Introduction to the Go compiler

`cmd/compile` contains the main packages that form the Go compiler. The compiler may be logically split in four phases, which we will briefly describe alongside the list of packages that contain their code.

You may sometimes hear the terms "front-end" and "back-end" when referring to the compiler. Roughly speaking, these translate to the first two and last two phases we are going to list here. A third term, "middle-end", often refers to much of the work that happens in the second phase.

Note that the `go/*` family of packages, such as `go/parser` and `go/types`, have no relation to the compiler. Since the compiler was initially written in C, the `go/*` packages were developed to enable writing tools working with Go code, such as `gofmt` and `vet`.

It should be clarified that the name "gc" stands for "Go compiler", and has little to do with uppercase "GC", which stands for garbage collection.

1. Parsing

- `cmd/compile/internal/syntax` (lexer, parser, syntax tree)

In the first phase of compilation, source code is tokenized (lexical analysis), parsed (syntax analysis), and a syntax tree is constructed for each source file.

Each syntax tree is an exact representation of the respective source file, with nodes corresponding to the various elements of the source such as expressions, declarations, and statements. The syntax tree also includes position information which is used for error reporting and the creation of debugging information.

2. Type-checking and AST transformations

- `cmd/compile/internal/gc` (create compiler AST, type checking, AST transformations)

The gc package includes its own AST definition carried over from when it was written in C. All of its code is written in terms of this AST, so the first thing that the gc package must do is convert the syntax package's syntax tree to the compiler's AST representation. This extra step may be refactored away in the future.

The gc AST is then type-checked. The first steps are name resolution and type inference, which determine which object belongs to which identifier, and what type each expression has. Type-checking includes certain extra checks, such as "declared and not used" as well as determining whether or not a function terminates.

Certain transformations are also done on the AST. Some nodes are refined based on type information, such as string additions being split from the arithmetic addition node type. Some other examples are dead code elimination, function call inlining, and escape analysis.

3. Generic SSA

- `cmd/compile/internal/gc` (converting to SSA)
- `cmd/compile/internal/ssa` (SSA passes and rules)

In this phase, the AST is converted into Static Single Assignment (SSA) form, a lower-level intermediate representation with specific properties that make it easier to implement optimizations and to eventually generate machine code from it.

During this conversion, function intrinsics are applied. These are special functions that the compiler has been taught to replace with heavily optimized code on a case-by-case basis.

Certain nodes are also lowered into simpler components during the AST to SSA conversion, so that the rest of the compiler can work with them. For instance, the copy builtin is replaced by memory moves, and range loops are rewritten into for loops. Some of these currently happen before the conversion to SSA due to historical reasons, but the long-term plan is to move all of them here.

Then, a series of machine-independent passes and rules are applied. These do not concern any single computer architecture, and thus run on all `GOARCH` variants. These passes include dead code elimination, removal of unneeded nil checks, and removal of unused branches. The generic rewrite rules mainly concern expressions, such as replacing some expressions with constant values, and optimizing multiplications and float operations.

4. Generating machine code

- `cmd/compile/internal/ssa` (SSA lowering and arch-specific passes)
- `cmd/internal/obj` (machine code generation)

The machine-dependent phase of the compiler begins with the "lower" pass, which rewrites generic values into their machine-specific variants. For example, on amd64 memory operands are possible, so many load-store operations may be combined.

Note that the lower pass runs all machine-specific rewrite rules, and thus it currently applies lots of optimizations too.

Once the SSA has been "lowered" and is more specific to the target architecture, the final code optimization passes are run. This includes yet another dead code elimination pass, moving values closer to their uses, the removal of local variables that are never read from, and register allocation.

Other important pieces of work done as part of this step include stack frame layout, which assigns stack offsets to local variables, and pointer liveness analysis, which computes which on-stack pointers are live at each GC safe point.

At the end of the SSA generation phase, Go functions have been transformed into a series of obj.Prog instructions. These are passed to the assembler (`cmd/internal/obj`), which turns them into machine code and writes out the final object file. The object file will also contain reflect data, export data, and debugging information.

Further reading

To dig deeper into how the SSA package works, including its passes and rules, head to cmd/compile/internal/ssa/README.md.