A Concurrent Virtual Machine for a Slice of Go

Playground: https://immanuelhume.github.io/gmm
Repo: https://github.com/immanuelhume/gmm

Phylicia Christel Lim A0241043X
Li Junyi A0233469X

April 2024

Contents

1 Overview

1.1 Project objectives

This project involved creating a concurrent virtual machine for a sublanguage of Go. The objectives for sequential language constructs included, most importantly, variable and function declarations, blocks, conditional statements, and for loops; this also included return, break, and continue statements for control flow. Additionally, concurrency constructs in the form of goroutines, channels, and mutexes were needed. Another requirement was a low-level memory model, with all runtime data structures being allocated from JavaScript's *ArrayBuffer*.

Beyond these basic objectives, we had also planned to implement slices (dynamic arrays), structs, type checking, and memory management via garbage collection. Of these, only structs and type checking were partially implemented in the end, mainly for implementing mutexes and statically binding to methods at compile time. We also introduced pointers as a means of indirection, as it turned out essential for mutexes. More discussion on pointers, types, and type checking can be found below.

Throughout the report, the word *thread* will be used interchangeably with *goroutine*.

1.2 Extent of progress

The virtual machine is stack-based and operates in the same spirit as the one from CS4215's labs. All sequential language constructs specified in the objectives are fully implemented, along with some bells and whistles like multiple assignments on one line. Goroutines, channels, and mutexes were added for the concurrency features. We also added user defined types in the form of structs, as well as some type checking.

1.3 Syntax rules for using *go* statements

We will describe the syntax for go statements, which spawn goroutines. This is a direct, but incomplete, translation of our Antlr grammar to EBNF. Some non-terminals are not described here for brevity. We would also like to enforce that the *primaryExpr* appearing in the go statement is indeed a function via type checking.

⟨*goStmt*⟩ ::= 'go' ⟨*primaryExpr*⟩ ⟨*args*⟩

⟨*primaryExpr*⟩ ::= ⟨*lit*⟩
| ⟨*name*⟩
| '(' ⟨*expr*⟩ ')'
| 'new' '(' ⟨*type*⟩ ')'
| 'make' '(' ⟨*channelType*⟩ ')'
| ⟨*primaryExpr*⟩ ⟨*args*⟩
| ⟨*primaryExpr*⟩ ⟨*selector*⟩

⟨*expr*⟩ ::= ⟨*primaryExpr*⟩
| ⟨*unaryOp*⟩ ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*mulOp*⟩ ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*addOp*⟩ ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*relOp*⟩ ⟨*expr*⟩
| ⟨*expr*⟩ ⟨*logicalOp*⟩ ⟨*expr*⟩

⟨*lit*⟩ ::= ⟨*number*⟩
| ⟨*litStr*⟩
| ⟨*litNil*⟩
| ⟨*litBool*⟩
| ⟨*litFunc*⟩
| ⟨*litStruct*⟩

⟨*litFunc*⟩ ::= 'func' ⟨*signature*⟩ ⟨*funcBody*⟩

⟨*args*⟩ ::= '(' (⟨*arg*⟩ (',' ⟨*arg*⟩)* | ⟨*arg*⟩?) ','? ')'

⟨*arg*⟩ ::= ⟨*expr*⟩

1.4 Platform used

The project comprises of a compiler and an interpreter. Go source code is compiled into a custom bytecode, which is then interpreted by a TypeScript program. There isn't a name for the bytecode we used, so let's just call it GVM code. In the T-diagrams below, we may assume that any JS-x86 interpreter is either NodeJS or a browser.

Figure 1 shows the compilation of Go into GVM. The compiler itself is a JavaScript program. The GVM code is then again interpreted by a JavaScript program, which is our virtual machine code. We wrote the compiler and virtual machine in TypeScript, so these must be compiled into JavaScript; this is shown in fig. 2. Finally, fig. 3 shows the translation of Antlr's grammar into TypeScript source code for the parser.

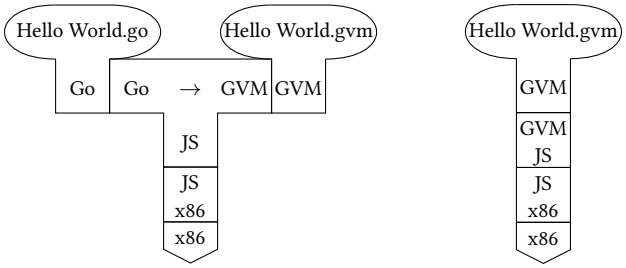

Figure 1: Compiling Go into GVM code, and then running it.

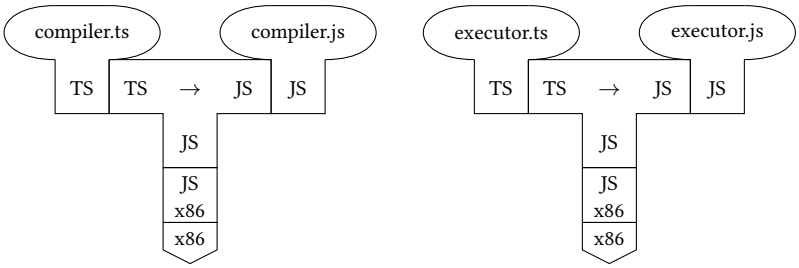

Figure 2: Compiling our compiler and interpreter into JavaScript. The TypeScript compiler is a JavaScript program.

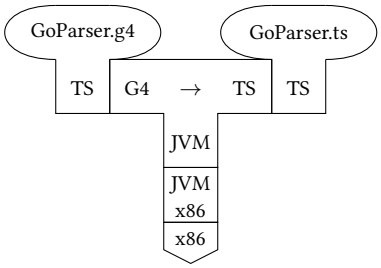

Figure 3: Generating the parser generator. Antlr is a Java program and runs on the JVM.

1.5 Memory

Our memory model is quite unlike Go's, in that everything is "boxed" and allocated on the heap. In that sense it is much more like Python or Ocaml, and is less efficient but easier to implement. All runtime data exists in the form of variable-sized nodes, which are at least one word in size. Each node can contain values (numbers in and of themselves) or references (addresses to other nodes).

1.6 Bytecode

We compile Go source code into bytecode. At the time of writing, there are 30 unique opcodes. Instructions are variable sized, where the first byte is the opcode, and the subsequent bytes carry additional data, if any.

1.7 Passing by value, and why we need pointers

We adhered to Go's convention of passing by value [1]. The upshot is that we must now introduce pointers to the language to support certain programs. For example, mutexes must always be passed as pointers [2]).

```
func foo(mu Mutex) {
        mu.Lock()
        // ...do something
        mu.Unlock()
}
```

In the code above, foo receives a *copy* of whatever mutex is passed in. In essence it represents a different lock from what the caller had intended. The solution is to pass a pointer instead.

Given our existing memory model where there is only a heap representing a graph of nodes, C-style pointers don't necessarily make a lot of sense. Nonetheless it would be required, for instance, to support the behaviour of mutexes described above and pass-by-value semantics. Pointers are thus a new kind of node, holding a word which corresponds to the address of another node. The new pointer node allows us to define the concept of a "shallow copy", where we recursively copy a node and any nodes

[1] See https://go.dev/ref/spec#Calls.
[2] See https://eli.thegreenplace.net/2018/beware-of-copying-mutexes-in-go/.

it references but avoid chasing pointers. Thus, function applications make shallow copies of its arguments.

1.8 Structs

Structs were introduced as a means of implementing the Mutex type, which is a struct in Go. [3] The struct node can be easily defined - a struct with N fields will appear in memory as a header, followed by N words, each word being the address of another node holding the data of that field. We can statically determine struct access at compile time, since accessing the i-th field is equivalent to accessing the node whose address is at the i-th word of the struct node.

1.9 Methods

Methods were introduced to support the `Mutex.Lock()` and `Mutex.Unlock()` idioms. Methods, in Go, can be defined on any type and choose to take either a pointer or value as its "receiver". Taking a pointer receiver is analogous to the notion of "this" in OO languages. Taking a value receiver is as if we made a copy of "this" before calling the method.

Methods, together with interfaces, enable polymorphism in Go. Since we do not support interfaces, there is really no possibility of late binding for methods. Thus, methods are just glorified functions, and can be resolved at compile time.

1.10 Type checking

As described above, we would like to statically resolve struct field accesses and method calls. We do not want to keep any string representation of method or field names at runtime. This requires, minimally, some notion of knowing what the type of some expression is, so that we can determine whether a field or method exists on that expression.

Our language supports type inference within function bodies, but requires explicit types for function parameters and return values. More on type checking rules will be discussed below in section 2.

Type checking is not, currently, implemented to its full potential. For example, function arguments are not type-checked. This allowed us to easily include variadic functions (e.g. `dbg` and `panic`) to help in development. Furthermore, error messages are not the most descriptive.

2 Type checking, in detail

2.1 Environment and store

We used a type environment Γ and type store Q. The type environment maps variable names to their declared or inferred types, and the type store maps type names to their type information. Both data structures follow the same "stack-like" behaviour of regular environments, where the pushing and popping of frames correspond to entering and exiting syntactic scopes.

More formally, $\Gamma(x)$ returns the type of x known by Γ, and $Q(x)$ returns the *type information* of x known by Q (e.g. x is a struct with field foo of type bar).

[3] See https://pkg.go.dev/sync#Mutex.

2.2 Handling recursive types

Recursive types are allowed with indirection. That is, this is okay:

```
type node struct {
        val   int
        left  *node
        right *node
}
```

but this is not:

```
type node struct {
        val   int
        left  node
        right node
}
```

There are practical reasons for disallowing the second case. Most prominently, values can be default initialised. [4] When we write

```
var v node
```

each field of `v` is recursively default initialised, until we hit primitive numeric, boolean, and string types, where numbers are initialised to zero, booleans to false, and strings to the empty string; pointers are initialised to nil (Go's equivalent of a null pointer).

The first node type can be default initialised to look like so:

```
node{val: 0, left: nil, right: nil}
```

But the second type, without pointers, would obviously devolve into an infinite loop while trying to initialise its left and right fields.

2.3 Parsing user-defined types

User-defined types are resolved for each syntactic scope. By "resolve" we mean the recursive process of figuring out, at compile time, complete information about the type. This is a three-step process.

1. Parse type declarations and treat all type references as aliases
2. Convert aliases to qualified type information
3. Resolve cyclic references

The first stage operates purely on a syntactic level. It does not do much more than parsing a type declaration and producing a data structure to represent it. Let's call this data structure a *stub type*. The difference between a stub type and a qualified type is that references to types are treated as aliases, i.e. just a string of some type's name. For instance, the stub type information of the following type declaration

```
type foo struct {
        bar int
}
```

[4] Go calls these zero values, see https://go.dev/tour/basics/12.

is a struct type named *foo* with a single field *bar*, whose type is an alias with the representation "*int*" (we don't care that *int* is the primitive type of integers yet).

The set of stub types is a disjoint union of stub struct, channel, function, pointer, and alias types. The set of qualified types is defined in the similar and obvious way, but there are no aliases in qualified types.

So the first stage outputs a frame of stub types $\{S_i\}$, which is fed into the second stage along with an existing type store Q, and resolves the each S_i in a depth-first manner; essentially, we want to resolve all aliases. This can be described with the $\rightsquigarrow_Q$ relation, where $S \rightsquigarrow_Q T$ means that the stub type S resolves to the qualified type T with respect to the type store Q. For a given S_i, we check the following six rules in sequence.

1. If S is a struct with fields f'_i, and $f'_i \rightsquigarrow_Q f_i$, then $S \rightsquigarrow_Q T$ where T is a struct with fields f_i.
2. If S is a channel with element type e', and $e' \rightsquigarrow_Q e$, then $S \rightsquigarrow_Q T$ where T is a channel with element e.
3. If S is a function with parameters p'_i and results r'_i, and $p'_i \rightsquigarrow_Q p_i$ and $r'_i \rightsquigarrow_Q r_i$, then $S \rightsquigarrow_Q T$ where T is a function with parameters of type p_i and results of type r_i.
4. If S is a pointer containing type t', and $t' \rightsquigarrow_Q t$, then $S \rightsquigarrow_Q T$ where T is a pointer containing something of type t.
5. If S is an alias with name x, and there exists $\xi \in \{S_i\}$ where ξ is named x and $\xi \rightsquigarrow_Q T$, then $S \rightsquigarrow_Q T$.
6. If S is an alias with name x, and $Q(x) = T$, then $S \rightsquigarrow_Q T$.

If none of the six rules succeed for some stub S, then S is ill-defined and we can emit an appropriate compile-time error. A simple example of such an S would be

```
type foo bar
```

where there is no occurrence of *bar* defined anywhere in the program. Here *foo* is parsed as an alias type, so the relevant rules are 5 and 6 which it both fails.

Unfortunately, these rules are somewhat idealized and do not cleanly apply to reality, as types can be recursively defined. Encountering a loop in the second step indicates the presence of recursive types. If the recursive reference is not indirected (i.e. it is not a pointer) then we can immediately throw an error. Otherwise, we evaluate it to a dummy pointer type, and resolve the pointer's element type in the third stage, which, to borrow some parlance from Haskell, "ties the recursive knot".

2.4 Resolving struct fields and methods

This deals with the '.' notation. Let's use the following code as an example.

```
type point struct {
        x float
        y float
}
func (p point) dot(p2 point) {
        return p.x*p2.x + p.y*p2.y
}
```

In the *dot* method, we need to resolve *p.x*. Concretely, it follows these steps:

1. Look up current type environment for "p"
2. Discover that the type of "p" is a struct named "point"
3. Discover the position of the field "x" to be 0
4. Emit instructions to access 0^{th} child of p

We can express this in slightly more formal rules. Let Γ be the current type environment, and define the ternary relation

$$\Gamma \vdash E : t$$

which means that the expression E evaluates to type t where the free variables of E have types provided from Γ. Now resolving an expression $b.f$ corresponds to checking these two rules in sequence.

1. If $\Gamma \vdash b : s$ where s is a struct with a field named f of type T, then $\Gamma \vdash b.f : T$.
2. If $\Gamma \vdash b : s$ where s has a method named f of type F, then $\Gamma \vdash b.f : F$.

This also highlights that *all* types can have methods defined on them.

2.5 Multiple assignments and return values

A neat little consequence of having type checking is that it is now easy to support Go's multiple assignment syntax. Consider this example.

```
func succ(x int) (int, int) {
        return x, x + 1
}
func main() {
        one, two := succ(1)
}
```

In the main function, we know, with type checking, that the expression *succ(1)* actually evaluates to two items. That is, the type of *succ(1)* is *(int, int)*. This allows us to safely compile the assignment.

More subtly, it actually allows us to generate the correct instructions for cleaning up the operand stack after each statement. Consider this example.

```
func main() {
        succ(1)
        // ...moving on with life
}
```

The call to *succ(1)* leaves two items on the operand stack. Both items should be cleared before we move on to the next statement. Without type checking, we couldn't be sure of how many values to pop! However, type checking tells us, clearly, that this expression would leave two values on the stack. It is now easy to generate two *Pop* instructions to remove both values.

3 Concurrency

3.1 Goroutines

We have a *Go* opcode specifically for starting goroutines. A *Go* instruction is always followed immediately with a *Call* instruction. The current goroutine forks itself, transfers part of its operand stack to the new goroutine, and then skips the following Call instruction. The new goroutine is scheduled to start running at the Call instruction.

Each goroutine is represented at runtime as a record of its own registers — program counter, runtime stack, operand stack, and environment — as well as data for its thread ID, whether it is live (or blocked), whether it is a zombie, and also the address of its final instruction. Goroutines are initialised with an empty runtime stack, and terminate once its runtime stack becomes empty.

3.2 Concurrent executor

Threads are stored in two queues, the *live* queue and *dead* queue. The live queue contains threads which we know can perform some meaningful work. That is, they are not blocked on anything. The dead queue stores threads which cannot progress and is waiting on some other threads. Conceptually, the executor polls from the live queue, tries to do some work with the thread, then returns it to either the dead or live queue. If there is no work left in the live queue, it looks through the dead queue to see if any threads are no longer blocked.

In practice, there is a bit of extra machinery needed to *signal* when threads should block and unblock. A thread will be blocked in one of three cases.

1. It tried to write a channel, but no one is ready to read.
2. It tried to read from a channel, but no one is ready to write.
3. It tried to lock a mutex locked by another thread.

A thread marks itself as blocked upon encountering these scenarios, and the executor stops running it and shifts it to the dead queue. The thread also subscribes to an event representing what would unblock it. For example, it may subscribe to the *mutex-unlock* event and mark itself as alive again when the event is published.

Once in a while, we will shuffle contents of the live and dead queue, i.e. transfer blocked threads in the live queue to the dead queue, and vice versa. The upshot is that, within the existing concurrency capabilities, it is easy to define when deadlock occurs. Deadlock occurs if and only if after shuffling, the live queue is empty. [5]

Events are published by threads upon taking certain actions. In the example above, when a thread unlocks a mutex, it publishes a *mutex-unlock* event, so that threads waiting on that same mutex, which should have subscribed to the event, can be unblocked. This means that each mutex needs to be assigned some unique identifier, which is then used as part of the event. We set this unique identifier upon the first use of the mutex. Channels are implemented in a similar fashion.

The subscribing to and publishing of events is implemented within the virtual machine itself using rather pedestrian JavaScript data structures — it is just a JavaScript object where the keys are the serialised events and the values are thread IDs.

[5] This would not be true if we had a *sleep* function.

We write here some pseudocode for demonstration. The procedures for locking a mutex, reading from a channel, and writing to a channel are shown in algorithm 1, algorithm 2, and algorithm 3 respectively.[6]

Channels can be in one of three states:

1. Full — some thread wants to write
2. Normal — no one is writing or reading
3. Hungry — some thread wants to read.

Algorithm 1 Locking a mutex

```
Require: mutex m and thread t
  if m.isLocked then
      t.isLive ← false
      subscribe to event (mutex-unlock, m.id)
  else
      m.isLocked ← true
      if m.id = 0 then                          ▷ first use of m
          m.id ← unique, positive integer
      end if
  end if
```

Algorithm 2 Reading from a channel

```
Require: channel c and thread t
  if c.id = 0 then                              ▷ first use of c
      c.id ← unique, positive integer
  end if
  if c.state = Full then
      c.state ← Normal
      publish event (chan-read, c.id)
  else
      if c.state = Normal then
          c.state ← Hungry
      end if
      t.isLive ← false
      subscribe to event (chan-send, c.id)
  end if
```

3.3 Representation of mutexes and channels

The runtime representation of mutexes and channels deserve further discussion. While they are built-in types, they do not have special treatment in terms of data representation. A mutex is a struct defined like so.

[6]The actual code for channels can be found at https://github.com/immanuelhume/gmm/blob/main/src/eval.ts#L448. And the actual code for mutexes can be found at https://github.com/immanuelhume/gmm/blob/main/src/eval.ts#L1085.

Algorithm 3 Writing to a channel

Require: channel c and thread t
  if c.id = 0 **then** ▷ first use of c
    c.id ← unique, positive integer
  end if
  if c.state = Full **then**
    t.isLive ← false
    subscribe to event (*chan-read*, c.id)
  else if c.state = Normal **then**
    copy data into c
    c.status ← Full
    t.isLive ← false
    subscribe to event (*chan-read*, c.id)
  else
    copy data into c
    c.status ← Full
    publish event (*chan-write*, c.id)
  end if

```
type Mutex struct {
        id       int
        isLocked bool
}
```

Where "locked" should be accessed atomically (this is guaranteed in JavaScript environments). Similarly, a channel is just a pointer to an underlying struct, which, for illustration purposes, might look like this:

```
type _channel struct {
        id     int
        status int
        data   T
}
type channel *_channel
```

where T is the type in the channel. Since channels are "generic" we can't really express them accurately here. Anyway, the point is that mutexes are just structs, and channels are just pointers to structs.

3.4 Example with channels

We now walk through a full example of using channels and discuss it from the perspective of the executor.

```
func main() {
        ch := make(chan int)
        go func() {
                ch <- 1
        }()
        n := <-ch
}
```

The literal execution sequence of instructions is shown below.

```
1  [ 0 ] 0x000 EnterBlock
2  [ 0 ] 0x002 LoadFn argc:0 pc:0x01d last:0x087
3  [ 0 ] 0x014 Goto 0x088
4  [ 0 ] 0x088 LoadNameLoc frame:0 offset:0
5  [ 0 ] 0x08b Assign 1
6  [ 0 ] 0x08d LoadName frame:0 offset:0
7  [ 0 ] 0x090 Call argc:0 go?:false
8  [ 0 ] 0x01d EnterBlock
9  [ 0 ] 0x01f LoadC kind:Int64 val:0
10 [ 0 ] 0x029 LoadC kind:Int64 val:0
11 [ 0 ] 0x033 LoadC kind:Int64 val:0
12 [ 0 ] 0x03d PackStruct 3
13 [ 0 ] 0x046 PackPtr
14 [ 0 ] 0x047 LoadNameLoc frame:0 offset:0
15 [ 0 ] 0x04a Assign 1
16 [ 0 ] 0x04c LoadFn argc:0 pc:0x067 last:0x078
17 [ 0 ] 0x05e Goto 0x079
18 [ 0 ] 0x079 Go
19 [ 0 ] 0x07d LoadName frame:0 offset:0
20 [ 0 ] 0x080 ChanRead
21 [ 1 ] 0x07a Call argc:0 go?:true
22 [ 1 ] 0x067 EnterBlock
23 [ 1 ] 0x069 LoadC kind:Int64 val:1
24 [ 1 ] 0x073 LoadName frame:2 offset:0
25 [ 1 ] 0x076 ChanWrite
26 [ 1 ] 0x077 ExitBlock
27 [ 0 ] 0x080 ChanRead
28 [ 0 ] 0x081 LoadNameLoc frame:0 offset:1
29 [ 0 ] 0x084 Assign 1
30 [ 0 ] 0x086 ExitBlock
31 [ 0 ] 0x087 Return
```

The first column indicates the currently executing goroutine in square braces. Thread 0 is the main thread, and thread 1 is the goroutine we spawned. Initially, we only have thread 0 in the live queue. At line 18, the *Go* instruction forks the current thread and prepares a new goroutine to start running. The new goroutine is assigned thread ID 1 and placed at the back of the live queue. It does not start executing immediately.

At line 20, the main thread attempts a *ChanRead*. Since the goroutine has not started, there is no one ready to write to the channel. Thus thread 0 sets itself as blocked, according to algorithm 2. The executor shifts thread 0 to the dead queue, and thread 0's program counter is *not incremented*.

The executor now polls from the live queue to find thread 1. It runs the thread. At line 25 we see that *ChanWrite* is executed, and *chan-write* is published according to algorithm 3. This unblocks thread 0, but it is not shifted to the live queue at this point; it just sets a flag on itself to indicate that it is no longer blocked. Thread 1 moves on with life, completes its final instruction, and is removed from the thread pool.

At this time, there would be nothing in the live queue, as thread 0 is still in the dead queue. The executor polls the live queue to find nothing, upon which it checks through the dead queue to see if any threads have become unblocked. Thread 0 is

indeed unblocked, so it is shifted to the live queue, and we poll it to execute. And Bob's your uncle.

As a final note, the instructions executed on lines 9 — 13 correspond to the creation of a channel. It is, as indicated above, just a pointer to a struct of three fields.

3.5 Zombie threads

The event pub-sub mechanism described can also be used to clean up zombie threads. In Go, when a parent goroutine exits, its children are also terminated. It is easy to implement this. When each new goroutine is forked, we subscribe to the *fin* event for its parent goroutine's ID. The *fin* event is published when any goroutine exits, upon which its children would mark themselves as zombies. The executor does not execute any zombie threads; they are removed from the thread pool.

4 Limitations and future work

As mentioned above, type checking could be more complete, and error messages could be improved. Though the type checker seems to work fine, the use of the type checker is currently inconsistent as we are not performing a type-checking pass followed by a code generation pass. So the type-checker is used during compilation in a rather ad-hoc way. This is, in part, due to the clunkiness of working with the parse tree generated by Antlr. Converting the parse tree into an AST structure defined by ourselves would make it easier to perform multiple passes and modify AST nodes as needed.

A logical next step for the project is to include garbage collection, or to explore a leaner memory model more akin to what Go actually does. Given that we have a functioning type-checker, we could potentially eliminate some runtime type information and streamline the VM's microcode for instructions.

4.1 Gotchas

Certain syntax, valid in actual Go, is invalid in our language. These are limitations of the parser or compiler.

1. Integer and float literals are strict. For instance, the literal "42" will always be interpreted as an int, and "42.0" is always a float. These will lead to compile errors:

   ```
   var x float = 42
   var y int   = 42.0
   ```

2. Certain one-liners are invalid. This is a purely cosmetic issue and does not affect runtime. These, although valid in actual Go, will not parse here:

   ```
   func main() { dbg("hello world") }
   type node struct { val int; left *node; right *node }
   ```

   Instead, please place things things between braces in their own lines, like so:

   ```
   func main() {
       dbg("hello world")
   }
   ```

```
type node struct {
    val   int
    left  *node
    right *node
}
```

3. Struct literals must specify all fields.

```
u := node{val: 1} // compile error!
v := node{val: 1, left: nil, right: nil} // all good
```

5 Building and testing

5.1 Building the project

There is only one git repository at https://github.com/immanuelhume/gmm. The project was built using NodeJS 21.7. Antlr 4.12 was used as the parser generator, but its output is already checked into the repository so Antlr4 binaries are not required for building the repo. Follow these steps to clone and install dependencies.

```
git clone git@github.com:immanuelhume/gmm
cd gmm
npm i
```

The live editor can be run locally as well. To build the editor, use

```
npx vite build
```

The static files for the editor are placed in *dist/*, and they can be served up using your choice of a file server. For instance, we can use python like so

```
python -m http.server 5183 --bind 127.0.0.1 --directory ./dist
```

and the editor will be available at http://localhost:5183.

5.2 Test cases

Test cases are placed in the *go/* folder, and divided into two kinds. In *go/pass/* we have programs which should pass, and in *go/fail/* we have programs which should fail. Tests are driven by a custom script - to run all the test cases, run

```
npm run runall
```

Individual files can be ran with

```
npm run run path/to/my/file.go
```

Most tests follow this structure, and use the panic function to indicate failure.

```
want := /* expected result */
got  := /* what we got */
if got != want {
    panic("expected", want, "got", got)
}
```

We will now describe a few test cases, truncating some of them for brevity.

```
func main() {
        f := getF()
        got := exec(f)
        if got != 2 {
                panic("expected 2, got", got)
        }
}

func exec(f func() int) int {
        return f()
}

func getF() func() {
        a := 1
        f := func() int {
                return a + 1
        }
        return f
}
```

Listing 1: Higher order functions, closures.

```
func main() {
        a, b := 5, 2
        if a-b != 3 {
                panic("expected 3, got", a-b)
        }

        b, a := a, b
        if a != 2 || b != 5 {
                panic("expected (2, 5), got", a, b)
        }
}
```

Listing 2: Multiple assignment.

```
type counter struct {
        n int
}

func (c *counter) incr() {
        c.n = c.n + 1
}

func main() {
        var c counter
        if c.n != 0 {
                panic("expected 0, got", c)
        }
        c.incr()
        if c.n != 1 {
                panic("expected 1, got", c)
        }
}
```

Listing 3: User-defined struct, with pointer receiver.

```
type counter struct {
        n int
}

func (c counter) incr() {
        c.n = c.n + 1
}

func main() {
        var c counter
        if c.n != 0 {
                panic("expected 0, got", c)
        }
        c.incr()
        if c.n != 0 {
                panic("expected 0, got", c)
        }
}
```

Listing 4: User-defined struct, with value receiver. Analogous to listing 3, but we use a value receiver, so the counter's value should not be incremented.

```
type point struct {
        x int
        y int
}

func main() {
        p := new(point)
        p.x = 3
        p.y = 7
        sum := p.x + p.y
        if sum != 10 {
                panic("expected 10, got", sum)
        }
}
```

Listing 5: Pointer initialisation with new.

```
func main() {
        var u node
        var v node
        var w node

        u.val, v.val, w.val = 1, 2, 3
        u.left, u.right = &v, &w

        sum := u.sum()
        if sum != 6 {
                panic("expected 6, got", sum)
        }
}

type node struct {
        val   int
        left  *node
        right *node
}

func (n *node) sum() int {
        if n == nil {
                return 0
        }
        return n.val + n.left.sum() + n.right.sum()
}
```

Listing 6: Pointers for recursive types.

```
func main() {
	x := 0
	for i := 1; i != 4; i = i + 1 {
		for j := 1; j != 4; j = j + 1 {
			if i == j {
				break
			}
			x = x + 1
		}
	}
	if x != 3 {
		panic("expected 3, got", x)
	}
}
```

Listing 7: *Break* statements.

```
func main() {
	y := 0
	for i := 0; i != 5; i = i + 1 {
		if y == 3 {
			continue
		}
		y = y + 1
	}
	if y != 3 {
		panic("expected 3, got", y)
	}
}
```

Listing 8: *Continue* statements.

```
func main() {
	go work("goroutine")
	work("main")
}

func work(s string) {
	for i := 0; i < 3; i = i + 1 {
		dbg(i, "hello", s)
	}
	dbg("bye", s)
}
```

Listing 9: Goroutines. No assertions for this case, it's more of "a run it and look at the output" thing.

```
func main() {
        ch := make(chan int)

        go func(ch chan int) {
                ch <- 3
        }(ch)
        x := <-ch
        if x != 3 {
                panic("expected 3, got", x)
        }

        // the channel can be reused
        go func() {
                ch <- 7
        }()
        x = <-ch
        if x != 7 {
                panic("expected 7, got", x)
        }
}
```

Listing 10: Channels.

```
func main() {
        go 1
}
```

Listing 11: Using go statement without function application. Fails at compile time with "CompileError: 2:1: expression in go must be function call".

```
func main() {
}

type foo bar
```

Listing 12: Using undeclared types. Fails at compile time with "CompileError: 4:0: undefined: bar".

```
func main() {
        var x int = "foo"
}
```

Listing 13: Assigning to the wrong type. Fails at at compile time with "CompileError: 2:1: expected int, got string".

```
func main() {
        var mu Mutex
        mu.Lock()
        mu.Lock()
}
```

Listing 14: Deadock. Fails at runtime with Error: “deadlock!”.

```
func main() {
        var x *int
        dbg(*x)
}
```

Listing 15: Dereferencing nil. Fails at runtime with: “PanicError: tried to dereference nil pointer”.

A The test case during our presentation

During the presentation, we tried to run this program demonstrating a concurrent summing over a binary tree.

```
func main() {
        var u node
        var v node
        var w node

        u.val, v.val, w.val = 1, 2, 3
        u.left, u.right = &v, &w

        res := make(chan int)
        go u.sum(res)

        x := <-res

        dbg("sum of tree is", x)
}

type node struct {
        val   int
        left  *node
        right *node
}

func (n *node) sum(ch chan int) int {
        if n == nil {
                ch <- 0
                return 0
        }
        cl := make(chan int)
        cr := make(chan int)
```

```
        go n.left.sum(cl)
        go n.right.sum(cr)
        l := <-cl
        r := <-cr
        sum := n.val + l + r
        ch <- sum
        return sum
}
```

It failed then with cryptic errors. It turns out the error was due to

1. Improper termination of goroutines. This was a logical bug where goroutines would try to run even after their function has returned.
2. A missing break statement in a switch-case block handling channel writes (yikes).

It is now fixed, and added to our playground under the “concurrent bintree” example.