Oscar

```
185 shift/reduce conflicts, 770 reduce/reduce conflicts.
```


Ethan Adams — *Project Manager*

Howon Byun — *Language Guru*

Jibben Hillen — *System Architect*

Vladislav Scherbich — *System Architect*

Anthony Holley — *Tester*

Concepts

Actor-Oriented Programming

The Actor Model allows lock-free concurrency

Actors can:

- Send messages to other actors
- Create new actors
- Designate the behavior to be used upon receipt of the next message
- CANNOT manipulate other actors' states

Similar Languages/Frameworks:

- Erlang
- Akka

Functional Programming

- Values in Oscar are immutable
  - **unless declared as mutable within actors as internal states of actors cannot be manipulated*

- List, map and set are immutable

- Reassigning values to immutable lists/maps/sets returns a new copy

Syntax

Syntax - Fundamentals

Operators

Basic Operators

Comparators

Logical Connectives

Function Syntax

Send & Broadcast

Map Key-Value Pair

```
( = + - * / % )

== != <= >= < >

&& ! ||

() : =>

|> |>>

->
```

Comments

```
/*
   comments
   are
   multi-line
 */
```

Syntax - Fundamentals

if-else

```
if (A == B) {
  Println("true");
} else {
  Println("false");
}
```

Immutable Declarations

```
int x = 42;
double d = 42.0;
string str = "42";
list<int> l = list<int>[42, 42, 42];
set<int> s = set<int>[42, 4, 2];
map<int, double> m =
        map<int, double>[42 -> 42.0, 4 -> 2.0];
```

Syntax - Messages

Declaring Messages

```
message hello()
message printThis(content: int)
message printThese(thing1: string, thing2: int)
```

Sending Messages

```
message<hello>() |> stranger;
message<printThis>(42) |> printer;
message<printThese>("42", 42) |>> printerPool;
```

Syntax - Functions

Top-Level Syntax

```
def identifier(arg: typ1, arg2: typ2 …) => return typ = {
  return retArg;
}
```

Lambda Syntax

```
func f = (i: int) => int = i + 1;
```

Using Lambda Syntax

```
def apply(f: [(int) => int], input: int) => int = {
  return f(input);
}
```

Syntax - Functions

Top-Level Syntax

```
def identifier(arg: typ1, arg2: typ2 …) => return typ = {
  return retArg;
}
```

Lambda Syntax

```
func f = (i: int) => int = i + 1;
```

Using Lambda Syntax

```
def apply(f: [(int) => int], input: int) => int = {
  return f(input);
}
```

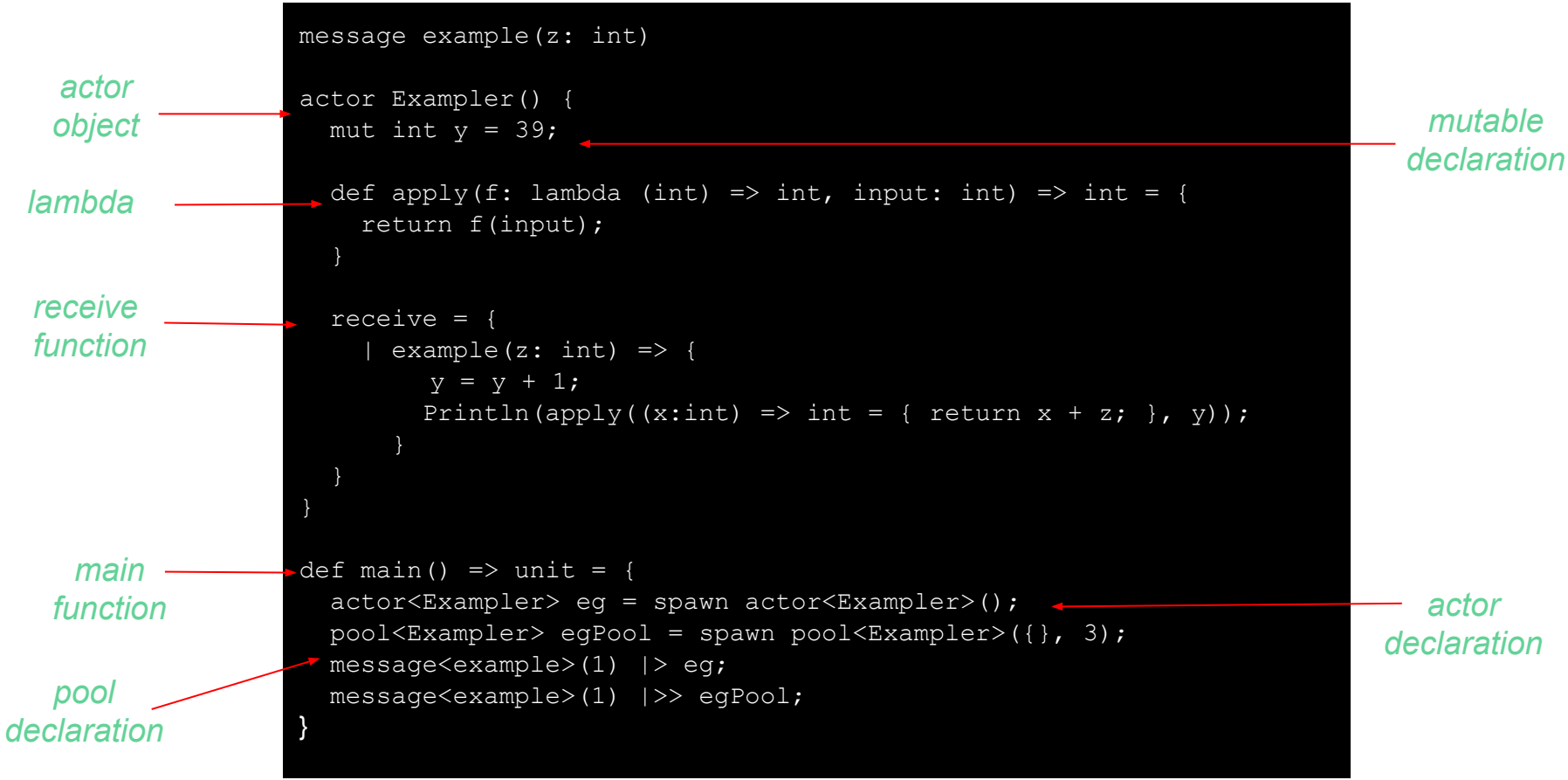
Syntax - Functions: Sample Code

```
def main() => unit = {
  list<int> intList = list<int>[1, 2, 3, 4, 5];

  int a = FoldLeft((x:int, y:int) => int = {
    return x + y;
  }, 10, intList));

  /* a == 10 + 1 + 2 + 3 + 4 + 5 == 25 */
}
```

Syntax - Functions: Sample Code

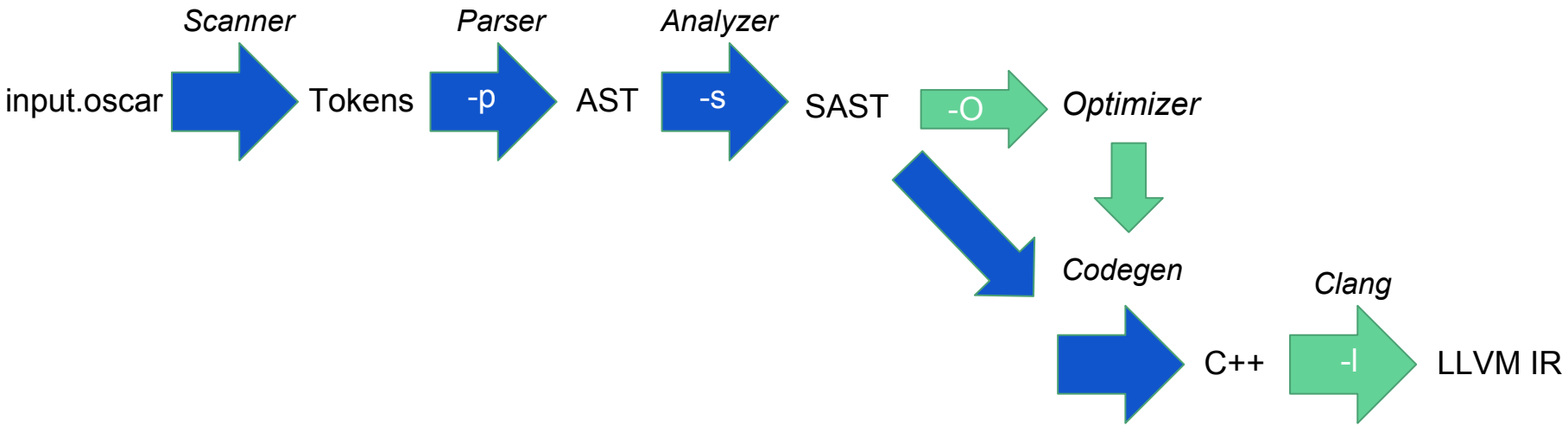
```
/* generate a filter function */
def genFilter(div: int) => [(int) => bool] = {
  func f = (x:int) => bool = (x % div == 0);
  return f;
}


def main() => unit = {
  list<int> intList = list<int>[1, 2, 3, 4, 5];

  func filt2 = genFilter(2);

  list<int> l = Filter(filt2, intList);
  /* l is [2, 4] */
}
```

Syntax - Actor

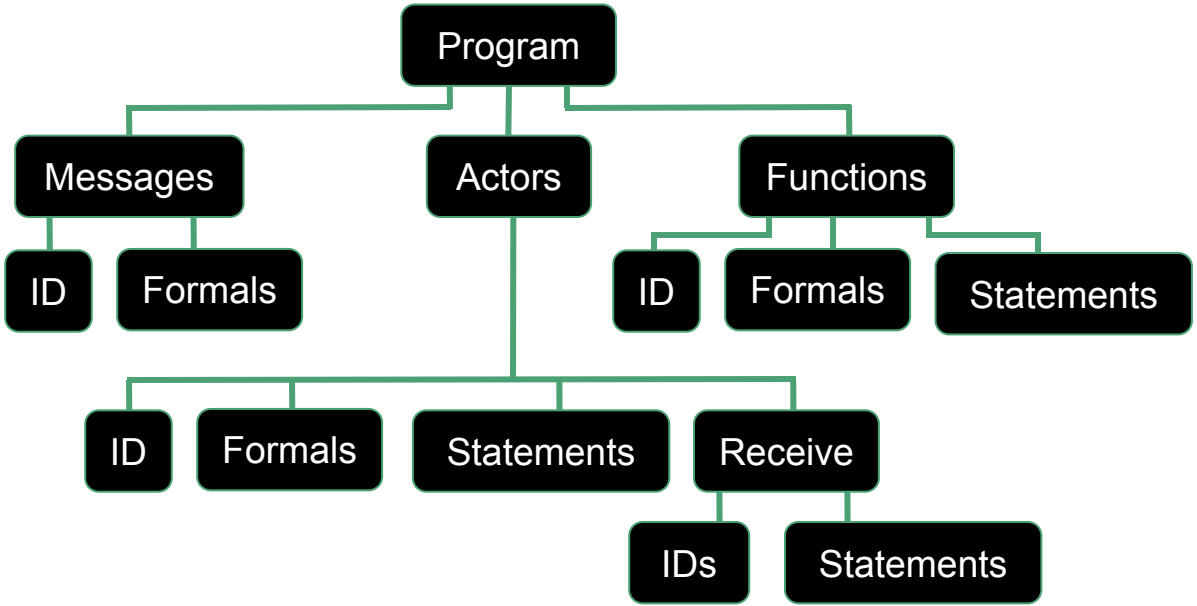


Implementation

Implementation - Stages of Compilation



Implementation - AST



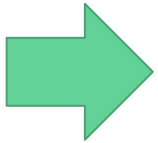
Implementation - The Optimizer

Every Oscar program is run through a two-pass optimizer:

- Unused Immutable primitive value declarations are removed
- Binary and unary operators with optimized expressions are evaluated to literal
- Unused code blocks are removed
- Unused function and variable declarations are removed
- if/else conditionals are evaluated

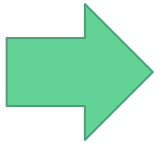
Implementation - The Optimizer

```
def main() => unit = {
  Println(3 + 39);
}
```



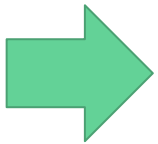
```
int main ()
{
      Println(42);
      return 0;
}
```

```
def main() => unit = {
  int x = 1;
  int y = 2;
  Println(x);
}
```



```
int main ()
{
      int x = 1;
      Println(1);
      return 0;
}
```

```
def main() => unit = {
  if(false) {
    Println(“false”);
  }
}
```



```
int main ()
{
      return 0;
}
```

Implementation - Builtin Functions: Print/Casting

- `Println(“Primitive/Container”) => unit`
- `AsInt(double) => int`
- `AsDouble(int) => double`
- `AsString(int/double/char/bool/list/set/map) => string`

Implementation - Builtin Functions: List-only

- Append(item, list<a>) => list<a>
- Prepend(item, list<a>) => list<a>
- PopFront(list<a>) => list<a>
- PopBack(list<a>) => list<a>
- MergeFront(list1<a>, list2<a>) => list<a>
- MergeBack(list1<a>, list2<a>) => list<a>
- Reverse(list<a>) => list<a>
- list<a>[0] => a

Implementation - Builtin Functions: Set-only

```
- Union(set1, set2) => set
- Diff(set1, set2) => set
- Intersection(set1, set2) => set
- Subset(set1, set2) => set
```

Implementation - Builtin Functions: Applied Functions

```
- ForEach(function a=>unit, list/set/map<a>) => unit
- FoldLeft(function (a, b)=>b, accumulator<b>, list<a>) => b
- Filter(function a=>bool, list/set/map<a>) => list/set/map<a>
- Map(function a=>b, list/set/map<a>) => list/set/map<b>
- Reduce(function => a, list<a>) => a
```

Implementation - Builtin Functions: Misc.

```
- Size(container/string) => int
- Put(item, set) => set
- Put(keyItem, valueItem, map) => map
- Contains(item, list/set) => bool
- Contains(key, map) => bool
```

Testing

- Unit Tests for Each Feature of Oscar
- Error Tests for invalid programs
- Larger files for Integration Testing (e.g. *gen_pi.oscar*)

Testing - Integration Scanner Test for *gen_pi.oscar*

```
plt4115@plt4115:~/Oscar/src$ ./oscar -p < ../test/oscar/scanner/gen_pi.oscar
message start()
message end()
message work(start: int, numElems: int)
message result(value: double)
message piApproximation(pi: double)

actor Worker() {


def genRange(start: int, end: int) => list<int> = {
return Reverse(MergeFront(genRange((start + 1), end), list<int>[start]));
}

def calcPi(start: int, numElems: int) => double = {
list<int> range = genRange(start, (start + numElems));
return Reduce((x: double, y: double) => double = {
return (x + y);
}, Map((i: int) => double = {
return (4. * AsDouble(((1 - ((i % 2) * 2)) / ((2 * i) + 1))));
}, range));
}

receive = {
| work(start: int, numElems: int) => {
double pi = calcPi(start, numElems);
message<result>(pi) |> sender;
}
}
}

actor Listener() {



receive = {
```

```
| piApproximation(value: double) => {
Println(("value of pi is approximately :" + AsString(value)));
message<end>() |> sender;
}
}
}

actor Master(numWorkers: int, numMsgs: int, numElems: int) {
mut double pi = 0.;
mut int numResults = 0;
pool<Worker> workerPool = spawn pool<Worker>({}, numWorkers);
actor<Listener> listener = spawn actor<Listener>();



receive = {
| start() => {
message<work> msg = message<work>(0, 10);
msg |>> workerPool;
}
| result(value: double) => {
(pi = (pi + value));
(numResults = (numResults + 1));
if ((numResults == numMsgs)) {
message<piApproximation>(pi) |> listener;
}
}
| end() => {
die();
}
}
}

def main() => unit = {
actor<Master> master = spawn actor<Master>(5, 10000, 10000);
message<start>() |> master;
}
```

Testing

```
$ test git:(master) ./test.sh

----------Testing Valid----------

AND passed

NOT passed

...

----------Testing Errors----------

uopMismatch passed

valAlreadyDeclared passed

...

Tests Passed: 128 $
Tests Failed: 0 $
```

Git History

Sep 25, 2016 – Dec 20, 2016

Contributions to master, excluding merge commits

Contributions: **Commits**

Demo