COMS W4115: RJEC Language Reference Manual

Really Just Elementary Concurrency

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

24 February 2021

Contents

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 contains some features that enable some more functional style. Concurrency abstractions are incorporated as language primitives. These features enable somewhat higher-level, CSP-style concurrency, which are useful in distributed systems applications.

2 Lexical Conventions

2.1 Comments

RJEC has support for both single-line and multi-line comments. Any tokens following /∗ are considered part of a comment (whether single or multi-line) and are essentially discarded (they are not included in further lexical analysis or parsing). A comment is over when the ∗/ token is encountered.

2.2 Reserved Keywords

The reserved keywords are as follows, grouped together by their particular uses:

The keywords representing data types are: `int`, `bool` (for which we have `true`, and `false`), `char`, `chan`, `struct`, `array`, `func`. Note that the first three indicate the basic data types, and the following four indicate the composite data types.

The keywords useful for dictating control flow are: `if`, `else`, `for`, `break`, `continue`, `defer`, `select`, `case`, `return`, `yeet`

The keywords necessary for declarations are: `var`

The keywords related to resource acquisition and management are: `make`, `close`. These are associated with the initialization and destruction of channels (discussed in greater detail below).

2.3 Literals

Literals can be used to represent either the basic data types (`int`, `char`, `bool`) or strings.

String literals are a sequence of characters wrapped in double quotation marks, i.e. `"rjec is the best"`. String literals do not need to be bound to a variable.

The lexer is able to identify string literals by going into a different mode of lexing when encountering a double quote (`"`) and reads contents to a buffer

(which constitute the contents of a string literal) until another double quote (") is encountered.

A char literal is a single character wrapped in single quotation marks, i.e. `var mychar char = 'r'`. Character literals do not necessarily have to be bound to a variable either.

An integer literal can be any sequence of integers, each of which are between 0 and 9. The corresponding regex that is matched to by the lexer is $[`0' - `9']+$.

A boolean literal can either be the `true` or `false` keyword, i.e. `var mybool bool = false`.

2.4 Separators

RJEC uses parentheses (`()`) to override any default expression precedence. Similarly, square brackets (`[]`), curly braces () are processed as separators. White spaces are indeed separators, but are not recognized as tokens. The other potential separators include commas (`,`), semicolons (`;`), dots (`.`), and colons (`:`).

The following provide examples of usage of separators:

```
a, b := 1, 2;
/* ’,’ as a separator in multiple variable declaration */

var newArr [2]char = [2]char{’a’, ’b’};
/* [] and {} for array declaration and initialization */

newArr[1] = ’c’;
/* [] in array access and ’;’ indicate the end of a statement */

x_coord = point.x;
/* ’.’ to access field of a struct */

case data <- x:
/* ’:’ in case statement */
```

3 Variables

3.1 Variable Naming

Variable, function, and type *identifier*s must begin with a letter, and must contain only letters, numbers, and underscores. As an example `var_name_1` is a valid identifier, as is `VarName2`. 2strong is not a valid variable, and neither is `_var_name_3`.

3.2 Declaring a variable

Variables may be declared by using their names followed by their type, except in the case of arrays, for which there is a special syntax for noting their type and size:

vdecl:

`var` *id-list vdecl-typ*

id-list:

identifier
identifier`,` *id-list*

vdecl-typ:

`int`
`bool`
`char`
`chan` *basic-typ*
`[`*expr*`]` *typ*
`struct` *identifier*

Variables may be declared both globally and in functions. Variables declared without being explicitly initialized are initialized to their zero value (see "Data Types" section 4). Variables may be declared and initialized in the same line within functions. The operator $identifier$ `:=` *expr* may act as shorthand for variable initialization by assigning the type of the expression to the new variable (see section 7.7).

As examples, the following are both valid variable initialization:

```
var i int = 5
j := 5
```

3.3 Multiple variable declaration

As can be seen in the grammar above, multiple variables of the same type may be declared in one line by being separated by commas. Furthermore, using the `:=` operator (as seen the grammar in 7.7), multiple variables of different types may be declared in one line by being assigned. As examples:

```
var i, j int
k, s := 5, "hi"
```

4 Data Types

4.1 Typing Methodology

RJEC is statically typed and strongly typed. The language supports three basic data types: `int`, `bool`, and `char`. In addition, it also supports four composite types: `array`, `struct`, `chan` and `func`. Note that to this end, string literals will be represented as char arrays rather than as its own type. The types are represented in RJEC's grammar as:

typ:

`int`
`bool`
`char`
`chan` *basic-typ*
`[]` *typ*
`struct` *identifier*
`func (`*type-list*$_{opt}$ `)` *return-types*$_{opt}$

RJEC enforces a strong typing system, which means the language does not conduct implicit type conversions or 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. RJEC does not support type casting.

Note that unlike some other language conventions, RJEC treats `int` and `bool` as distinct types, which means statements such as `1 == true` will lead to compiler errors as well.

4.2 Basic Data Types

The basic data types in RJEC consist of the following:

basic-typ:

`int`
`bool`
`char`

4.2.1 `int`

The `int` type represents signed, 4-byte integer numbers. The integer values are implicitly non-negative, and negative integers are represented by prefixing the unary minus operator `-` to the numerical value. The zero value for `int` is 0.

4.2.2 `bool`

The `bool` type represents 1-byte boolean values. A `bool` type variable can have the value of either `true` or `false`. The zero value for `bool` is `false`.

4.2.3 `char`

The `char` type represents 1-byte ASCII characters. A char literal is represented by enclosing ASCII symbols in single quotes, i.e.

```
'<ASCII_symbol>'
```

Note that in RJEC, string literals are represented as a char array, and are represented by enclosing 0 or more ASCII symbols in double quotes, i.e.

```
"[<ASCII_symbol>]*"
```

The zero value for `char` is `'\0'`.

4.3 Arrays

In RJEC, `array` is a composite data type that allows for the storage of an ordered set of elements in a consecutive memory. Note that the length of an array is fixed upon declaration, and that all the elements in an array must be of the same data type as specified in the declaration. The data type contained in an array can be any other primitive data type supported by RJEC, including composite data types such as `array`, `struct` and `chan`.

4.3.1 Declaring Arrays

One declares an array by specifying the data type of the elements, the length, and the identifier of the array. The declaration takes the following form:

`var` *id-list* `[`*expr*`]`*typ*

For example, to declare an integer array of length 10 and identifier "myArr":

```
var myArr [10]int;
```

Once declared, an array object has the type of `[]<data_type>`.

Note that the *expr* that specifies the length of the array must evaluate to a non-negative integer value, i.e. 0-length arrays are allowed in RJEC.

4.3.2 Defining Arrays

In the grammar, an array literal is defined as:

`[`*expr*`]`*typ* { *args-list* }

Where *expr* should evaluate to a non-negative integer value that represents the length of the array. With this, one can initialize the contents of an array upon declaration through enumeration. That is:

```
var identifier [length]<data-type>=
    [length]<data-type>{elem1, elem2, ...};
```

If the variable to be declared is also to be initialized immediately in the same statement, one can also choose to use a short-form assignment statement instead:

```
identifier := [length]<data-type>{elem1, elem2, ...};
```

For example, the two ways to declare and initialize an integer array of length 3:

```
var myArr [3]int = [3]int{1, 2, 3}
/* or equivalently */
myArr := [3]int{1, 2, 3}
```

Note that if an array is not immediately initialized upon declaration, its fields will be filled by the zero values of the data type specified for its elements.

One can also define an array post-declaration, either by assigning a new value to the entire array, or by accessing and modifying individual array elements with the following format:

```
identifier[index] = <new_value>;
```

4.3.3 Accessing Array Elements

One can access specific elements in an array by their index, with the following expression:

identifier`[`*expr*`]`

Note that in RJEC, the array index starts with 0, so for any array, except for the 0-length arrays, any index value between `0` and `length - 1` (inclusive) is within range.

4.3.4 Nested Arrays

RJEC supports nested arrays, and treats them as arrays containing elements also of the array data type. For example, to declare a 2D integer array with 10 "rows":

```
var myArr [10][]int;
```

4.4 Structs

A `struct` in RJEC is a user-defined data type that consists of fields taken up by elements of the basic data types (`int`, `char`, `bool`).

4.4.1 Defining Structs

In RJEC program files, all structs are defined globally, with all fields public. A struct can be defined with the following grammar:

sdecl:

`struct` *identifier* { *member-list* }

member-list:

identifier basic-typ;
member-list identifier basic-typ;

For example, to define a coordinate type that represents a point in a cartesian coordinate system:

```
struct coordinate {
    x int;
    y int;
}
```

Note that the RJEC compiler grammar enforces that the identifiers for both the struct and the struct fields need to be globally unique. Also, note again that currently RJEC does not support composite data types for struct fields.

4.4.2 Declaring and Initializing Variables of Type Struct

Once a struct type has been globally defined, one can declare variables of the struct type globally or locally, similar to how one declares objects of the primitive types. A struct literal is defined in the grammar as:

expr:

...
`struct` *identifier* { *element-list*$_{opt}$ }
...

element-list:

identifier : *expr*
identifier : *expr*, *element-list*

A struct object can be declared and initialized with the following syntax:

```
var identifier struct struct-id = struct struct-id {
    field-id1 : <value1>,
    field-id2 : <value2>
}
/* or equivalently */
identifier := struct struct-id {
    field-id1 : <value1>,
    field-id2 : <value2>
}
```

For example, to declare and initialize a coordinate struct object:

```
coord := struct coordinate {
    x : 1,
    y : 2
}
```

Note that similar to arrays, if a struct is declared without being immediately initialized, then all of its fields will be taken up by the zero values specified for their respective data types. Later, one can assign values to the individual struct fields with the struct access operator:

```
struct-id.field-id = <value>;
```

4.4.3 Accessing Fields of a Struct

All fields in a struct are public, and can be accessed through the struct access operator `.` with the following expression:

struct-id . *field-id*

4.5 Channels

A `chan` object, or channel, provides a way for concurrently executing subroutines to communicate by sending and receiving values of a specified data type.

4.5.1 Declaring and Initializing Channels

A `chan` object can be declared with the following grammar:

`var` *id-list* `chan` *basic-typ*

The declared object will then have a type specification of `chan <basic-typ>`.

In addition, a `chan` object can be initialized by the built-in function `make`, which allocates resources for either an unbuffered channel, or a buffered channel with a user-defined size. The expression for initializing an unbuffered channel:

`make(chan` *basic-typ*`)`

For initializing a buffered channel:

`make(chan` *basic-typ*`,` *expr*`)`

Here, the expression used to specify the buffer size should evaluate to a non-negative integer value.

When a channel is unbuffered, or when it has a buffer size of 0, the communication succeeds only when both the sender and the receiver are ready for the transaction. When a channel has a positive buffer size, then the elements can "queue up" in the channel, and communication will succeed as long as the buffer is not full (for sending) and not empty (for receiving).

4.5.2 Sending and Receiving Items

Once a channel has been created and is shared between two subroutines, the two functions can act as sender and receiver of elements through the channel, by the use of the arrow operator `<-`.

To send an element through a channel uses the expression:

chan-identifier `<-` *expr*

To receive an element uses the expression:

`<-` *chan-identifier*

The receive operation returns two values, first being the element received (if successful), and second being a `bool` value about whether the channel was open (`true` means the channel was open and the transaction actually took place, and `false` indicates otherwise). The program can also assign the received element directly to a variable by:

```
identifier, ok := <- chan_id
```

Note that the send and receive operations might block under the circumstances where either a buffered channel is full (for sending) or empty (for receiving), or the sender and receiver for an unbuffered channel are not both ready for the transaction.

4.5.3 Passing Channels as Function Parameters

A `chan` object can be shared between two concurrently executing subroutines / functions for them to communicate with each other. To do this, the channel could be locally created in one of the functions, and then passed into the other function as a parameter. An example of this can be seen in the example code in section 10.2.

4.5.4 Closing Channels

The built-in function `close` can be used to close an existing channel and deallocate the associated resources. The associated expression is

`close(`*chan-identifier*`)`

Note that a channel only needs to be closed once for a thorough cleanup. An attempt to close a channel for more than once would result in an error.

After a channel is closed, an attempt to receive from the channel will result in the two values returned being the zero value for the specified data type, and the boolean value `false` indicating the channel was not open. However, attempt to send an item through a closed channel would result in an error. Therefore, although not enforced in the grammar, it's recommended practice to always close the channel from the sender's side.

5 Functions

A RJEC program consists of a sequence of global variable, function, and struct declarations, which can be defined in any order in relation to one another.

5.1 Defining Functions

Function definitions have the form

function-declaration:

`func` *identifier* `(` *parameter-list*$_{opt}$ `)` *return-types*$_{opt}$ `{` *statement-list*$_{opt}$ `}`

parameter-list:

identifier type
identifier type `,` *parameter-list*

return-types:

type
`(` *type-list* `)`

type-list:

type
type-list `,` *type*

statement-list:

statement-list statement

A function definition notes its identifying name, its parameters, its return types, and a list of statements (which are executed in order unless otherwise specified by control flow).

The following is a valid complete function definition:

```
func add(x int, y int) int {
    return x+y;
}
```

5.2 Calling Functions

Functions may be called via the following expression:

identifier `(` *args-list*$_{opt}$ `)`

We list the function parameters as expressions separated by commas:

args-list:

expr
expr `,` *args-list*

Functions in RJEC are pass-by-value only. Calling a function passes the parameters to the function by value, and then executes the statements within that function.

5.3 Passing Functions as Function Parameters

It is possible to pass functions as function parameters. Functions may be passed using their identifier and then referred to in the function body via the name defined in the paramters list. The syntax for defining the function type is defined in the *typ* grammar under the "Data Types" section (4).

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

5.4 The `main` Function

The `main` function is the entry point for code execution. It runs immediately after all global definitions are taken. It has no parameters and no return types.

```
func main() {
    /* execute code */
}
```

5.5 The `return` Statement

The `return` statement is defined as follows:

`return` $\textit{args-list}_{opt}$ `;`

The `return` statement ends execution of the current function and returns to the calling function (or in the case of the `main` function, ends execution of the program). It must be followed by a list of expressions representing the return arguments. These expressions must be of the respective return types specified in the function declaration.

5.6 `yeet`ing Functions

The `yeet` statement is defined as follows:

`yeet` *expr* `;`

Where the expression must be a function call.

`yeet`ing a function runs that function concurrently in a separate thread. That thread terminates when the function returns.
Here is an example of a valid `yeet`:

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

6 Statements & Expressions

As noted in the previous section, a function body consists of a list of statements. A statement is defined as follows:

statement:

expr `;`
vdecl `;`
assign-stmt `;`
`return` *args-list*$_{opt}$ `;`
`{` *statement-list* `}`
`if` *expr* `{` *statement-list* `}` *else-statement*$_{opt}$
`for` *assign-stmt*$_{opt}$ `;` *expr* `;` *assign-stmt*$_{opt}$ `{` *statement-list* `}`
`for` *expr* `{` *statement-list* `}`
`for` `{` *statement-list* `}`
`select` `{` *case-list* `}`
`defer` *expr* `;`
`yeet` *expr* `;`
`break;`
`continue;`

Most statements are expression statements; i.e. consisting of an expression (*expr* `;`). Other statements are described in their relevant sections in this document.

Expressions are defined as follows:

expr:

int-literal
string-literal
char-literal
bool-literal
identifier
expr binop expr
`-` *expr*
`!` *expr*
identifier `(` *args-list*$_{opt}$ `)`
`(` *expr* `)`
identifier `.` *identifier*
identifier `[` *expr* `]`
identifier `<-` *expr*

`<-` *identifier*
`make(chan` *basic-typ* `)`
`make(chan` *basic-typ* `,` *expr* `)`
`close(` *identifier* `)`

Where *binop* is defined as the arithmetic, boolean, and logical binary operators `+`, `-`, `*`, `/`, `%`, `==`, `<`, `<=`, `&&`, and `||`.

These expressions are all explained in their various relevant sections.

7 Operators

7.1 Associativity and Order of Precedence

The associativity and order of precendence for operators in RJEC can be seen in table 7.1, with the various operators listed in order of precedence from top to bottom.

Operators	**Symbols**	**Associativity**
Struct access operator	`.`	left-to-right
Array access and instantiation operator	`[]`	left-to-right
Logical NOT operator and unary negation operator	`!`, `-`	right-to-left
Arrow operator	`<-`	right-to-left
Multiplication, Division and Modulo operators	`*`, `/`, `%`	left-to-right
Addition and Subtraction operators	`+`, `-`	left-to-right
Less than or equal to operators	`<=`, `<`	left-to-right
Equality operator	`==`	left-to-right
Logical AND operator	`&&`	left-to-right
Logical OR operator	`||`	left-to-right
Assignment operators	`=`, `:=`	right-to-left

7.2 Arithmetic Operators

RJEC provides operators that perform basic arithmetic operations, as seen in the following subsections. Note that the binary operators only allow for objects of the same data type as operands, and by default only support integer arithmetic operations.

7.2.1 Addition operator

The addition binary operator `+` returns the sum of the operands. Its associated expression is:

expr `+` *expr*

7.2.2 Subtraction operator

The subtraction operator `-` subtracts the right operand from the left operand and returns the result. Its associated expression is:

expr `-` *expr*

7.2.3 Multiplication Operator

The multiplication operator `*` returns the product of the operands. Its associated expression is:

expr `*` *expr*

7.2.4 Division Operator

The division operator `/` divides the left operand with the right operand and returns the result. Its associated expression is:

expr `/` *expr*

7.2.5 Modulo Operator

The modulo operator is used to obtain the remainder produced by dividing the left operand with the right operand. Its associated expression is:

expr `%` *expr*

7.2.6 Unary Negation Operator

The unary negation operator flips the sign of the original integer value of its operand. Its associated expression is:

`-`*expr*

7.3 Boolean Operators

7.3.1 Equality Operator

The equality operator `==` conducts a deep / value-wise comparison of its two operands, and return `true` if and only if the two operands are structurally identical, or `false` otherwise. This means two struct objects are determined to be equal if they are of the same type, and all of their fields also have the same types and values. Similarly, two arrays are identical if they contain the same number of elements with the same types and values. Its associated expression is:

expr `==` *expr*

7.3.2 Less Than Operator

The less than operator `<` returns `true` if the left operand evaluates to a smaller value than its right operand, or `false` otherwise. By default, this operator only supports types of `int` and `char`. Its associated expression is:

expr `<` *expr*

7.3.3 Less Than or Equal To Operator

The less than or equal to operator `<=` returns `true` if the left operand evaluates to a smaller value than, or is equal to its right operand, or `false` otherwise. By default, this operator only supports types of `int` and `char`. Its associated expression is:

expr `<=` *expr*

7.4 Logical Operators

The logical operators are used to test the truth value of various combinations of one or two expressions. Note that these logical operators by default only support `bool` operands.

7.4.1 AND operator

The logical AND operator, or logical conjunction operator, `&&` returns `true` if and only if both operands evaluate to true, or `false` otherwise. If the left operand already evaluates to `false`, then the expression returns `false` without evaluating the right operand. Its associated expression is:

expr `&&` *expr*

7.4.2 OR operator

The logical OR operator, or logical disjunction operator, `||` returns `true` if and only if at least one of the two operands evaluates to true, or `false` otherwise. If the left operand already evaluates to `true`, then the expression returns `true` without evaluating the right operand. Its associated expression is:

expr `||` *expr*

7.4.3 NOT operator

The logical NOT operator, or logical negation operator, `!` flips the truth value of its operand. Its associated expression is:

`!` *expr*

7.5 Arrow Operator

The arrow operator `<-` is used for sending and receiving elements through objects of type `chan`. The expressions associated with this operator are:

chan-identifier `<-` *expr*
`<-` *chan-identifer*

For more details on its usage, see section 4.5.2.

7.6 Access Operators

The access operators are used to access specific items contained in the composite data types `struct` and `array`. You can use the struct access operator `.` to access the fields of a struct with the following expression:

struct-id `.` *field-id*

You can use the array access operator `[]` to access elements at specific indices of an array with the following expression:

array-identifier `[`*expr*`]`

See sections 4.3.3 and 4.4.3 for more usage of access operators in composite data types.

7.7 Assignment Operators

Assign operators `=` and `:=` are used to store values in variables. The various assignment statements are defined as such in the RJEC grammar:

assign-stmt:

id-list = *args-list*
vdecl = *args-list*
id-list := *args-list*

If the left-hand-side variable(s) have already been declared beforehand, then `=` is used to store the right-hand-side expression value(s) in the variables, designated by the identifier(s) on the left hand side. Both the variables and expressions are separated by commas. For example, to assign 2 integer values to two previously-declared `int` type variables:

```
a, b = 1, 2;
```

The assignment operators can also be used in the case where variable(s) are declared and initialized in the same statement, in which case one could use either the long-form or the short-form assignment statements.

A long-form assignment statement first declares the variable(s) to be initialized, specifying their data type using the `var` keyword on the left hand side, and then uses the `=` operator for the assignment. For example, to declare and initialize one or two `int` variables:

```
var a int = 0;
var b, c int = 1, 2;
```

Note that in the long-form statement, if there are multiple variables to be declared, they are required to be of the same type.

A short-form assignment statement simply refers to variable(s) to be declared by their identifier(s) on the left hand side, and directly initializes each variable with its corresponding expression on the right hand side, using the `:=` operator. For example,

```
a := 0;
b, c := 1, 2;
```

Note that in this case, the variables to be declared together can have values of different data types assigned to each of them. For example,

```
d, e := 3, "4";
```

8 Control Flow

8.1 `for` Loops

`for` loops are used in order to repeat the execution of a sequence of statements and expressions for a specified number of repetitions, for until the termination condition is reached, or infinitely otherwise. The conventional `for` loop definition involves specifying the *initialize* statement (run prior to the loop), *test* boolean expression (termination condition checked after each iteration of the loop), and *step* statement (run after each iteration of the loop).

`for` *assign-stmt*$_{opt}$ `;` *expr* `;` *assign-stmt*$_{opt}$ { *statement-list* }

A `for` loop may also have the behavior of a conventional `while` loop if only the termination condition, or the *test* expression, is specified, in which case the statements will keep executing until the condition evaluates to `false`.

`for` *expr* { *statement-list* }

If no conditions are specified, the `for` loop will continue executing infinitely, unless a `break` or `return` statement is executed at some point.

`for` { *statement-list* }

8.2 `if` and `else` Statements

`if` statements are used to place a condition on the execution of sequence of statements. The statements wrapped in an `if` statement will only execute if the condition expression evaluates to `true`.

The inclusion of `else` statements as well as `else if` statements are optional but can be provided in order to further specify a group of a statements for execution under additional circumstances.

In an `if` statement block, there exists one `if` condition, at most one `else`, and 0 or more `else if` conditions in-between. The conditions are evaluated in order, and statements contained in the first `true` condition will be executed. If an `else` exists, its statements are executed only after all preceding conditions have evaluated to `false`.

statement:

> ...
> `if` *expr* { *statement-list* } *else-statement*
> ...

else-statement:

> *no-else*
> `else if` *expr* { *statement-list* } *else-statement*
> `else` { *statement-list* }

Where *no-else* is defined as an empty token and has lower precedence than the `else` token.

8.3 `break`

`break` statements are used in order to directly specify when to terminate a `for` loop and continue with the next statement after the

Grammar for the `break` statements:

```
break;
```

8.4 `continue`

`continue` statements are used to skip the remaining statements for the current iteration of a `for` loop and begin the next iteration.

Grammar for the `continue` statements:

```
continue;
```

8.5 `select`

The `select` statement is used in order to block and wait on multiple channel-related expressions, each represented as a `case`. It then executes the statements contained in the first case that successfully executes, i.e. the first channel through which it is able to send or receive an element.

The `select` statement can be defined by the following grammar:

statement:

...
`select{` *case-list* `}`
...

case-list:

`case` *case-stmt* `:` *statement-list*
`case` *case-stmt* `:` *statement-list case-list*

case-stmt:

identifier `<-` *expr*
`<-` *identifier*
assign-stmt

Note that for each `case`, the `expr` can take the form of either sending or receiving an item through a channel object. It can also be an assignment statement where an item is received from a channel and then directly assigned / used to initialize a variable. An example of using `select` statements:

```
select {
    case a := <- myChan:
        ...
    case myChan2 <- b:
        ...
}
```

8.6 `defer`

The defer statement is used to defer the evaluation of certain expressions until right before the current function returns.

A `defer` statement is defined as follows:

`defer` *expr;*

9 Standard Library

We plan to include the `print` function (a formatted print function), an array `length` function, and some `map`/`reduce`/`filter` functions in the standard library.

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.

10 Example Code

10.1 Euclid's Algorithm

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

10.2 Single Producer-Consumer

```
func producer (data chan int, quit chan int) {
    i := 0
    for {
        i = i + 1
        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
    }
```