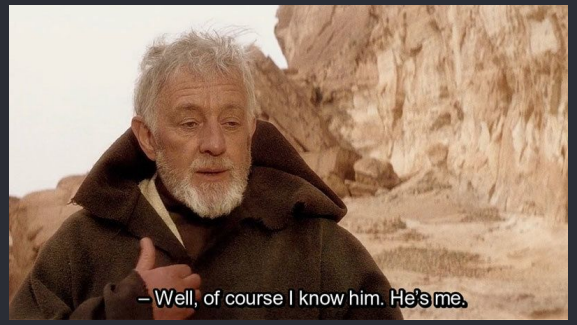
RJEC: Really Just Elementary Concurrency

By: Riya Chakraborty, Justin Chen, Yuanyuting (Elaine) Wang, Caroline Hoang

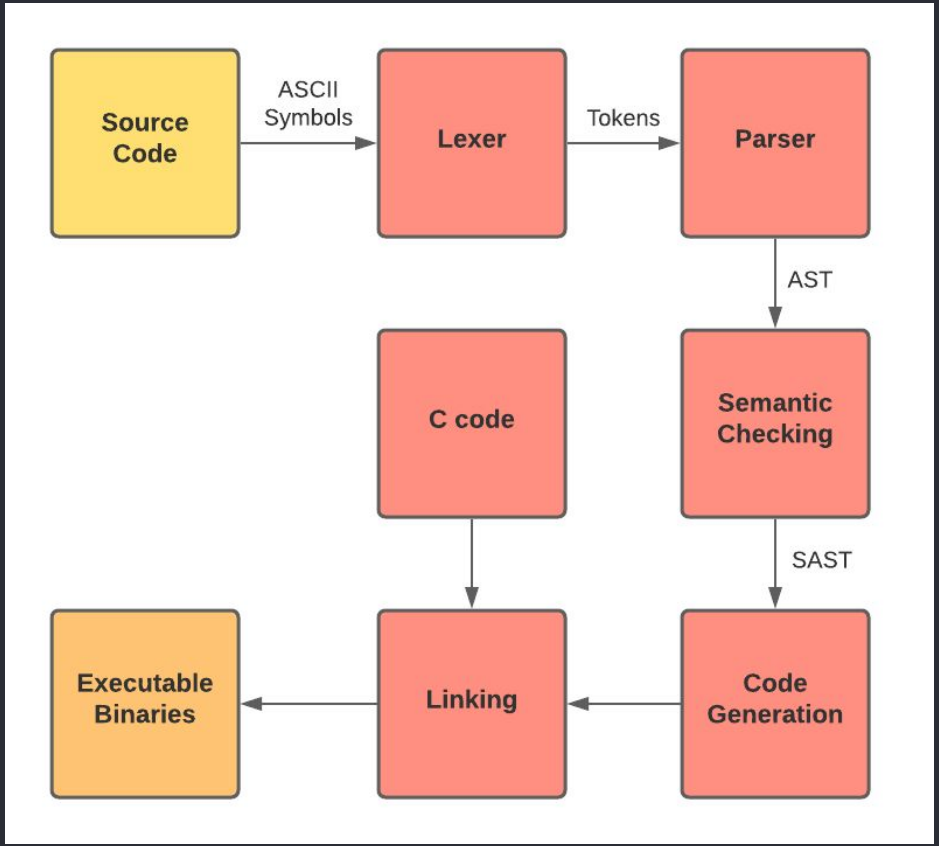
Introduction

Who is RJEC?



- **Motivation:** a simple imperative language with strong concurrency primitives
  - Go-like: nice, consistent, concise, productive syntax
  - CSP-style concurrency abstractions that allow for general purpose concurrent programming
- **Intended audience & use cases**
  - General purpose with a focus on concurrent programming
  - With an eye to distributed applications (though we do not support RPC)

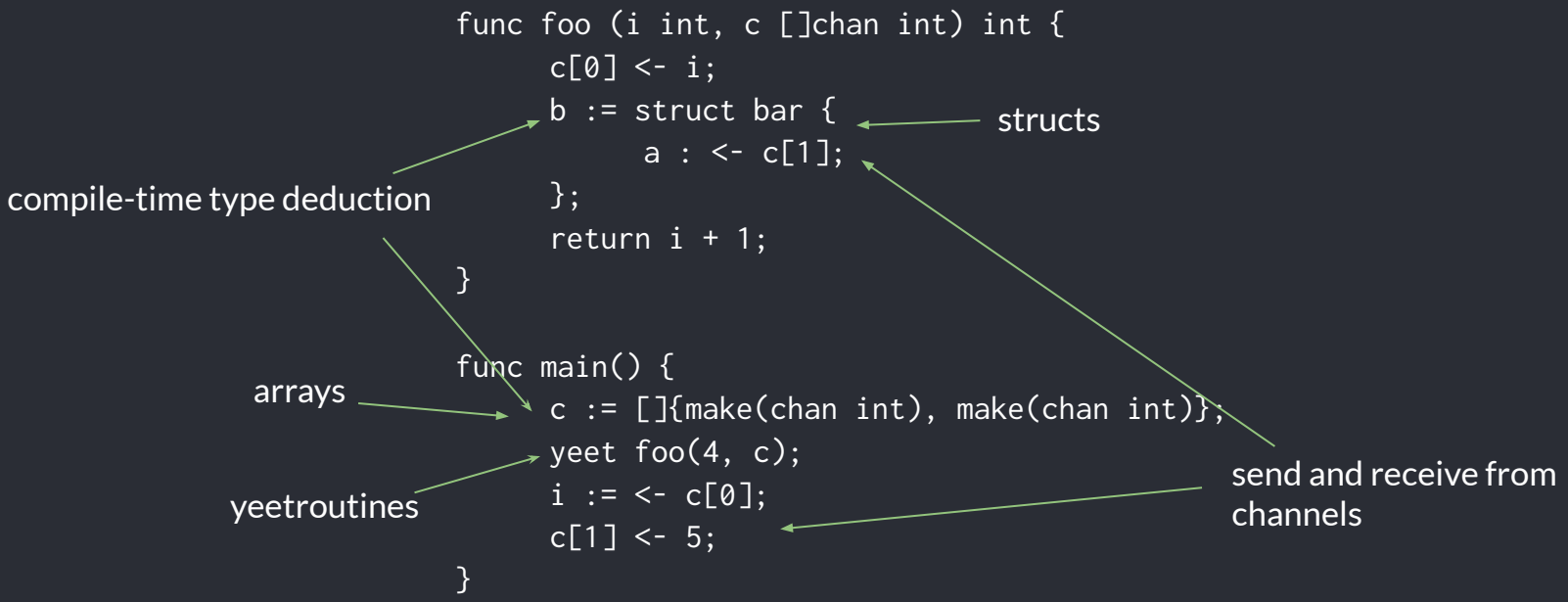
Architectural Design



Language Features

Syntax: RJEC in one slide

- Go-like
- Pass-by-value semantics for basic types and structs



Variable Declaration and Type Deduction

```
struct my_struct {
    a int;
    b char;
    c bool;
}

func foo(x int, y char) struct my_struct {
    i := struct my_struct {
        a : x,
        b : y
    };

    return i;
}

func main() {
    var w, z bool;
    w, z = true, false;
    var x, y int = 42, 30;

    t, k, l := foo(1, 'r'), foo(2, 'j'), 3;
    a, b, c := t.a, t.b, t.c;
}
```

statically and strongly typed

- := init
  - directly initialize to RHS
- = Long-form
  - declare, then initialize
  - **var** keyword on LHS
- One line, multi-var
  - same type!
- No casting
- No implicit type conversion
- ...or else, compiler error

Arrays and Structs

```
struct foo {
    x int;
    y bool;
    z char;
}
func main () {
    n := 10;
    var x, y [n]int;
    for i := 0; i < n; i = i + 1 {
        printi(x[i]);
    }
    z := []int{5, 3, 2};
    str := "rjec";
    prints(str);

    var i struct foo;
    i = struct foo {
        x : 1,
        y : true,
        z : 'a'
    };

    printi(i.x);
    printb(i.y);
    printc(i.z);
}
```

- Arrays
  - Mutable; fixed but var length
  - Array type defined by element type
  - Strings = null-terminated char array
  - No nested arrays
- Structs
  - Globally defined
  - Members stored and assigned by value
  - Passed by value in functions
  - Members of basic type

Yeetroutines

- `yeet`: starts a concurrent thread executing the function call
- Uses coroutines from Libmill library by Martin Sustrik
- Supports functions with any number of formals!
  - RJEC function formals are implemented as a single struct of formals in LLVM to allow for this

```
yeet foo(a1, b1, c1, d1);
yeet foo(a2, b2, c2, d2);
yeet bar();
```

Channels

- Use `make()` to create (un)buffered channels, and `close()` when done using the channel
- Channels block when full until a receiver appears
- Pass data between concurrent processes through channels

“Don't communicate by sharing memory, share memory by communicating.”
-Rob Pike

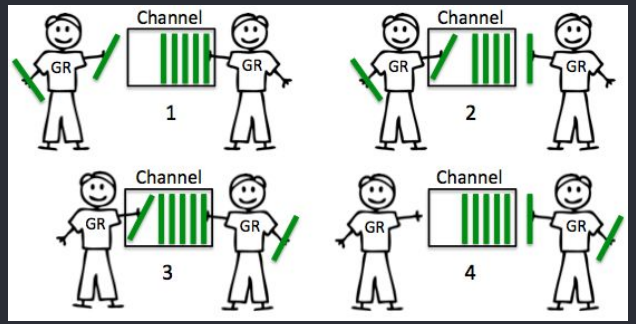


Image courtesy of Soham Kamani

Select: Concurrent Control Flow

- Blocks until able to send or receive from any of the specified channels
- Reverse engineered from `choose` macro in Libmill library to allow for function call
  - Implemented as an array of select clause structs which are passed into C function
- Uses LLVM `switch` instruction

```
func foo(ch1 chan char, ch2 chan int, quit chan bool) {
    for {
        select {
        case ch1 <- 'a':
        case val2 := <- ch2:
            printi(val2);
        case q := <- quit:
            if q { return; }
        }
    }
}
func main() {
    ch1, ch2 := make(chan char, 2), make(chan int);
    quit := make(chan bool);
    yeet foo(ch1, ch2, quit);
    for i := 0; i < 5; i = i + 1 {
        printc(<-ch1);
        ch2 <- i;
    }
    quit <- true;
}
```

So where do we go from here?

- Multiple return values
  - Have support in grammar, can implement similarly to how our formals are implemented
- Lambdas and closures, higher-order functions
  - Could enable built-in map, filter, reduce functions
- **RPC** support
  - To support distributed programming features

Acknowledgements

- Professor Edwards!
- Compilers referenced: MicroC, Shoo, Harmonica, Go
- LRMs referenced: C, Go
- Libraries used: Libmill (by Martin Sustrik), POSIX

Demonstration

Simple Producer-Consumer Problem

- Classic concurrent programming problem
- Use channels to synchronize sending and receiving

Mutex Implementation

- Use a channel with buffer size of 1
- Use `defer` keyword to structure unlocking of mutex at function return

Random Number Generation

- Use a linear congruential generator to generate 1,000,000 random integers
- Times the amount of time taken

MapReduce

- Concurrent algorithm by Google, inspired by functional programming, intended for distributed applications
- Map workers take different parts of the data and process them into categories
- Reduce workers take each category and operate on them
- Demo: primality test

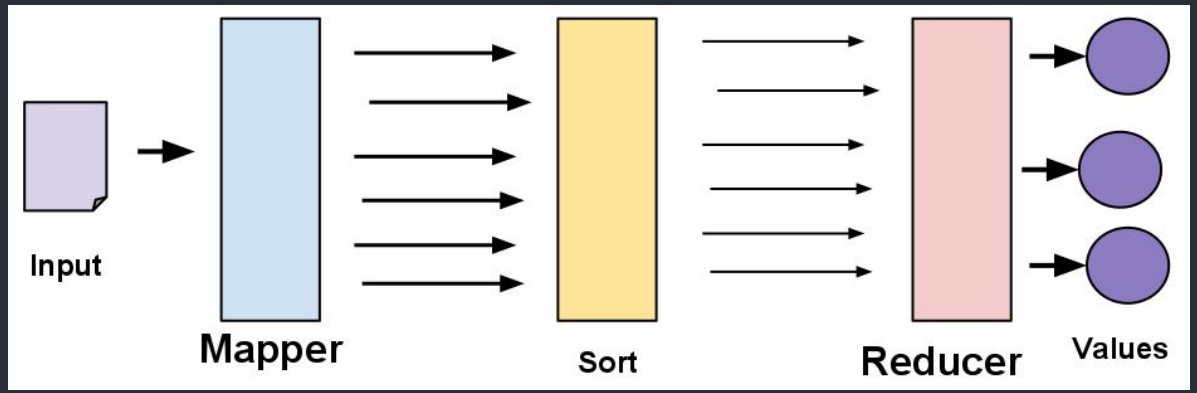


Image courtesy of Berkeley EECS