Oscar
Functional, Actor based Programming Language

Manager: Ethan Adams EA2678
Language Guru: Howon Byun HB2458
System Architect: Jibben Lee Grabscheid Hillen Jlh2218
System Architect: Vladislav Sergeevich Scherbich VSS2113
Tester: Anthony James Holley AJH2186
Table of Contents

Introduction
Core Concepts
Immutability
Actor
Lexical Conventions
Identifiers
Keywords
Punctuation
Operators
Types
Programming Structure
Introduction

We will be implementing an Actor-Oriented Language. We all know there are difficulties associated with parallel computation, namely data collision when multiple threads access the same data. Most often, this requires explicitly defining locks and mutexes to control data access to resolve this issue, which adds significant mental and code overhead. The Actor model is a different approach to this problem that relies on the concept of message-passing between processes. The model simplifies the aforementioned difficulties of multiprocessing by abstracting explicit method calls, and relying on each “actor” to handle tasks and encapsulate mutable data. We would like to construct our language based off Erlang and Akka framework.

Core Concepts

Immutability

Values in Oscar are immutable like Haskell and Ocaml. This means that when a non-atomic value is declared, it cannot be reassigned and function parameters are passed by value. For instance, this is not allowed.

```java
int x = 5;
x = 4; // error
```

For primitive container types like list, map and set, they have an additional property that re-assigning values returns a new copy of the data structure with those updates applied.

```java
list<int> initList == [1, 2, 3, 4, 5];
list<int> changedList = (initList[3] = -4);
changedList == [1, 2, 3, -4, 5];
initList == [1, 2, 3, 4, 5];
```

Passing objects by value seems like a massive performance drain. After all, if a large container, say a list of five million entries, were passed in, then there is a huge memory overhead to copy such structure. List, and map and set that are implemented by it, in Oscar are implemented in as HAMT(Hash Array Mapped Trie), which guarantees practically O(1) runtime for append, update, lookup and slice operations\(^1\) and allows pass by value where value copied in is a pointer to one of the nodes in the list. In fact, persistent vectors in Clojure, yet another functional programming language, are implemented using this structure. As such, immutability can be enforced without sacrificing performance.

The main reason for implementing this is to enable painless parallel operations. Compared to traditional imperative languages like C++ and Java where variables are mutable by default and explicit synchronization methods are required to prevent data races, Oscar’s way of delegating actors to handle these mean that the user does not have to worry about synchronization primitives. Actors are discussed in further detail below.

\(^1\) http://hypirion.com/musings/understanding-persistent-vector-pt-1
With this said mutability is allowed within one context: actors. When variables are declared with \texttt{mut} keyword, reassignments are allowed. Non-container primitives can have their values changed and container types allow reassignments of the values held.

```lean
actor {
  mut int x = 10 // allowed within actors
  x = 14 // x == 14 now since x was declared to be mutable
  mut list<int> mutableList = [1, 2, 3, -4, 5]
  list<int> alteredList = (mutableList[3] = 5);
  mutableList == alteredList == [1, 2, 3, 5, 5]
}
```

Section on actors will provide more details as to why mutables are allowed within actors.

\textbf{Actor}

Actor is the basic unit of parallel processing in our language. A one sentence startup pitch for actors would be like a Java class that extends Runnable interface where all of the fields are private and the only way to affect it is through passing a specialized construct called \textit{messages}.

The most often used method of achieving concurrency is threading. Threads allow a process to efficiently perform multiple tasks at once (given necessary hardware/OS). Issue with most imperative languages is that, as mentioned above, data races must be handled explicitly, which means synchronization methods like locks and semaphores are needed that sacrifices performance.

Actors fixes this issue by preventing data races through encapsulation of data within its own unit of processing and strict enforcing of means of communication among them.\textsuperscript{2} As mentioned before, fields declared inside actors are private, meaning other actors and programs cannot access and change them. This prevents side effect mutations that often cause aforementioned problems. As such, the only means of affecting other actors is through message passing. Each actor defines the types of message it can process and corresponding action it will take based on that message. By doing so, the concern of data races are eliminated- one actor simply cannot implicitly modify states of others. This is why mutable variables allowed within actors. Since these variables cannot be modified by outside functions or actors unless message handles are specifically implemented, these mutations can happen safely without worrying about data race conditions.

Oscar elevates these concepts as primitives types as well as multiple operations to enable inter-actor communications.

\textsuperscript{2} https://en.wikipedia.org/wiki/Actor_model
Lexical Conventions

One of the main goals of Oscar is to be consistent and clear. In languages like Java, there are way too many syntactic sugars that lend itself to inconsistent coding conventions. For example, arrays contain value of length yet ArrayLists rely on size(). In Oscar, all contains use the same convention of initializing with \([\text{value1, value2, \ldots}]\) to maintain consistency and ease of development by preventing developers from having to lookup different APIs for achieving the same goal.

Identifiers

Identifiers consist of ASCII characters that must start with a letter then followed any combination of letters and numbers. In other words, it must follow this regex

\[ ['a'-'z' 'A'-'Z'] ['a'-'z' 'A'-'Z' 0-9]* \]

Whitespace

Any combination of tabs, spaces and new lines are allowed. Unlike Python, Oscar is not whitespace sensitive. Semicolons are used for statement separation

## Keywords

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Usage</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>mut</td>
<td>Declaration for mutable data&lt;br&gt;Inside actors only</td>
<td>mut int i = 5;&lt;br&gt;i = 4;</td>
</tr>
<tr>
<td>actor</td>
<td>Actor as a first class object</td>
<td>actor Worker {}</td>
</tr>
<tr>
<td>die</td>
<td>Kills the actor and all actors spawned by this actor</td>
<td>receive() =&gt; die();</td>
</tr>
<tr>
<td>spawn</td>
<td>Used to spawn actors and pools</td>
<td>spawn Worker(&quot;worker&quot;);&lt;br&gt;spawn pool(&quot;worker pool&quot;);</td>
</tr>
<tr>
<td>pool</td>
<td>Construct used to manage a collection of workers</td>
<td>pool p = spawn pool(&quot;worker pool&quot;);</td>
</tr>
<tr>
<td>receive</td>
<td>Message handler in actor</td>
<td>receive {&lt;br&gt;</td>
</tr>
<tr>
<td>sender</td>
<td>Source worker of the message</td>
<td>message(&quot;hi&quot;) =&gt; sender;</td>
</tr>
<tr>
<td>message</td>
<td>Signal between actors and pools</td>
<td>message(&quot;hi&quot;) =&gt; sender;</td>
</tr>
<tr>
<td>return</td>
<td>Return</td>
<td>def f(x: int) =&gt; () =&gt; return x;</td>
</tr>
<tr>
<td>if/else/else if</td>
<td>Conditionals</td>
<td>if (true) {} else if (false) {} else {}</td>
</tr>
</tbody>
</table>
| for/while | Loop construction | while(true) { } 
| --- | --- | --- 
| | | for (int i <- 0 to 5 by 1) { } 
| to/by | For loop range and loop increment/decrementer | for (int i <- 0 to 5 by 1) { } 
| break | Breaks out of the loop | while (true) { break; } 
| int | Integer data type | int x = 4; 
| double | Double data type used for floating point arithmetics | double d = 4.4; 
| char | Characters | char c = 'c'; 
| bool | Boolean data type. Lowercase true/false | bool lie = true; 
| string | String | string str = "hi"; 
| maybe/none/some | Optional type. “Null” should be wrapped with none in our language | maybe<int> m = some<int>(404); maybe<int> n = none; 
| list | List backed with HAMT\(^3\) for persistence with performance. | list<int> intList = [1, 2, 3]; 
| def | Functions | def f(x: int) => () = return x; 
| main | Main function | def main(args: list<string>) { } 
| tup | Tuple | tup<int, int> t = (1, 4); 

### Punctuation

<table>
<thead>
<tr>
<th>Punctuation</th>
<th>Usage</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>.</td>
<td>Decimal part indicator</td>
<td>41.17</td>
</tr>
<tr>
<td>,</td>
<td>List separator</td>
<td>list&lt;int&gt; a = [1, 2, 3, 4, 5];</td>
</tr>
<tr>
<td>;</td>
<td>Statement separator</td>
<td>int x = 4; int j = 4;</td>
</tr>
<tr>
<td>[]</td>
<td>List/set/map declaration</td>
<td>list&lt;int&gt; a = [1, 2, 3, 4, 5];</td>
</tr>
<tr>
<td>[]</td>
<td>List/set/map/tuple getter</td>
<td>a[1]; // 1</td>
</tr>
<tr>
<td>{}</td>
<td>Statement block</td>
<td>if (true) { println(&quot;hi&quot;); }</td>
</tr>
<tr>
<td>()</td>
<td>Conditional delimiter</td>
<td>if (false) { }</td>
</tr>
<tr>
<td>()</td>
<td>Tuple construction</td>
<td>tup&lt;int, int&gt; t = (1, 4);</td>
</tr>
<tr>
<td>()</td>
<td>Function arguments declaration</td>
<td>def func(arg: type) =&gt; {}</td>
</tr>
</tbody>
</table>

\(^3\) [http://lampwww.epfl.ch/papers/idealhashtrees.pdf](http://lampwww.epfl.ch/papers/idealhashtrees.pdf)
| `''` | Character literal | `char c = 'a';` |
| `""` | String literal | `String str = "hi";` |
| `<>` | List type | `list<string> strList = ["hi"];` |
| `/* */` | Multi-line comment | `/* hi
* hello
*/` |
| `//` | Single line comment | `// here` |

**Operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Usage</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
<td>Left</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>Left</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>Left</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>Left</td>
</tr>
<tr>
<td>%</td>
<td>Modulo</td>
<td>Left</td>
</tr>
<tr>
<td>=</td>
<td>Assignment</td>
<td>Right</td>
</tr>
<tr>
<td>==</td>
<td>Equal to</td>
<td>None</td>
</tr>
<tr>
<td>!</td>
<td>Logical negation</td>
<td>Right</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to</td>
<td>None</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>None</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
<td>None</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>None</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
<td>None</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
<td>Left</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise AND</td>
<td>Right</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Bitwise OR</td>
</tr>
<tr>
<td>^</td>
<td>Bitwise XOR</td>
<td>Right</td>
</tr>
<tr>
<td>Symbol</td>
<td>Description</td>
<td>Precedence</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td><code>&lt;&lt;</code></td>
<td>Bitwise left shift</td>
<td>Right</td>
</tr>
<tr>
<td><code>&gt;&gt;</code></td>
<td>Bitwise right shift</td>
<td>Right</td>
</tr>
<tr>
<td><code>~</code></td>
<td>Bitwise NOT</td>
<td>Right</td>
</tr>
<tr>
<td><code>()</code></td>
<td>Function invocation</td>
<td>Right</td>
</tr>
<tr>
<td><code>.</code></td>
<td>Method application</td>
<td>Left</td>
</tr>
<tr>
<td><code>:</code></td>
<td>Function argument type declaration</td>
<td>Left</td>
</tr>
<tr>
<td><code>=&gt;</code></td>
<td>Function return type declaration</td>
<td>Left</td>
</tr>
<tr>
<td><code>-&gt;</code></td>
<td>Map key-value pair</td>
<td>Left</td>
</tr>
<tr>
<td>`</td>
<td>&gt;`</td>
<td>Send a message to an actor</td>
</tr>
<tr>
<td>`</td>
<td>&gt;&gt;&gt;`</td>
<td>Broadcast a message to a pool</td>
</tr>
<tr>
<td><code>&lt;-</code></td>
<td>List comprehension assignment</td>
<td>Right</td>
</tr>
</tbody>
</table>

Precedence as would be in ocamlyacc:

```
,   <-- low
=
:  -> => <-
& & || & ^ |
< <= > >= == !=
<< >>
+ -
* / %
~
|> |>>
.   <-- high
```
Types
By default, everything is immutable in Oscar. This means that values cannot be re-assigned and container values return a new copy of the container when modified. Mutable values are allowed in one instance: as field variables of actors. This allows explicit, rather than implicit, changes of values through message passing.

Primitive Types

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Usage</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Integer data type</td>
<td>int x = 4;</td>
</tr>
<tr>
<td>double</td>
<td>Double data type used for floating point arithmetics</td>
<td>double d = 4.4;</td>
</tr>
<tr>
<td>char</td>
<td>Characters</td>
<td>char c = 'c';</td>
</tr>
<tr>
<td>bool</td>
<td>Boolean data type. Lowercase true/false</td>
<td>bool t = true; Bool f = false;</td>
</tr>
<tr>
<td>unit</td>
<td>Unit</td>
<td>def nothing() =&gt; () = println();</td>
</tr>
</tbody>
</table>

Integer “`-?[0-9]+`”
An integer type is an immutable, signed 4 byte value of digits

```
int x = 5;
x = 4; // ERROR
```

Integers can be casted into doubles using “.asDouble()” method.

```
int x = 5;
x = 5.; // ERROR
x.asDouble() == 5.;
```

Floating Point Numbers “`-?[0-9]+\.[0-9]*`”
Oscar uses double for floating point numbers. A double is a signed 8 byte double-precision floating point data like that in Java. Fractional part can be omitted.

```
double x = 5.4;
double y = -3.4;
double z = 3.;
```

Implicit castings are not allowed

```
double error = 3; // ERROR
int error2 = 2.4; // ERROR
```
Like integers, doubles can be casted down using ".asInt()" method, which drops the fractional part of the value.

```java
double a = 3.0; // ERROR
a == 3; // ERROR
a.asInt() == 3;
```

Arithmetic operations with double always modify the return type to double

```java
5 / 2 == 2;
5 / 2. == 5./2 == 5.0/2. == 2.5;
```

**Character**

```
"\((a'-'z'|A'-'Z')|\'|\'|\'|n'|r'|t'|''|''|''\)"
```
A character is a 1 byte data consisting of ASCII values.

```java
char c = 'c';
c - 'f' == -2; // true
```

**Boolean** "true"|"false"

Boolean values take 1 byte and can be either true or false. Other data types cannot be casted as boolean. For example, none cannot be used as false

```java
bool t = true;
```

**Unit**

Unit type is purely for functions that do not return any value denoted by ()

```java
def println(s: string) => () = { }
```

### Non-Primitive Types

It is worth noting that Oscar treats list, set and map as “container primitives”

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Usage</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>string</td>
<td>String</td>
<td>string str = “hi”</td>
</tr>
<tr>
<td>maybe/none/some</td>
<td>Optional type. “Null” should be wrapped with none in our language</td>
<td>maybe&lt;int&gt; m = some&lt;int&gt;(404); maybe&lt;int&gt; n = none;</td>
</tr>
<tr>
<td>list</td>
<td>List backed with HAMT(^4) for persistence with performance.</td>
<td>list&lt;int&gt; intList = [1, 2, 3]; list&lt;char&gt; charList = list&lt;char&gt;[];</td>
</tr>
<tr>
<td>set</td>
<td>Set of unique values</td>
<td>set&lt;int&gt; tSet = [1, 2, 3]</td>
</tr>
<tr>
<td>map</td>
<td>Map of key value pairs</td>
<td>map&lt;int, string&gt; tMap = [0 -&gt; “hi”]</td>
</tr>
<tr>
<td>tup</td>
<td>Tuple</td>
<td>tup&lt;int, int&gt; t = (1, 4)</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>actor</th>
<th>Actor as a first class object</th>
<th>actor Worker {}</th>
</tr>
</thead>
<tbody>
<tr>
<td>message</td>
<td>Signal between actors and pools</td>
<td>message(&quot;hi&quot;)</td>
</tr>
<tr>
<td>pool</td>
<td>Construct used to manage a collection of workers</td>
<td>pool p = spawn pool(&quot;worker pool&quot;);</td>
</tr>
<tr>
<td>def</td>
<td>Functions</td>
<td>def f(x: int) =&gt; () = return x;</td>
</tr>
</tbody>
</table>

**String** `"\[char]*"`  
A string is just a character list like found in C++. They come with a few built-in functions for convenience. Since strings are list backed, built-in operations for lists apply to strings as well. Since explicit type conversion does not exist in Oscar, primitives must be casted by calling “.asString()” method.

```plaintext
string hi = "hello world";
hi[0] == 'h';  
me == "hello world"; // == on strings compare values  
me.size() == 11;
me.substring(0, 6); // “hello”  
me.filter(x:char => char = x != 'l') == "heo word";
println("hi :" + 5); // ERROR
println("hi :" + 5.asString()); // "hi :5"
```

**Optional**  
Oscar supports optional values for NullPointerException-safety.

```plaintext
maybe<int> perhaps = some<int>(0);  
maybe<int> no = none;
```

Type optional has a very simple api for safely interacting with values

```plaintext
perhaps.isDefined() == true;  
no.isDefined() == false;  
no.isNone() == true;  
perhaps.get() == 0;  
no.get() == None;  
perhaps.map(x:int => double = x + 5.4); // some<double>(5.4);
```

For example,

```plaintext
def printMaybe(x: maybe<int>) => () = {
  if (x.isDefined()) {
    println(x.get());
  }
}
```
**List**

A list is an immutable collection of values of the same type.

```cpp
list<int> intList = list<int>[]; // empty list of type int
list<double> listSizeTen = list<double> (10, 0.0);
    // size 10 of 0.0's
list<int> intList = [1, 2, 3, 4, 5]; // list literal
list<double> intList = [1, 2.0, 3, 4, "5"]; // ERROR
```

Since everything without `mut` keyword is immutable, assignment just returns a new list with assignment applied

```cpp
list<int> changedList = (initList[3] = -4)
changedList == [1, 2, 3, -4, 5]
initList == [1, 2, 3, 4, 5]
```

If declared with `mut`, lists behave like vectors in C++

```cpp
mut list<int> mutableList = [1, 2, 3, -4, 5]
list<int> alteredList = (mutableList[3] = 5);
mutableList == alteredList == [1, 2, 3, 5, 5]
```

Type list has a few built-in functions. Here are some

```cpp
list<int> exam = [1, 2, 3, 4, 5];
exam[0] == 1;
exam.slice(0, 4) == [1, 2, 3];
exam.size() == 5;
list<int> prepend = exam.prepend(0); // [0, 1, 2, 3, 4, 5];
list<int> append = exam.append(6); // [1, 2, 3, 4, 5, 6];
list<int> popFront = exam.popFront(); // [2, 3, 4, 5];
list<int> popBack = exam.popBack(); // [1, 2, 3, 4];
exam.contains(0); // true
exam.foreach(x:int => unit = {
    println(x) // prints elements of exam
});
exam.filter(x:int => int = x % 2 == 0); // [2, 4]
exam.map(x:int => int = x + 2); // [3, 4, 5, 6, 7]
exam.foldLeft((x:int, y:int) => int = x + y); // 15
```

List comprehension can also be used to declare lists.

```cpp
list<int> comp = [int i <- 0 to 10 by 2]; // [0, 2, 4, 6, 8]
[int i <- 0 to 5 by 2].map(i:int => int = {
    return i + 4;
}).reduce((x:int, y:int) => int = x + y) // == 14
```
Set/Map
A set is a collection of distinct elements of the same type.

```cpp
set<int> intSet = [1, 2, 3, 4, 5];
set<int> dupIntSet = [1, 2, 3, 4, 5, 1];
set<int> intSetTwo = [1, 2, 3, 7, 8, 9];
intSet == dupIntSet;
intSet.contains(3) == true;
intSet.intersect(intSetTwo); // [1, 2, 3]
intSet.union(intSetTwo); // [1, 2, 3, 4, 5, 7, 8, 9]
intSet.diff(intSetTwo); // [4, 5]
intSetTwo.diff(intSet); // [7, 8, 9]
intSet.add(8); // intSet stays, returns a new set with 8 added
```

Similarly, maps are collections of key value pairs.

```cpp
map<int, int> test = map<int, int>[];
map<int, double> tMap = [0 -> 1.1, 4 -> 5.3, -3 -> 5.3];
map<int, double> tMapCpy = [0 -> 1.1, 4 -> 5.3, -3 -> 5.3];
map<int, double> errMap = [0 -> 1]; // ERROR as int != double
map<int, double> errMap2 = [0 -> 1.2];
map<int, double> union = [2 -> 1.2];
tMap.size() == 3;
tMap == tMapCpy; // compares all key-value pairs
tMap.union(union) == [0 -> 1.1, 4 -> 5.3, -3 -> 5.3, 2 -> 1.2];
tMap.union(errMap2); // error as key 0 exist in both maps
tMap.contains(0) == true;
tMap.add(4 -> 5.3); // tMap stays, returns a new map
tMap.remove(0); // tMap stays, returns a new map without ket 0
```

Get operations are protected through use of optionals

```cpp
tMap[0] == some<int>(1.1);
tMap[2] == none;
```

Like lists, sets and maps can be declared with `mut` keyword inside `actor` context to make them mutable.

```cpp
map<int, double> tMap = [0 -> 1.1, 4 -> 5.3, -3 -> 5.3];
map<int, double> unionMap = [2 -> 1.2];
tMap.add(5 -> 4.3);
tMap == [0 -> 1.1, 4 -> 5.3, -3 -> 5.3, 5 -> 4.3];
tMap[-3] = 3.4; // [0 -> 1.1, 4 -> 5.3, -3 -> 3.4, 5 -> 4.3];
tMap.remove(4);
tMap == [0 -> 1.1, -3 -> 5.3, 5 -> 4.3];
```
tMap = tMap.union(unionMap);
   // [0 -> 1.1, 4 -> 5.3, -3 -> 5.3, 2 -> 1.2];

**Tuple**

Tuples are like structs. These can be used for conveniently wrapping multiple values of different types.

```cpp
tuple<int, string> test = tuple<int, string>[1, “23”];
tup[0] == 1;
```

Tuples can be destructed into values on the left hand side;

```cpp
int i, string s, double d = tup;
i == 1;
s == “23”;
d == 2.3;
```

**mut tuples** can have their contents changed but their types are immutable

```cpp
mut tuple<int, string, double> mutTup = (1, “23”, 2.3);
tup[0] = 2;
mutTup == (2, “23”, 2.3);
tup[0] = 2.4; // ERROR
```

**Actor**

Actors refer to units of concurrency processing defined in the Actor Model. Oscar elevates these into first class constructs.

Actors are special. They can hold functions and variables, declared using mut keyword. Also, all actors must define receive function to handle messages. The spawn keyword is used to create them. As explained in “Core Concepts” section, values inside actors are private. As such, explicit handling of messages is the only way to affect actors states.

```cpp
actor Worker(initValue: int) { // immutable initializer value
   mut int x = 10 // allowed within actors
   x = 14 // x == 14 now since x was declared to be mutable
   receive { // all actors must have receive method defined
      | messageType1 => { } // do something 1
      | end => die() // used to kill this actor and
         // all other workers declared in this
   }
}
actor worker = spawn Worker(3);
```

---

Pool
pool manages a group of actors. When messages are sent into a pool, they are
distributed in a round-robin fashion so that many actors can work concurrently. Like
actors, pools are created using spawn keyword.

    pool workerPool = spawn pool(Worker(5), 10)
    // puts ten workers into a pool

Message
Messages constructs used in inter-actor communications. They are structurally similar
to tuples but hold extra data and operations. |> operator is used to send a message
to actors and pools

    message helloMessage(prefix: string, suffix: string)
    helloMessage("Hello", "World!") |> worker
    helloMessage("everyone gets a", " message!") |>> workerPool
    // list of messages can be mass broadcasted
    [msg1, msg1, msg1] |>> workerPool

sender keeps track of the source actor of a message. This can be used to bounce
back a message to whomever sent it.

    helloMessage("hi", "there") |> sender

Function
Functions are declared with def keyword. In Oscar, functions are declared using
arrow syntax like in ECMAScript 2016 and how most lambda functions are declared.

    def <identifier>(arg: type, arg2: type2 ...) => <return type> = {
        return;
    }
    def addTwoNums(a: int, b: int) => int = a + b;
    int b = addTwoNums(a, 6);
    b == 11;

Since functions are first class objects in Oscar, they can be passed into functions as
well.

    def apply(f: (double) => double, a: double) => double = {
        return f(a); // return is needed for multi-line functions
    } // multi-line functions are surrounded with brackets
    def d = apply((x:double) => double = x * 2, 44.5) == 89.0

No argument functions are declared like this:

    def sayHi() => () = println("hi!"); // void function
sayHi(); // prints “hi!”

**Program Structure**
A typical Oscar program is composed of functions, actors and message declarations. This section talks about how execution flows in Oscar.

**Logics and Relations**
Oscar has traditional and familiar relational operators for non-container primitive types.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Usage</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>None</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
<td>None</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>None</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
<td>None</td>
</tr>
</tbody>
</table>

4 > 3 == true;
3.4 < 2.4 == false;

Oscar does not allow implicit casting of values. For example, this expression would
3.4 < 2;
yield an error because value 2 is of type int but 3.4 is of type double. Explicit casting via methods like “asInt()” and “asDouble()” must be invoked for logical comparisons across types.

Relation operations also extend to container primitives. For instance, comparison of
setA > setB
will evaluate to true if setA contains all elements of setB. Similarly, this means that every key-value pairs in one map would be contained in the other.

Oscar’s logical operations also follow traditionally used syntaxes. Though functionality

<table>
<thead>
<tr>
<th>Operator</th>
<th>Usage</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Equal to</td>
<td>None</td>
</tr>
<tr>
<td>!</td>
<td>Logical Negation</td>
<td>Right</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal to</td>
<td>None</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>Logical AND</td>
<td>Left</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Remains largely the same, it is worth pointing out that Oscar does not support implicit conversions of values into boolean values. For instance, type `none` cannot be evaluated as `false`. Also worth noting that the equality operator `==` extends to container types where every single value is compared.

**Control Flow**

In Oscar, statements are executed from top to bottom. However this order of execution can be modified using control flow statements.

**Branching**

Non-loop branching in Oscar is handled as follows:

```java
if (condition) {
    statements block 1
} else if (another condition) {
    statements block 2
} else {
    statements block 3
}
```

If the first condition is satisfied, first statement is executed. If, however, the second condition is satisfied then the `else if` branch is taken. If none of these conditions pass, then statements in `else` branch is executed.

Conditions have to be boolean expressions. These can be constructed either by passing a boolean value types as conditions or through relational and logical operations explained previously.

**Loop**

Oscar supports both `while` and `for` keywords for looping. For loops allow iteration over a range of values at increments defined by the value that comes after `by`.

```java
for (int i <- 0 to 10 by 1) {
    statements block;
}
```

While loops, on the other hand, repeatedly executes a block of code until the condition is broken.

```java
while (condition) {
    statements block;
}
```
Like for branches, `condition` has to be a boolean expression. Loops can be forcefully exited when `break` is called. This returns the stage of execution to the immediate outside score. For example,

```java
while (condition) { // outer loop
  while (condition) { // inner loop
    break;
    println("hello");
  }
  println("hi");
}
```

Because the inner loop is broken before its print statement, it will exit out and execute the statement that comes after the while block and repeatedly print ("hi") as if the inner loop did not exist.

**Actor, Pool & Message**

Messages are defined as globals so that they can be used by actors. When multiple Oscar programs are linked together, messages are shared across so that unnecessary redeclaration of messages are prevented. They must be declared before any actor declarations.

```java
message message1(msg: string, payload: double)
message message2(prefix: string, suffix: string)
message message3(prefix: string, suffix: string, extra: string)

actor Actor() {} ...
```

As mentioned before, all actors need to have `receive` handler defined to prevent compilation failures. Inside `receive` a series of pattern matches need to be defined. Values contained in each message are destructured and available to its corresponding handler.

```java
receive = {
  message1 => {
    println(msg + " " + payload.asString());
  }
  message2 => {
    println(prefix + " " + suffix);
  }
}
```
To prevent an extraneous amount of actors from lingering around, each actor can call `die()` method to clean up and kill its thread of execution. Since this inherently changes the state of the actor, this must be explicitly performed by setting a message handler.

```plaintext
receive = {
    | message1 => {
        println(msg + " " + payload.asString());
    }
    ...
    | kill => {
        die();
    }
}
```

Once actor types are declared, a pool to manage these actors can be declared.

```plaintext
pool<Actor> newPool = spawn pool<Actor>(10); // ten actor pool
```

**Executable**

Every executable Oscar program needs to define a function named `main()` which takes in a list of strings and returns nothing. This is a reserved keyword that denotes the first function that gets executed. When multiple Oscar programs are linked together, the compiler will throw an error upon encountering multiple `main()` functions.

```plaintext
def main(args: list<string>) => () = {}
```

Here is a sample Oscar to approximate the value of pi that incorporate these features

```plaintext
message start() // empty message
message end()
message work(start: int, numElems: int)
message result(value: double)
message piApproximation(pi: double)

actor Worker {
    let calcPi = (start: int, numElems: int) : double = {
        return [i <- start to (start + numElems)].map(i => {
            return 4.0 * (1 - (i % 2) * 2) / (2 * i + 1)
        }).reduce((x, y) => x + y) // list operations
    }

    let receive => {
        | work(start: int, numElems: int) => {
```
result(calcPi(start, numElems)) |> sender
}
}

actor Listener(name: string) {
  let receive => {
    | piApproximation(value: double) => {
      print("value of pi is approximately \:" + value)
      end() |> sender
    }
  }
}

// Master worker for pi approximation
actor Master(numWorkers: int, numMsgs: int, numElems: int) {
  mut pi = 0.0;
  mut numResults = 0;
  mut pool<Worker> workerPool = spawn pool<Worker>(numWorkers);
  actor listener = spawn Listener("pi listener");

  let receive => {
    | start => {
      list<work> msgList = [int i <- 0 to numMsgs by 1].map(i => {
        work(i * numElems, numElems);
      })
      msgList |>> workerPool;
    }
    | result => {
      pi = pi + value;
      numResults = numResults + 1;
      if (numResults == numMsgs) {
        piApproximation(pi) |>> listener;
      }
    }
    | end => {
      die();
    }
  }
}

let main => () = { // main is a keyword
  spawn Master(5, 10000, 10000);