - 1;
            }
          }
        Scatter → { // Switch mode to chasing the pacman...
          state := Chase;
          modeCount := changeMode;
        }
        Chase when modeCount > 0 →  // Chasing heads towards the pacman...
          case getMove(m.getPacman().getX(),m.getPacman().getY()) {
            Loc(x::Int,y::Int) → {
              self.moveTo(x,y);
              modeCount := modeCount - 1;
            }
         }
        Chase → { // Switch mode to heading back home...
          state := Scatter;
          modeCount := changeMode;
        }
      }
    }

  Start →
    m ← RegisterGhost(self);
}
```

Figure 8: Abstract Ghost Behaviour

```
act autoPacman(x::Int,y::Int,m::Maze)::Pacman extends pacman(x,y,m) { export move;
  plan::Plan              = [];         // A sequence of locations in reverse order.
  fleeing::Bool           = false;      // Is the pacman in danger of being eaten?

  moves::(Plan) → [Loc] = cache[Plan,Loc,[Loc]](head[Loc],fun(p::Plan)::[Loc]
    case head[Loc](p) {
      Loc(x::Int,y::Int) →
        [ Loc(if x+dx = 0 then maxWidth-1 else x+dx,y+dy) |  dx::Int ← [1,0,-1], dy::Int ← [1,0,-1],
            ?(not(dx=0 and dy=0)),
            ?(self.canMove(x+dx,y+dy)) ];
    });

  ghostDistance(l::Loc)::Float = min(1000.0,[ distance(locX(l),locY(l),g.getX(),g.getY()) | g::Ghost ← m.getGhosts() ]);

  nextMove(p::Plan)::Loc = case p {  p1 + [l] + p2 when length[Loc](p2) = 1 → l; }

  move()::Void =
    let d::Float = ghostDistance(Loc(self.getX(),self.getY()));
    in if d < 5.0 and not(fleeing) then { flee(7); fleeing := true; doPlan(); }
       else if d > 5.0 and fleeing then { eat(9); fleeing := false; doPlan(); }
       else doPlan();

  ghostMoves::(Loc) → [Loc] = cache[Loc,Loc,[Loc]](id[Loc],fun(l::Loc)::[Loc]
    case l { Loc(x::Int,y::Int) →
      [ Loc(x+dx,y+dy) | dx ← [1,0,-1], dy ← [1,0,-1], ?(dx=0 or dy=0), ?(not(dx=0 and dy=0)), ?(self.canMove(x+dx,y+dy))
      ];
    });

  getDangerZones(moves::Int)::[[Loc]] =
    letrec ghostLocs::[Loc] = [ Loc(g.getX(),g.getY()) | g::Ghost ← m.getGhosts() ];
           zones(zone::[Loc],i::Int,max::Int)::[[Loc]] =
             if i >= max then []; else let z::[Loc] = [ m  | l::Loc ← zone, m::Loc ← ghostMoves(l) ]; in z:zones(z,i+1,max);
    in zones(ghostLocs,0,moves);

  eat(n::Int)::Void =
    letrec dangerZones::[[Loc]] = getDangerZones(n+1);
           eatPlans(plans::Plans,i::Int,max::Int)::Plans =
             if i >= max then plans;
             else eatPlans([ l:p |  p ← plans, l ← moves(p),
               ?(not(exists[[Loc]](fun(zone::[Loc])::Bool member[Loc](l,zone),take[[Loc]](dangerZones,i+1)))),
               ?(not(exists[Plan](fun(p::Plan)::Bool member[Loc](l,p),plans)))
             ],i+1,max);
      foodCount(plan::Plan)::Int = let count::Int=0; in { for l in plan do count := count + food[locY(l)][locX(l)]; count;}
      moreFood(plan1::Plan,plan2::Plan)::Bool = foodCount(plan1) > foodCount(plan2);
      mostFood(plans::Plans)::Plan = max[Plan](plans,moreFood);
    in case mostFood(eatPlans([[Loc(self.getX(),self.getY())]],0,n)) {
         p when p = null[Plan] → { print[Str]('Yikes - no plan!'); plan := []; }
         p → plan := butlast[Loc](p);
       }

  flee(n::Int)::Void =
    letrec dangerZones::[[Loc]] = getDangerZones(n+1);
           fleePlans(plans::Plans,i::Int,max::Int)::Plans =
             if i >= max  then plans;
             else fleePlans([ l:p | p::Plan ← plans, l::Loc ← moves(p),
               ?(not(exists[[Loc]](fun(zone::[Loc])::Bool member[Loc](l,zone),take[[Loc]](dangerZones,i+1)))),
               ?(not(exists[Plan](fun(p::Plan)::Bool member[Loc](l,p),plans)))
             ],i+1,max);
    in case fleePlans([[Loc(self.getX(),self.getY())]],0,n) {
         [] → { print[Str]('Yikes - no plan!'); plan := []; }
         p:ps → plan := butlast[Loc](p);
       }

  doPlan()::Void =
    if plan = [] then replan(); else case plan {
      p1 + [l] → { plan := p1; case l { Loc(x,y) → { self.moveTo(x,y);  gui ← Player(self.getX(),self.getY()); } }
    }

  replan()::Void = if fleeing then flee(7); else eat(9);
}
```

Figure 9: Automatic Pacman

HTML Contains HTML that has been generated by an ESL application or from a file. The HTML is displayed using a browser implemented by JxBrowser[1]. Handlers can be registered with EDB that process hyperlinks when they are clicked.

ESL Contains the contents of an ESL file that can be edited within EDB, written to the file system, compiled to produce Java and then run. An ESL tab provides syntax checking and type checking for the ESL language.

Java EDB is not intended to provide extensive support for Java development, however it provides limited support for editing, pretty-printing and compilation of Java source code to support the debugging and modification of Java that is produced when ESL is compiled.

TabbedActor Java based applications that use Swing libraries can be integrated into EDB by defining a class that implements the `edb.editor.TabbedActor` interface and that extends a Java Swing container.

15.1. EDB Interface

The EDB interface is shown in figure 10 with a single ESL tab. The text is highlighted according to ESL syntax rules which are checked dynamically as the text is modified. Errors are shown in two ways: (1) by underlining the text containing the error in red; (2) as a red marker in the margin. Hovering over either of these with the mouse will describe the error.

The toolbar buttons provide access to EDB functionality. For an ESL tab the following functionality is supported:

button	key	description
		Load file into EDB.
		Create new file.
		Delete file.
		Stop a running application.
		Change font.
	Meta-S	Save file.
		Save file as.
	Meta-T	Touch file and imports.
	Meta-K	Compile ESL to Java.
	Meta-J	Pretty print the Java source.
	Meta-R	Compile and run file.

Pressing command and clicking on an identifier in an ESL editor will jump to its definition. The background menu provides access to all definitions via their name. All the defined names in an ESL file can be copied to the clipboard via the background menu; this is very useful in order to export the names in a file via the `export` clause at the start of the file, which can be created via copy and paste.

A Java tab provides a subset of the ESL functionality in addition to:

Both types of editor support a background menu that offers functionality for search, find and replace, copy text as a bitmap to the clipboard, and zoom.

15.2. EDB Actor

EDB can be referenced within a running ESL application using the variable `edb` whose type is defined in the module `esl/displays.esl`:

```
1  Act EDB {
2    export
3      math::Math;
4      message::Forall[T] (T)→T;
5      display::Forall[T] (Str,T)→Void;
6      button::(Str,Str,Str,Str,()→Void)→Void;
7    Show(Str,Display);
8    Filmstrip(Str,[Display]);
9    AddBrowserListener(BrowserListener);
10   Edit(EditType);
11 }
12
13 type Math = {
14   circlePos::(Int,Int,Int,Int) → Point
15 };
16
17 Act BrowserListener {
18   BrowserEvent(Str);
19 }
20
21 data EditType =
22   ESLSource(Str,Str)
23 | JavaSource(Str,Str,Str)
24 | RawText(Str);
```

The EDB object can be used to display graphics (as described below), register hyperlink event listeners, and to edit source code by creating an EDB tab.

15.3. EDB Displays

EDB supports a data type `Display` that can be used to create a variety of graphical displays. Sending the message:

```
1 edb ← Show(label,display);
```

creates (or replaces) an EDB tab with the supplied label and the graphical display. The display type `Display` is defined in appendix B.

15.4. Tables

EDB can display HTML tables, for example the following table:

[1] `https://www.teamdev.com/jxbrowser`

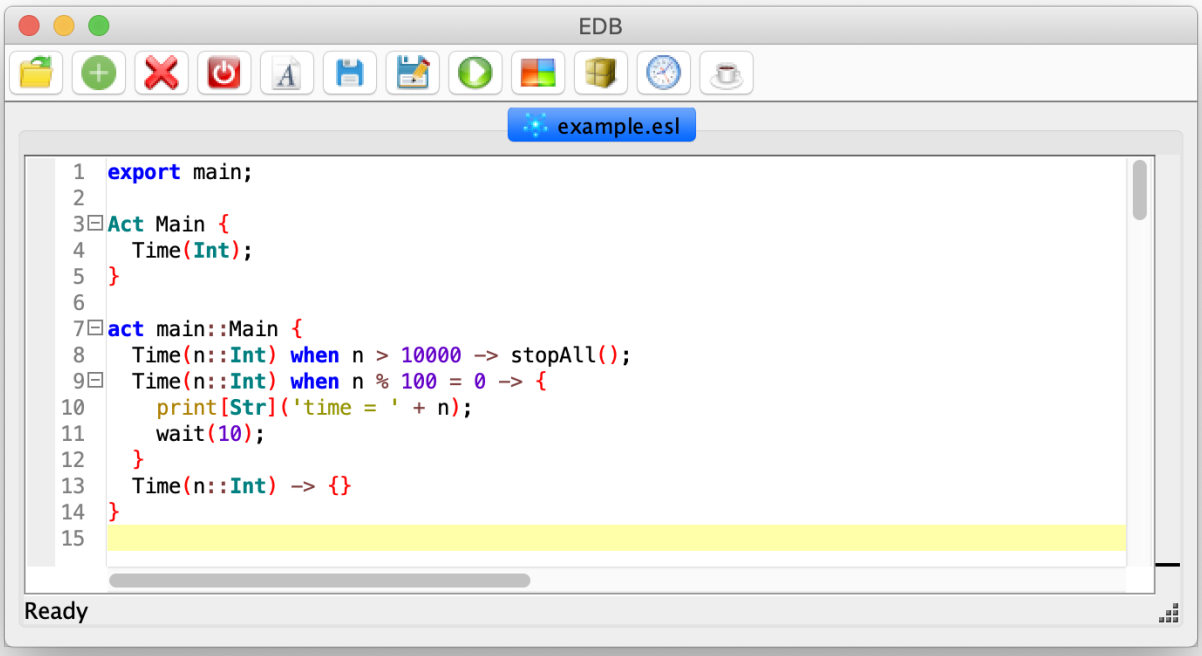


Figure 10: EDB Showing an ESL Tab

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

is produced by:

```
1  props::Props = [Prop('style','border: 1px solid black;')];
2
3  table::Display =
4    Table(props,
5      [ Row([],
6         [ Data(props,HTML('' + j))
7         | j ← (i*10)..((i*10)+10)
8         ])
9      | i ← 0..10
10     ]);
11 edb ← Show('mytable',table);
```

Since tables are HTML, it is possible to include links. The links are processed using browser events:

```
1  props::Props = [Prop('style','border: 1px solid black;')];
2
3  table::Display =
4    Table(props,
5      [ Row([],
6         [ Data(props,
7             HTML('<a href="edb:' + j + '">' + j + '</a>'))
8         | j ← (i*10)..(i*10)+10
9         ])
10     | i ← 0..10
11     ]);
12
13 act tableListener::BrowserListener {
14   BrowserEvent('50') → stopAll();
15   BrowserEvent(s::Str) → print[Str]('you pressed: ' + s);
16 }
17
18 edb ← AddBrowserListener(new tableListener);
19 edb ← Show('mytable',table);
```

Produces the following table:

0	1	2	3	4	5	6	7	8	9
10	11	12	13	14	15	16	17	18	19
20	21	22	23	24	25	26	27	28	29
30	31	32	33	34	35	36	37	38	39
40	41	42	43	44	45	46	47	48	49
50	51	52	53	54	55	56	57	58	59
60	61	62	63	64	65	66	67	68	69
70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89
90	91	92	93	94	95	96	97	98	99

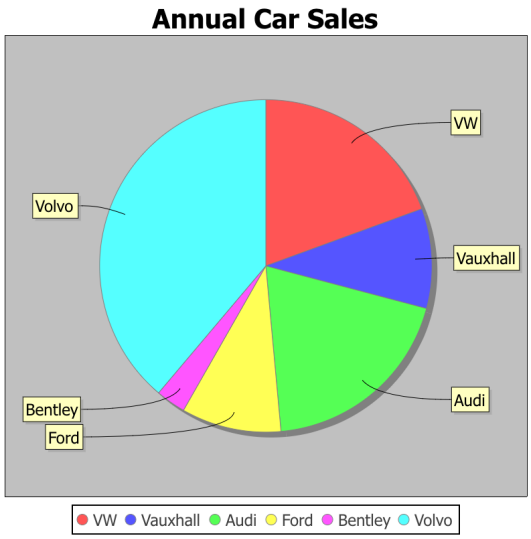
Clicking on any of the links (except 50) prints the number. Clicking on 50 stops the application. Subsequent clicks do nothing.

15.5. Pie Charts

Pie charts can show results. The values associated with each slice are shown as proportions of the pie:

```
pie::Display = Pie([],'Annual Car Sales',400,400,[
  Slice([],'VW',20),
  Slice([],'Vauxhall',10),
  Slice([],'Audi',20),
  Slice([],'Ford',10),
  Slice([],'Bentley',3),
  Slice([],'Volvo',40)]);

edb ← Show('mypie',pie);
```

which produces:

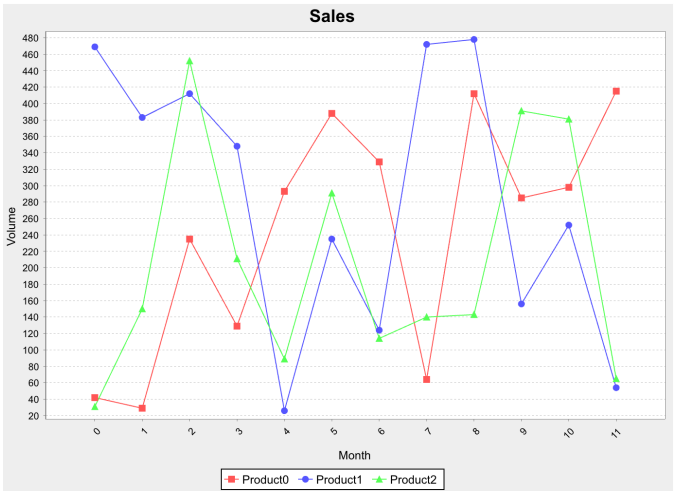


15.6. Line Graphs

Line graphs show data points:

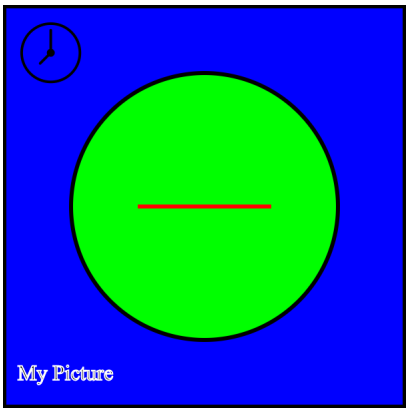
```
lgraph::Display =
  LineGraph([],'Sales','Month','Volume',700,500,[
    GLine([],'Product' + i,[
      LPoint([],month,random(500)) | month ← 0..12 ])
    | i ← 0..3 ]);

edb ← Show('mylines',lgraph);
```

producing:



15.7. Pictures

EDB provides a simple model of picture elements that map directly onto SVG. The following picture:



is produced by:

```
picture::Display = Picture(500,500,[
  Rectangle(100,100,300,300,
    'fill:rgb(0,0,255);stroke-width:3;stroke:rgb(0,0,0)'),
  Circle(250,250,100,
    'fill:rgb(0,255,0);stroke-width:3;stroke:rgb(0,0,0)'),
  Line(200,250,300,250,
    'stroke-width:3;stroke:rgb(255,0,0)'),
  Image(110,110,50,50,
    'https://img.icons8.com/ios/1600/clock.png'),
  Text(110,380,'My Picture',
    'stroke:rgb(255,255,255)')
]);

edb ← Show('mypicture',picture);
```

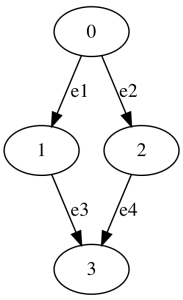
15.8. Graphs

EDB can display graphs using GraphViz. For example the following command:

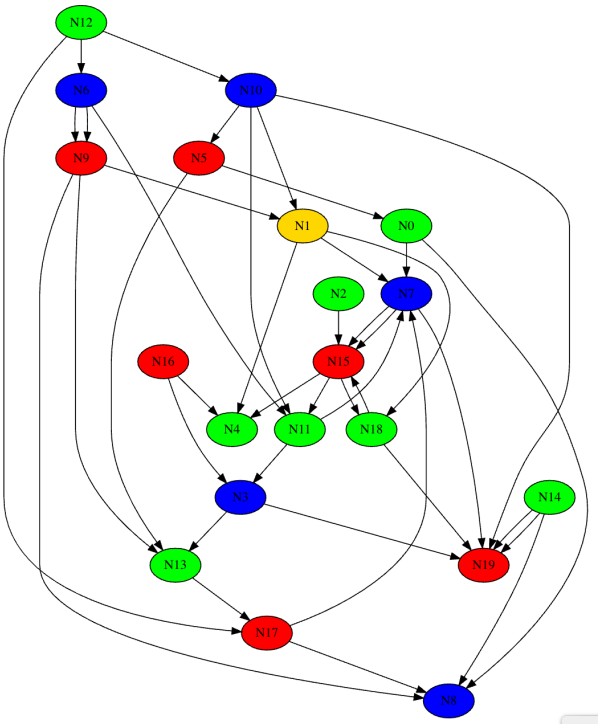
```
graph::Display = Graph([],[
  Node([],0,HTML('0')),
  Node([],1,HTML('1')),
  Node([],2,HTML('2')),
```

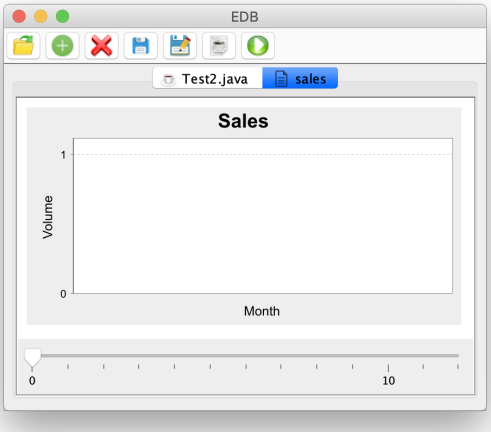
```
5    Node([],3,HTML('3'))],[
6    Edge([],0,1,HTML('e1')),
7    Edge([],0,2,HTML('e2')),
8    Edge([],1,3,HTML('e3')),
9    Edge([],2,3,HTML('e4'))]);
10
11 edb ← Show('mygraph',graph);
```

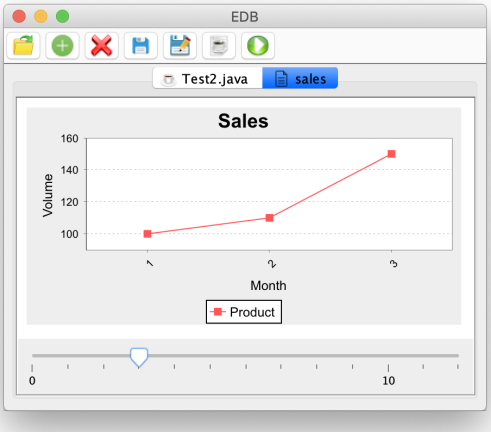
produces the following display:

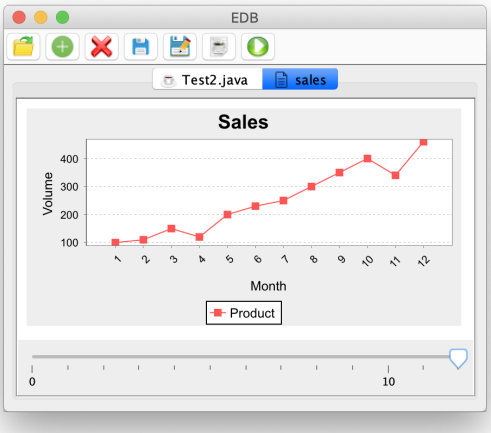


Properties in graphs, nodes and edges are used to control GraphViz. Figure 11 shows an example of EDB displays being used to create planar graphs and then colour the nodes. Any planar graph can be coloured using no more than 5 colours using Kempe's algorithm. The node properties are used to set the fill colour. The following is an example output:









15.9. Filmstrips

An EDB *filmstrip* is a list of displays that are shown on an EDB tab with a slider. When the slider is moved, the tab is updated to show the display at the appropriate index in the list. The following shows a sequence of line graphs where the slider can be moved to a month causing the graph to show the sales up to and including that month:

The filmstrip is generated as follows:

```
1 data Sale = Sale(Int,Int);
2
3 salesFigures::[Sale] = [
4   Sale(1,100),
5   Sale(2,110),
6   Sale(3,150),
```

```
export main;

import 'esl/lists.esl', 'esl/displays.esl';

nodeColour(n::Node)::Str = propValue(nodeProps(n),'fillcolor');
setNodeColour(n::Node,c::Str)::Node = setNodeProps(n,setProp(nodeProps(n),'fillcolor',c));
edgeBetween(e::Edge,n1::Node,n2::Node)::Bool = edgeTo(e,n1,n2) or edgeTo(e,n2,n1);
edgeTo(e::Edge,n1::Node,n2::Node)::Bool = edgeSource(e) = nodeId(n1) and edgeTarget(e) = nodeId(n2);
edgeOn(e::Edge,n::Node)::Bool = edgeSource(e) = nodeId(n) or edgeTarget(e) = nodeId(n);

kempe(graph::Display,colours::[Str])::Display =
  // Simple algorithm to colour a planar graph.
  // This is taken from http://www.cs.princeton.edu/~appel/Color.pdf
  // graph must be planar and has no more than 3v - 6 edges.
  case graph {
    Graph(props,[],edges)     → graph;
    Graph(props,nodes,edges) →
      let degreeLessThan5(n::Node)::Bool = length[Edge]([ e | e ← edges, ?(edgeOn(e,n)) ]) <= 5; in
      let n::Node = select1[Node](nodes,null[Node],degreeLessThan5); in
      let es::[Edge] = [ e | e ← edges, ?(edgeSource(e) = nodeId(n) or edgeTarget(e) = nodeId(n)) ]; in
      let g::Display = kempe(Graph(props,remove[Node](n,nodes),removeAll[Edge](es,edges)),colours);
      in case g {
           Graph(props,nodes,edges) →
             let adjacent::[Node] = [ n0 | n0 ← nodes, ?(exists[Edge](fun(e::Edge)::Bool edgeBetween(e,n,n0),es)) ]; in
             let usedColours::[Str] = [ nodeColour(n) | n ← adjacent ]; in
             let available::[Str] = removeAll[Str](usedColours,colours);
             in Graph(props,setNodeColour(n,head[Str](available)):nodes,edges+es);
      }
  }

mkPlanar(v::Int)::Display = Graph([],
  [ Node([Prop('style','filled'),Prop('fillcolor','')],i,HTML('N' + i)) | i ← 0..v ],
  [ Edge([],source,target,HTML('')) |
      i ← 0..random((3*v)-6),
      source ← [random(v)],
      target ← [random(v)],
      ?(source <> target) ]);

act main::Act {} {
  → edb ← Show('graph',kempe(mkPlanar(20),['red','green','blue','gold','deeppink']));
}
```

Figure 11: Graph Colouring

```
  Sale(4,120),
  Sale(5,200),
  Sale(6,230),
  Sale(7,250),
  Sale(8,300),
  Sale(9,350),
  Sale(10,400),
  Sale(11,340),
  Sale(12,460)];


lgraphs::[Display] = [
  LineGraph([],'Sales','Month','Volume',400,200,[
    GLine([],'Product',[ LPoint([],m,s)
    | Sale(m,s) ← take[Sale](salesFigures,month)
    ])
  ]) | month ← 0..13 ];

edb ← Filmstrip('sales',lgraphs);
```

15.10. Sequence Diagrams

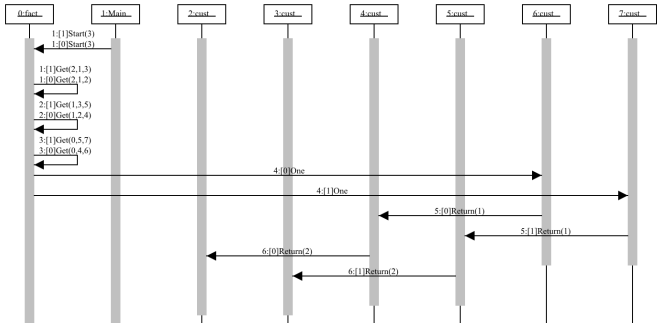
Sequence diagrams are a useful way of visualising the dynamic progress of an application. For example, sequential factorial is usually defined recursively so that each calculation of $!n$ waits on the stack until the calculation of $!(n-1)$ returns. Concurrent factorial can use actors to implement that stack frames that are waiting for results and allow any number of factorial calculations to occur at the same time. This is done as follows:

```
type Customer = Act { Value(Int,Int,Int); }
type Fact = Act{ Get(Int,Int,Int,Customer); }

act fact::Fact {
  Get(0,c) → {
    c ← Value(1);
  }
  Get(n,c) →
    let cc::Customer = new cust(n,c);
    in self ← Get(n-1,cc);
}

act cust(n::Int,c::Customer)::Customer {
  Value(m) → c ← Value(n*m);
}

act main::Customer {
  f::Fact = new fact;
```

```
  → {
    fact ← Get(3,self);
    fact ← Get(3,self);
  }
  Value(n) →  print[Str]('result = ' + n);
}
```

When `fact` is send a request to calculate a factorial, it tests whether the supplied value is `0`. If so, it immediately replies to the customer `c` supplying `1`. Otherwise, the supplied value is `n` and the factorial actor sends itself a message to calculate `!n-1` and to reply to a newly created customer.

Although this is recursive, it does not prevent interleaving of factorial requests. This can be seen if we create a sequence diagram for two different factorial calculations:



The diagram is produced by creating an EDB sequence diagram. Each factorial documents its progress as a sequence of messages tagged with an identifier that designates its thread of execution (shown as `[0]` and `[1]` on the diagram. Once the factorials are completed, a value of the form `Sequence(actors ,messages)` is created and displayed where `actors` contains a representation of all the actors involved in the calculation and `messages` is a list of messages sent between them. The ESL code is shown below:

```
type Customer = Act { export id::Int; Value(Int,Int,Int);}
type Fact = Act{ export id::Int; Get(Int,Int,Int,Customer);}

data M =
  GetValue(Int,Int,Int,Fact,Fact,Customer,Customer)
| ReturnValue(Int,Int,Int,Customer,Customer)
| Zero(Int,Int,Fact,Customer)
| Start(Int,Int,Customer,Fact);

actors::[Actor] = [];
messages::[M] = [];

addActor(behaviour::Str)::Int =
  grab(actors) {
    let id::Int = length[Actor](actors);
    in {
      actors := actors + [Actor(id,0,timeOut,behaviour)];
      id;
    }
  }

act fact::Fact {
  export id;
  id::Int = 0;
  → id := addActor('fact');
  Get(0,thread,time::Int,c::Customer) → grab(messages) {
    messages := messages + [Zero(thread,time,self,c)];
    c ← Value(thread,time+1,1);
  }
  Get(n::Int,thread,time::Int,c) → {
    let cc::Customer = new cust(n,c);
    in grab(messages) {
      messages := messages +
        [GetValue(n-1,thread,time,self,self,c,cc)];
      self ← Get(n-1,thread,time+1,cc);
    }
  }
}

act cust(n::Int,c::Customer)::Customer {
  export id;
  id::Int = 0;
  → id := addActor('cust');
  Value(thread,time::Int,m::Int) → grab(messages) {
    messages := messages +
      [ReturnValue(n*m,thread,time,self,c)];
    setDeath(id,time);
    c ← Value(thread,time+1,n*m);
  }
}

translate(m::M)::Message =
  case m {
    [] → [];
    Start(n,id,c,f) →
      Message(1,c.id,f.id,'['+id+']Start('+n+')');
    GetValue(n,id,t,src,tgt,srcC,tgtC) →
      Message(t,src.id,tgt.id,
        '['+id+']Get('+n+','+srcC.id+','+tgtT.id+')');
    ReturnValue(n,id,t,src,tgt) →
      Message(t,src,tgt.id,'['+id+']Return(' + n + ')');
    Zero(id,t,f,c) →
      Message(t,f.id,c.id,'['+id+']One');
  }

getActor(id::Int,as::[Actor])::Actor =
  case as {
    Actor(i,time,death,b):l when i = id →
      Actor(i,time,death,b);
    a:l → getActor(id,l);
  }

changeDeath(a::Actor,t::Int)::Actor =
  case a {
    Actor(i,birth,death,b) →  Actor(i,birth,t,b);
  }

setDeath(id::Int,time::Int)::Void =
  let oldActor::Actor = getActor(id,actors); in
  let newActor::Actor = changeDeath(oldActor,time);
  in grab(actors) {
   actors := subst[Actor](newActor,oldActor,actors);
  }

act main::Customer {
  export id;
  id::Int = 0;
  counter::Int = 0;
  count::Int = 1;
  f::Fact = new fact;
  numberOfFacts::Int = 2;
  computeFact(n::Int)::Void = grab(messages)) {
    messages := messages + [Start(n,counter,self,f)];
    counter := counter + 1;
    f ← Get(n,counter-1,1,self);
  }
  → {
    id := addActor('Main');
    computeFact(3);
```

```
    computeFact(3);
  }
  Value(threadId::Int,time::Int,n::Int) → {
    setDeath(id,time);
    if count = numberOfFacts
    then {
      edb ← Show('Factorial',Sequence(actors,[
        translate(m) | m ← messages
      ]));
      stopAll();
    } else count := count + 1;
  }
}
```

15.11. Combining Pictures

The `Display` data type allows pictures to be combines using trees. A tree consists of a collection of h-boxes and v-boxes that EDB will auto-layout. These are similar to HTML tables, but the size of the tree can be controlled. The following example shows how a filmstrip of trees can be constructed to create a simple predator-prey application. The predator-prey world consists of marked locations:

```
data Location =
  EmptyLoc
| PredLoc
| PreyLoc
| Rock;
```

Predators and prey, are located at points in the world and can be requested to move:

```
Act Predator {
  export getX::()→Int;
         getY::()→Int;
         at::(Int,Int)→Bool;
         getId::()→Int;
  Move;
}
Act Prey {
  export getX::()→Int;
         getY::()→Int;
         at::(Int,Int)→Bool;
  Move;
}
```

The board is a nested list of locations that is set up at random:

```
data Point          = Point(Int,Int);
pointX(p::Point)::Int = case p { Point(x,y) → x; }
pointY(p::Point)::Int = case p { Point(x,y) → y; }
board::[[Location]]  = [];
rocks::[Point]       = drop[Point](points,numOfPredators+1);
points::[Point]      =
  letrec generate(ps::[Point],n::Int)::[Point] =
    if n = 0
    then [];
    else
      let x::Int = random(width);
          y::Int = random(height);
      in if member[Point](Point(x,y),ps)
         then generate(ps,n);
         else Point(x,y):generate(Point(x,y):ps,n-1);
  in generate([],numOfPredators+1+numOfRocks);
```

Control of positions uses the following predicates:

```
onBoard(x::Int,y::Int)::Bool =
  (x >= 0) and (x < width) and (y >= 0) and (y < height);
onRock(x::Int,y::Int)::Bool =
  member[Point](Point(x,y),rocks);
legalPreyPos(x::Int,y::Int)::Bool =
  (not(onRock(x,y))) and onBoard(x,y) and
  not(exists[Predator](fun(p::Predator)::Bool
    p.at(x,y),predators));
legalPredatorPos(x::Int,y::Int)::Bool =
  (not(onRock(x,y))) and onBoard(x,y) and
  (not(exists[Predator](fun(p::Predator)::Bool
    p.at(x,y),predators))) and
  not(thePrey.at(x,y));
```

When they make a move, each of the actors in the world save their activity as a message on a global list. This will be used to create a graphical display of each snapshot of the world and eventually turned into a filmstrip:

```
data Message =
  MovePredator(Int,Int,Int)
| MovePrey(Int,Int);
messages::[Message] = [];
```

A predator moves towards the prey. The variable `turn` is used to achieve equitable movement:

```
act predator(id::Int,x::Int,y::Int)::Predator {
  export getX, getY, at, getId;
  getX()::Int = x;
  getY()::Int = y;
  getId()::Int = id;
  at(px::Int,py::Int)::Bool = (px = x) and (py = y);

  move(dx::Int,dy::Int)::Void = {
    if legalPredatorPos(x+dx,y+dy)
    then {
      x := x + dx;
      y := y + dy;
      messages := messages + [MovePredator(id,x,y)];
    } else { }
  }

  Move → grab(messages) {
    if turn = id
    then {
      turn := (turn + 1) % (numOfPredators+1);
      let dx::Int =
            if thePrey.getX() > x
            then 1;
            else if thePrey.getX() < x
            then - 1;
            else 0;
          dy::Int =
            if thePrey.getY() > y
            then 1;
            else if thePrey.getY() < y
            then -1;
            else 0;
      in {
        if (dy <> 0) and legalPredatorPos(x,y+dy)
        then move(0,dy);
        else if (dx <> 0) and legalPredatorPos(x+dx,y)
        then move(dx,0);
        else probably(50) {
          move(0,1-random(3));
        } else move(1-random(3),0);
      }
    } else {}
  }
}
```

The predators are created in a list:

```
predators::[Predator] = [
  new predator(p,pointX(nth[Point](points,p)),pointY(nth[
     Point](points,p)))
| p::Int ← 0..numOfPredators
];
```

The prey moves away from the predators. Unlike the predators, the prey can move diagonally:

```
act prey(x::Int,y::Int)::Prey {

  export getX, getY, at;

  getX()::Int = x;
  getY()::Int = y;

  at(px::Int,py::Int)::Bool = (px = x) and (py = y);
  dir(n::Int)::Int          = if n < 0 then -1; else 1;

  dirX::Int = 1 - random(3);
  dirY::Int = 1 - random(3);

  move(dx::Int,dy::Int)::Void = grab(messages) {
    if legalPreyPos(x+dx,y+dy)
    then {
      x := x + dx;
      y := y + dy;
      messages := messages + [MovePrey(x,y)];
    } else { }
  }

  predatorWins()::Void = {
    print[Str]('predator wins!');
    stop := true;
  }

  legalMoves()::[Point] = [ Point(dx,dy) |
    dx ← [-1,0,1],
    dy ← [-1,0,1],
    ?(dx <> 0 or dy <> 0),
    ?legalPreyPos(x+dx,y+dy)
  ];

  changeDir()::Void =
    case legalMoves() {
      Point(dx,dy):ps → { dirX := dx; dirY := dy; }
      []              → {}
    }

  Move → grab(messages) {
    if turn = numOfPredators
    then {
      turn := (turn + 1) % (numOfPredators+1);
      if legalMoves() = []
      then predatorWins();
      else if legalPreyPos(x+dirX,y+dirY)
      then move(dirX,dirY);
      else changeDir();
    } else {}
  }
}
```

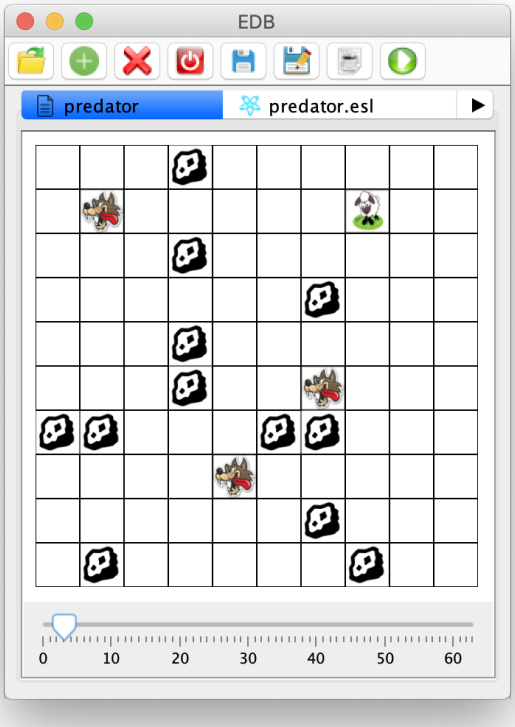
A single prey is created:

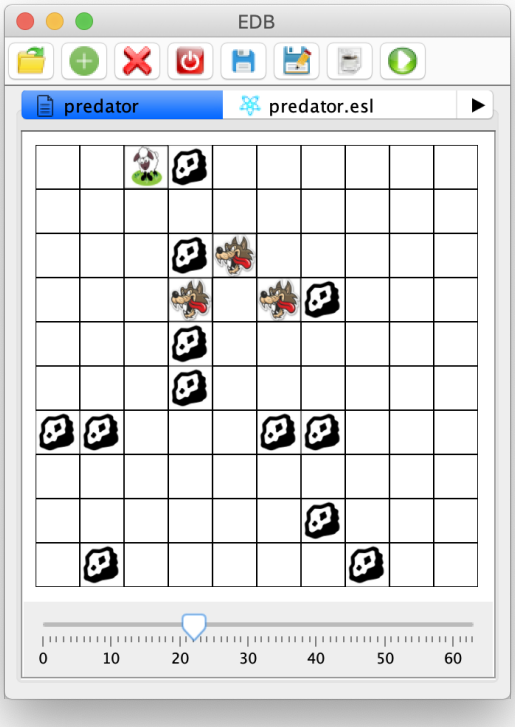
```
thePrey::Prey =
  let x::Int = pointX(nth[Point](points,numOfPredators));
      y::Int = pointY(nth[Point](points,numOfPredators));
  in new prey(x,y);
```

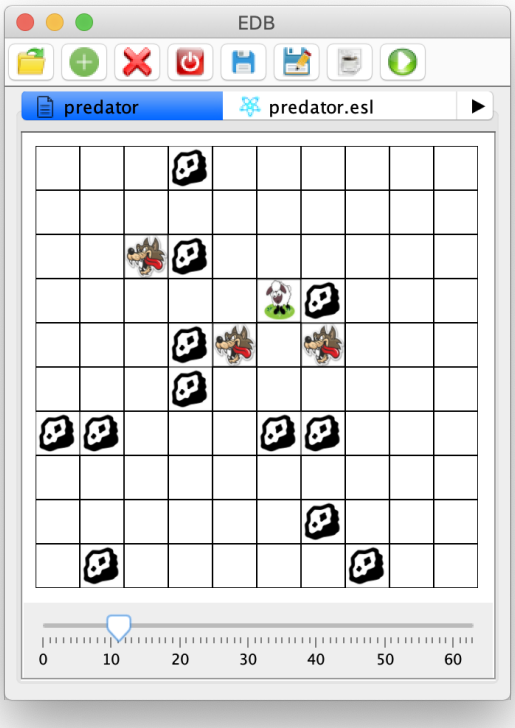
At this point we have sufficient to run the world and send all actors a `Move` message. A sequence of messages can be transformed into a snapshot where the occupied locations. The snapshots can be merged and the resulting composite snapshots merged using `mapBoard`:

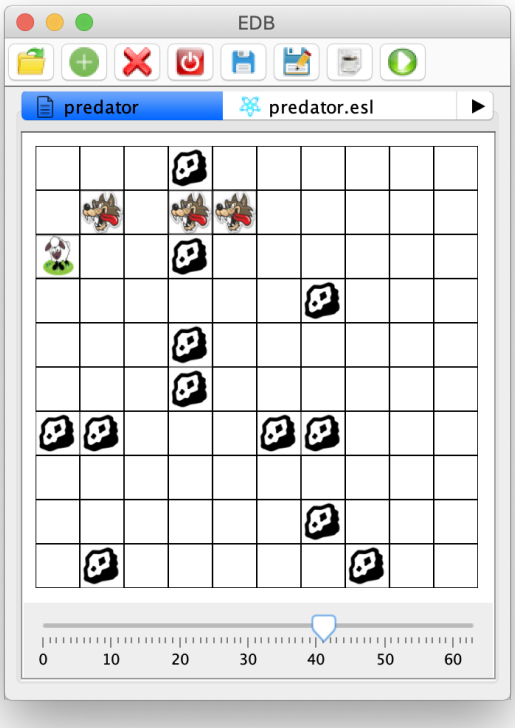
```
mapMessage(m::Message)::[[Location]] =
  case m {
    MovePredator(i,x0,y0) →
     [[ if (x=x0) and (y=y0)
        then PredLoc
        else if onRock(x,y)
        then Rock
        else EmptyLoc
      | x ← 0..width ]
     | y ← 0..height ];
    MovePrey(x0,y0) →
      [[ if (x=x0) and (y=y0)
         then PreyLoc
         else if onRock(x,y)
         then Rock
         else EmptyLoc
       | x ← 0..width ]
      | y ← 0..height ];
  }

mapBoard(b::[[Location]])::Display =
  let mapRow(row::[Location])::TreeElement =
        HBox([],[
          case l {
            PredLoc → predIcon;
            PreyLoc → preyIcon;
            EmptyLoc →space;
            Rock → rockIcon; } | l ← row ]);
  in Tree(width*size,height*size,
       VBox([],[
         mapRow(nth[[Location]](b,y)) | y ← 0..height ]));
```

The resulting list of displays is then presented as a filmstrip. The following shows steps in an example filmstrip:


EDB
predator
predator.esl
0
10
20
30
40
50
60


EDB
predator
predator.esl
0
10
20
30
40
50
60


EDB
predator
predator.esl
0
10
20
30
40
50
60


EDB
predator
predator.esl
0
10
20
30
40
50
60

16. Java Interface

An instance of such a class is created in ESL using `new` and registered with EDB using:

```
1 edb.display::Forall[T] (Str,T) → Void
```

for example: `edb.display[G]('TabName',graphics)`

17. Compilation

This section will describe the compilation process and the Java constructs produced by the ESL compiler.

18. Implementation

This section will describe the Java libraries that support ESL.

[1] Rajesh K Karmani, Amin Shali, and Gul Agha. Actor frameworks for the jvm platform: a comparative analysis. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 11–20. ACM, 2009.

A. Syntax and Type Checking

B. Displays

```
data Property = Prop(Str,Str);
type Props    = [Property];

propValue(props::Props,n::Str)::Str =
  case props {
    ps1 + [Prop(m,v)] + ps2 when m = n → v;
  }

hasProp(props::Props,n::Str)::Bool =
  case props {
    ps1 + [Prop(m,v)] + ps2 when m = n → true;
    props → false;
  }

setProp(props::Props,n::Str,v::Str)::Props =
  case props {
    ps1 + [Prop(m,v0)] + ps2 when m = n →
      ps1 + [Prop(n,v)] + ps2;
  }


data PictureElement =
  Rectangle(Int,Int,Int,Int,Str)
| Circle(Int,Int,Int,Str)
| Line(Int,Int,Int,Int,Str)
| Image(Int,Int,Int,Int,Str)
| Text(Int,Int,Str,Str);

data TreeElement =
   TreeNode(PictureElement)
|  VBox(Props,[TreeElement])
|  HBox(Props,[TreeElement])
|  Padding(Props)
|  LabelledTree(Str,TreeElement);

data LinePoint = LPoint(Props,Int,Int);
data Row       = Row(Props,[Data]);
data Data      = Data(Props,Display);
data Slice     = Slice(Props,Str,Int);
data GLine     = GLine(Props,Str,[LinePoint]);
data Message   = Message(Int,Int,Int,Str);
data Actor     = Actor(Int,Int,Int,Str);
data Node      = Node(Props,Int,Display);
data Edge      = Edge(Props,Int,Int,Display);

edgeSource(e::Edge)::Int  = case e { Edge(ps,s,t,d) → s; }
edgeTarget(e::Edge)::Int  = case e { Edge(ps,s,t,d) → t; }
nodeId(n::Node)::Int      = case n { Node(ps,id,d) → id; }
nodeProps(n::Node)::Props = case n { Node(ps,id,d) → ps; }
setNodeProps(n::Node,ps::Props)::Node =
  case n { Node(ps0,id,d) → Node(ps,id,d); }
edgeProps(e::Edge)::Props =
  case e { Edge(ps,s,t,d) → ps; }
setEdgeProps(e::Edge,ps::Props)::Edge =
  case e { Edge(ps0,s,t,d) → Edge(ps,s,t,d); }

data Display =
  HTML(Str)
| Table(Props,[Row])
| Pie(Props,Str,Int,Int,[Slice])
| LineGraph(Props,Str,Str,Str,Int,Int,[GLine])
| Picture(Int,Int,[PictureElement])
| Sequence([Actor],[Message])
| Graph(Props,[Node],[Edge])
| Tree(Int,Int,TreeElement);
```

C. List Operations

```
adjoin[T](x::T,l::[T])::[T] =
  if member[T](x,l)
  then l;
  else x:l;

select1[T](l::[T],d::T,p::(T)→Bool)::T =
  case l {
    [] → d;
    h:t when p(h) → h;
    h:t → select1[T](t,d,p);
  }

map[M,N](f::(M)→N,l::[M])::[N] =
  case l {
    [] → [];
    m:ms → (f(m)):map[M,N](f,ms);
  }

remove[T](v::T,l::[T])::[T] =
  case l {
    h:t when (h=v) → remove[T](v,t);
    h:t        → h:remove[T](v,t);
    []         → [];
  }

remove1[T](v::T,l::[T])::[T] =
  case l {
    h:t when (h=v) → t;
    h:t        → h:remove1[T](v,t);
    []         → [];
  }

length[T](l::[T])::Int =
  case l {
    h:t → 1 + length[T](t);
    [] → 0;
  }

flatten[T](lists::[[T]])::[T] =
  case lists {
    h:t → h+flatten[T](t);
    [] → [];
  }

count[T](x::T,l::[T])::Int =
  case l {
    h:t → if h=x then 1+count[T](x,t); else count[T](x,t);
    []  → 0;
  }

hasPrefix[T](list::[T],prefix::[T])::Bool =
  case list,prefix {
    l1,[] → true;
    x:list,y:prefix when x=y → hasPrefix[T](list,prefix);
    l1,l2 → false;
  }

nth[T](l::[T],n::Int)::T =
  case l {
    h:t    → if n = 0 then h; else nth[T](t,n-1);
    []  → throw[T]('cannot take nth element.');
  }

take[T](l::[T],n::Int)::[T] =
  if n = 0
  then [];
  else
    case l {
      h:t → h:(take[T](t,n-1));
      [] → throw[[T]]('cannot take element ' + n);
    }

drop[T](l::[T],n::Int)::[T] =
  if n = 0
  then l;
  else
    case l {
      h:t → drop[T](t,n-1);
      [] → throw[[T]]('cannot drop element ' + n);
    }

subst[T](n::T,o::T,l::[T])::[T] =
  case l {
    [] → [];
    h:t →
      if h = o
      then n:(subst[T](n,o,t));
      else h:(subst[T](n,o,t));
  }

head::Forall[T]([T])→T = fun(l::[T])::T
  case l {
    h:t → h;
    [] → throw[T]('cannot take the head of []');
```

```
  }

tail::Forall[T]([T])→[T] = fun(l::[T])::[T]
  case l {
    h:t → t;
    [] → throw[[T]]('cannot take the tail of []');
  }

isNil[T](l::[T])::Bool =
  case l {
    [] → true;
    l → false;
  }

member[T](e::T,l::[T])::Bool =
  case l {
    [] → false;
    x:xs when x = e → true;
    x:xs→ member[T](e,xs);
  }

reverse[T](l::[T])::[T] =
  case l {
    [] → [];
    x:xs → reverse[T](xs) + [x];
  }

exists[T](pred::(T)→Bool,l::[T])::Bool =
  case l {
    [] → false;
    x:xs when pred(x) → true;
    x:xs → exists[T](pred,xs);
  }

forall::[T](pred::(T)→Bool,l::[T])::Bool =
  case l {
    [] → true;
    x:xs when pred(x) → forall[T](pred,xs);
    x:xs → false;
  }

replaceNth[T](l::[T],n::Int,x::T)::[T] =
  case l {
    [] → throw[[T]]('cannot replace nth of []');
    h:t when n=0 → x:t;
    h:t → h:replaceNth[T](t,n-1,x);
  }

indexOf[T](t::T,l::[T])::Int =
  case l {
    [] → -1;
    h:l when h=t → 0;
    h:l → 1 + indexOf[T](t,l);
  }

select[T](p::(T)→Bool,l::[T])::[T] =
  case l {
    [] → [];
    h:t when p(h) → h:select[T](p,t);
    h:t → select[T](p,t);
  }

reject[T](p::(T)→Bool,l::[T])::[T] =
  case l {
    [] → [][T];
    h:t when p(h) → reject[T](p,t);
    h:t → h:reject[T](p,t);
  }

last[T](l::[T])::T =
  case l {
    [] → throw [T]('no last element of empty list');
    x:[] → x;
    h:l → last[T](l);
  }

butlast[T](l::[T])::[T] =
  case l {
    [] → [];
    [x] → [];
    h:l → h:butlast[T](l);
  }

occurrences[T](x::T,l::[T])::Int =
  case l {
    []:[T] → 0;
    h:t when h=x → 1 + occurrences[T](x,t);
    h:t → occurrences[T](x,t);
  }

filter[T](pred::(T)→Bool,l::[T])::[T] =
  case l {
    [] → [];
    h:t →
      if pred(h)
      then h:filter[T](pred,t);
      else filter[T](pred,t);
  }

foldr[M,N](map::(M)→N,op::(N,N)→N,empty::N,list::[M])::N =
  case list {
    [] → empty;
    h:t → op(map(h),foldr[M,N](map,op,empty,t));
  }

id[T]::(T)→T = fun(x::T)::T x;

sum(list::[Int])::Int =
  foldr[Int,Int](id[Int],fun(x::Int,y::Int)::Int x+y,0,list);

removeDups[T](list::[T])::[T] =
  case list {
    [] → [];
    x:l → x:removeDups[T](remove[T](x,l));
  }

prefixes[T](l::[T])::[[T]] =
  letrec pre(prefixes::[[T]],rest::[T])::[[T]] =
    if rest = []
    then prefixes;
    else pre(prefixes + [
           last[[T]](prefixes)+[head[T](rest)]
         ],tail[T](rest));
  in pre([[]],l);

takeWhile[T](p::(T)→Bool,l::[T])::[T] =
  case l {
    [] → [];
    t:l when p(t) → t:takeWhile[T](p,l);
    t:l → [];
  }

dropWhile[T](p::(T)→Bool,l::[T])::[T] =
  case l {
    [] → [];
    t:l when p(t) → dropWhile[T](p,l);
    l → l;
  }

removeAll[T](removables::[T],l::[T])::[T] =
  case removables {
    [] → l;
    t:r → removeAll[T](r,remove[T](t,l));
  }
```