Design Document

Milestone 3

COMP 520: Compiler design

Maxence Frenette (260 685 124)
Maxime Plante (260 685 695)
Mathieu Lapointe (260 685 906)

Mar. 27, 2018

Semantics & Code Generation Status

Language choice

We will generate Javascript code targeting the latest version of Node.js (that is, version 9 at the time of writing). This version of Node implements the ECMAScript 2017 standard, which is the 8th edition of the Javascript standard. Targeting Javascript is a realistic real-world example, since the trend in the last decade or so has been to compile languages to Javascript in order to target browsers without having to write any javascript. We are targeting Node instead of the browser for the sake of easier testing, since node can easily run scripts from the command line.

Javascript was an obvious choice for us for four main reasons. First, it is a high-level scripting language, which means that we won't have to worry about memory management and we can use many high-level constructs. Second, the semantics are really similar to those of Go. Third, the tooling is great. We don't have to write code to format the output Javascript since we can offload that work to Prettier[1]. And last, but not least, one of our team members, Maxence, knows the language really well, including the different versions of the standard and the language quirks.

Javascript still has some downsides as a (target) language. First, it's slow. It's not as bad as Python[2], but it still falls short of compiled, statically typed languages such as Java or C++. Second, the dynamically-typed nature of the language means that during development, some bugs in the codegen will be found at runtime instead of at compile-time.

Semantics of Go

Identifiers

Semantics of Go

In Go, identifiers serve as names for variables, struct members, functions and types. They must be unique within some enclosing set that depends on the type of identifier. They cannot conflict with keywords, but that set is pretty restrictive. That is because some constructs, such as true and false, are implemented as identifiers instead of reserved keywords. Finally, there is a blank identifier which can only be used as a placeholder on the left side of an assignment, in declarations, as a function argument and as a struct member.

[1] https://prettier.io/
[2] https://benchmarksgame.alioth.debian.org/u64q/compare.php?lang=node&lang2=python3

Semantics of Javascript and Mapping Strategy

Javascript accepts the same kind of identifiers as Go. Since the set of keywords is different in both languages, all identifiers will be prefixed with an underscore. As it turns out, this will be the only mangling needed (Javascript block scopes work roughly the same way as in Go). Predeclared identifiers such as `true` and `false` will be declared in the global scope in the Javascript code. Finally, to handle blank identifiers, a global blank identifier will be declared in the global javascript code. The parser, weeder and typechecker will ensure that nothing illegal happens to this blank identifier.

Variable declarations

Go allows declaring variables in multiple levels of grouping. First, multiple declarations specs can be grouped with the following syntax:

```
var (
    // ...
)
```

Each element inside such a declaration are called `VarSpec`s by the Go spec. Each VarSpec can then declare and initialize multiple variables separated by commas like so:

```
var a, b int = 1, 2
```

It is also possible of specifying the type like in the above example. In general, a variable needs to either have a type, be initialized when it's declared or both. In the case where no type is specified, the type is inferred from the initialization. If the variables aren't explicitly initialized, they are initialized to some default value depending on the type.

Semantics of Javascript and Mapping Strategy

Javascript doesn't have a syntax equivalent to Go's for grouping multiple `VarSpec`s into one variable declaration. Instead, the `VarSpec`s are simply written one after the other, separated by newlines. Javascript neither support declaring and initializing multiple variables in one statement. Finally, Javascript variables default to undefined when they aren't explicitly initialised. Thus, the generated code must explicitly initialize those variables according to their types. Here is a complete example that includes all the cases mentioned above.

Go	Javascript

`var (`
`  a, b int`
`  c, d = "Hi", 2.0`
`  e string`
`)`
	`let [a, b] = [0, 0];`
	`let [c, d] = ["Hi", 2.0];`
	`let e = "";`

If statements

Semantics of Go

In Go, If statements have an optional init statement, a condition and a block that will be executed is the condition is true. The init statement must be a simple statement. It is allowed to redefine previously defined variable, which effectively makes it a new scope. The condition is simple an expression that must be of type boolean. Finally, the block is a block of statements that will be executed is the condition is true. Like any other block in Go, it opens a new scope. The else if and else constructs also exist in Go. Else if constructs are simply

Semantics of Javascript and Mapping Strategy

Javascript has if statements that are semantically equivalent to Go's, except for the *init* part. Indeed, since the introduction of the `let` keyword in recent years, Javascript now has proper scoping rules, as long as the `let` and `const` keywords are used to declare new variables instead of the `var` keyword. This means that if statements without an *init* statement can be trivially compiled like so:

Go	Javascript

`var a = 1`
 	
`if 3 < 4 {`
`  a := 2`
`  print(a)`
`}`
	`let a = 1;`
	
	`if (3 < 4) {`
	`  let a = 2;`
	
	`process.stdout.write(a.toString());`
	`}`

Meanwhile, to mimic the scope of init, we create another scope that encloses the init statement and the javascript if statement.

Go	Javascript

`var a = 1`
 	
`if a := 2; 3 < 4 {`
`  print(a)`
`}`
	`let a = 1;`
	
	`{`
	`  let a = 2;`
	`  If (3 < 4) {`
	
	`process.stdout.write(a.toString());`
	`  }`
	`}`

For compound if statements involving init statement, we simply go recursively. Handling the else is trivial since the else block also has access to any potential declaration made as the *init* statement. This block encasing is not necessary when the later else-ifs don't have init statements.

Go	Javascript

`var a, b = 1, 1;`
 	
`if a:= 2; 3 < 4 {`
`  print(a)`
`} else if b := 2; 3 < 4`
`{`
`  print(b)`
`} else {`
`  print("Hello")`
`}`
	`let [a, b] = [1, 1];`
	
	`{`
	`  let a = 2;`
	`  if (3 < 4 {`
	`  process.stdout.write(a)`
	`  } else {`
	`  { // This block isn't necessary, but the codegen might not be able to optimise it out.`
	`  {`
	`  let b = 2;`
	`  if (3 < 4) {`
	`  Process`
	`  .stdout`
	`  .write(b.toString());`
	`  } else {`
	`  Process`
	`  .stdout`
	`  .write("Hello");`
	`  }`
	`  }`
	`  }`
	`  }`
	`}`

Current Progress

Our AST is currently undergoing considerable refactoring. This is delaying the completion of our typechecker, which is delaying the implementation of the code generation. Some features of codegen are implemented, but cannot be tested, since the corresponding programs cannot make it through the typechecker.

The features that are currently implemented all didn't require any information from the typechecker nor any non-trivial code generation. The former will be done when the ongoing refactoring effort on the rest of the compiler is done, while the latter will be done when we have more tests written to test them. Some easy features also weren't implemented due to a lack of time. Here is a list of the features that are implemented right. They are barely tested and thus might not fully work.

- Print and println statements
- Unary operators
- Binary operators, including `&^` which doesn't exist natively in javascript
- Identifier use as a value in an expression
- String, int and float literals
- Rune literals
- Function declarations and calls
- Assignments
    - Basic assignments
    - Multiple assignments
    - Assignment operators (`+=`, etc.)
- Block statements
- If statements (init clauses not supported yet)
- Type declarations (They simply don't appear in the output Javascript)
- `true` and `false` identifiers

Notably, arrays, slices, loops and short declarations are missing from the above list. Short declarations have been attempted, but involve lots of tricky-to-implement semantics and thus aren't completed yet.