COMS W4115: RJEC Language Proposal

Really Just Elementary Concurrency

Riya Chakraborty (rc3242), Justin Chen (jbc2186),
Yuanyuting (Elaine) Wang (yw3241), Caroline Hoang (cjh2222)

3 February 2021

1 Introduction

RJEC (Really Just Elementary Concurrency) is an imperative language with a primary focus on concurrent programming. It is based on Go, and its syntax and features are largely a strict subset of Go's. Like Go, RJEC is imperative, but may contain some features that enable some more functional style. We plan to incorporate syntactical features and concurrency abstractions from Go as language primitives. These features will enable somewhat higher-level, CSP-style concurrency, which are useful in distributed systems applications.

2 Language Features

2.1 Basic Syntax

With RJEC, curly brackets ({}) are used to mark the scope of function definitions. RJEC's syntax enforces that the opening brace be on the same line as the function header.

Statements in RJEC can be terminated by either semicolons or newlines. Idiomatically, statements should be terminated by newlines, with semicolons being used for specific use cases only, such as separating the conditions of a for loop. During the lexical analysis stage, when a terminating newline is detected at the end of a statement, a semicolon will be automatically inserted by the lexer.

The syntax uses `//` for single line comments, and `/* */` for multi-line comments.

2.2 Data Types

RJEC is statically typed and strongly typed. The language will support five basic types: int, float, bool, char and mutex. In addition, it will also support four composite types: arrays, struct, chan and func. To this end, strings will be represented as char arrays rather than as its own type.

Data Type	Description	Example
Basic:		
`int`	numeric integer type (4 bytes)	`var i int = 5`
`float`	floating-point numeric type (8 bytes)	`var f float = 5.1`
`bool`	boolean type represented by literals 'true' and 'false' This language does not support 'truthy' values. (i.e. 0 != false)	`var b bool = true`
`char`	character type (1 byte)	`var c char = 'a'`
`mutex`	represents a POSIX mutex	`var m mutex = make(mutex)`
Composite:		

`array`	fixed-length array type that contains mutable elements It is initialized with a length parameter (in the example, length = 10)	`var a1 [10]int`
		`var a2 [10]char`

`struct`	fixed structure struct type that contains mutable elements	`type coord struct{`
		`   x float`
		`   y float`
		`}`

`chan`	represents a channel object	`var m chan int = make(chan int)`

`func`	represents a function object	`func add(x int, y int) int{`
		`   return x + y`
		`}`

2.3 Variable Declaration

Variables are declared and initialized with the following syntax:

```
var i int = 5
```

Each type has a "zero value": `0` for numerics, `false` for bools, and `'\0'` for chars. A variable that is declared without explicit initialization will be initialized to its type's zero value.

We plan to implement compile-time type deduction for declaring variables. This feature is borrowed directly from Go, and is also similar to `auto` from C++11.

Unlike type inference in functional langauges, it is fairly basic and is just setting the type of the newly-declared variable to the type of the value on the right-hand-side:

```
    i := 5 // equivalent to previous declaration
```

Arrays may be declared as follows:

```
    var a [10]int // initializes an array of size 10 of all 0s
```

Or:

```
    b := [5]int{0, 1, 2, 3, 4}
```

Structs may be defined as follows:

```
    type coord struct {
        x float
        y float
    }
```

And declared/initialized as follows:

```
    c := coord{
        x = 5.0,
        y = 5.0,
    }
```

Any struct element which is not explicitly initialized is set to its zero value.

Channels and mutexes are initialized and allocated resources using the `make()` function:

```
    c := make(chan int)
```

Variables only remain in scope, except for global variables, which are declared outside of functions.

2.4 Type Conversion

RJEC enforces a strong typing system, which means the language doesn't conduct implicit type conversions/casting. Operands on the two sides of binary operators have to be of the same type, and failure to abide by the typing system will lead to compiler errors. For example, `1 + 2.0` will lead to a compiler error without casting either operand to the other operand's type first.

Note that `int` and `bool` are treated as distinct types, which means statements such as `1 == true` will lead to errors as well.

Operator	Example
`+, -, /, *, %`	arithmetic operators
`=, :=, +=, -=, /=, *=, %=`	assignment operators
`++, --`	increment and decrement operators
`==, !=, >, <, <=, >=`	same type equality operators
`&&, ||, !`	logical AND, OR and NOT operators
`->, <-`	channel send and receive operators
`[]`	array selection and instantiation operator
`.`	struct access operator

2.5 Operators

RJEC will support the operations displayed in the above chart. The precedence of operators is dictated in the same ways as Go.

2.6 Reserved Words

1. Data types: `int`, `float`, `bool`, `char`, `mutex`, `chan`, `struct`, `array`, `nil`.
2. Control flow: `if`, `else`, `for`, `break`, `continue`, `select`, `case`, `defer`, `return`.
3. Declarations: `var`, `type`, `yeet` (for opening a new concurrent routine/thread), `func`.
4. Miscellaneous: `range`, `main`.

2.7 Control Flow

The main elements of control flow of RJEC are `for`, `range`, `if/else`, `select`, `defer`.

The `for` loop works the way it does in Go.

We would also like to support `range` for loops for arrays and channels, which can be used as following:

```
for i, v := range arr {}
```

This allows for iteration through an array, with the first variable `i` as the index and the second variable `v` as the value.

```
for v := range channel {}
```

This allows for iteration through that which is received via a channel - it also blocks until next value is received in said channel.

`if/else` statements must contain boolean conditions, do not have parentheses and must use cuddled `else` clauses:

```
if 7%2 == 0 {
    ...
} else {
    ...
}
```

`select` blocks and waits on multiple channels and acts on the first one it receives:

```
select {
    case c <- x:
        x, y = y, x+y
    case <- quit:
        return
}
```

`defer` is used to defer the execution of certain statements until right before the current function returns. Just as in Go, `defer` can be used to ensure a mutex will be unlocked immediately before its associated function returns:

```
func sample(data int) {
    lock(mu)
    defer unlock(mu)

    if data... {
        return ...
    }

    if data... {
        return ...
    }

    // do more stuff

    return ...
}
```

`break, continue, return` are all reserved words that can be written alone inside of the contents of a `for` loop (as an example). They have the usual functionality that we associate with them.

2.8 Functions

All functions in RJEC are pass-by-value and may return more than one value. That is, there are no pointers to change a pre-existing object's contents directly via a function.

Function declaration syntax is as follows:

```
func add(x <parameter_type1>, y <parameter_type2>) <return_type>{
    return x+y
}
```

It is also possible to pass functions as parameters for other functions.

```
func compute(fn func(int, int) int) int {
    return fn(3, 4)
}
```

2.9 Built-in Functions

RJEC includes existing built-in functions which support various useful and common operations on arrays and mutexes.

Arrays are fixed-length but mutable, and the associated RJEC functions are:

- `length(arr)` - we note that arrays in RJEC do not dynamically resize

Mutexes are also included as a key aspect of how concurrency works in RJEC:

- `lock(mu)` - a lock mechanism for the mutex, ensures exclusive access to data (this avoids data read/modification conflicts)
- `unlock(mu)` - unlock or free mutex such that data is again openly accessible by a RJEC routine

Printing is supported in two contexts:

- `printf("%d\n", x)` - formatted C-like print
- `println(x)` - print `x`, inserting a new line at the end

We also support:

- `typeof(x)` - to determine the data type of variable `x`

Finally, for allocating or deallocating resources:

- `make(chan int)` - to create new channel or mutex
- `close(ch)` - to close/destroy channels or mutexes

2.10 Standard Library

Some functions specified above, such as the `print` function, may be contained in the standard library instead.

Furthermore, we intend to write some basic concurrent algorithms useful for distributed programming, such as MapReduce and Paxos. If we can make them general enough, we intend to include them in our standard library.

2.11 Concurrency

RJEC has threads, based off Go's goroutines but implemented as POSIX threads. A thread running a given function `foo` is started with the following:

```
yeet foo(2, 4)
```

And terminates when the function terminates.

Mutexes are used for lower-level locking. Mutexes can be initialized as follows:

```
mu := make(mutex)
```

And can be locked/unlocked as follows:

```
lock(mu)
unlock(mu)
```

As is standard, a `lock()` waits until the lock is released if another thread is holding the lock.

RJEC also offers channels for higher-level concurrency abstraction. Channels basically work as queues to send/receive information between threads. Channels hold a particular type, and are initialized as follows:

```
c := make(chan int)
```

The sending thread follows this syntax:

```
c <- 5
```

And the receiving thread follows this syntax, which blocks until receiving a value:

```
i := <- c
```

Channels are by default unbuffered, which means they can hold an unbounded number of values at a time. Bounded channels, which hold a specified maximum number of values, can also be declared as follows:

```
c := make(chan int, 10)
```

Mutexes and channels can be closed as follows:

```
close(mu)
close(c)
```

In terms of implementation details, channels should contain a pointer to some shared memory for communication purposes. However, no pointer logic is exposed to the programmer.

2.12 Extra Features

We have discussed the potential of adding a few other features that we thought may be interesting to implement. Here are some of the options that we came up with:

1. `sync` library - similar to Go's `sync`, we could potentially add greater support for concurrency features.
2. interfaces - Go `interface`
3. higher order functions - pass and return functions into/from other functions

3 Example Code

3.1 Euclid's Algorithm

```
func gcd (a int, b int) int {
    for b != 0 {
        t := b
        b = a % b
        a = t
    }
    return a
}
```

3.2 Single Producer-Consumer

```
func producer (data chan int, quit chan int) {
    i := 0
    for {
        i ++
        select {
            case data <- i:
            case <- quit:
                close(data)
                return
        }
    }
}

func main () {
    data := make(chan int)
    quit := make(chan int)

    // producer
```

```
        yeet producer(data, quit)

        // consumer, terminates when i reaches arbitrary value
        for i := range data {
            print(i)
            if i == 5 {
                quit <- 1
                close(quit)
            }
        }
        return
    }
```