

Disclaimer

This document is not comprehensive of all Golang features.

Contents

1	.gobc File Format	2
1.1	gobcFile structure	2
1.2	functionInfo structure	3
1.2.1	Function Descriptor	4
1.3	attributeInfo structures	5
1.3.1	codeAttribute structure	5
2	The Interpreter	7
2.1	Operand Stack	7
2.2	Local Variable Table	7

1 .gobc File Format

This document describes the Golang bytecode file format, `.gobc`. Each `.gobc` file contains the bytecode for a Go source file.

A class file consists of a stream of 8-bit bytes. All 16-bit, 32-bit, and 64-bit quantities are constructed by reading in two, four, and eight consecutive 8-bit bytes, respectively.

This document uses a short-hand for specifying the number of bytes associated with the data. The types `u8`, `u16`, `u32`, and `u64` represent a one-, two-, four- or eight-byte quantity, respectively.

This document presents the `.gobc` file format using pseudo-code of C syntax. Arrays are zero-indexed.

1.1 gobcFile structure

A `.gobc` file consists of a single `gobcFile` structure:

```
gobcFile {  
    u32      magicNumber  
    u64      functionCount  
    functionInfo  functions[functionCount]  
}
```

The items that appear in the `gobcFile` structure are defined below:

magicNumber

The `magicNumber` item supplies the magic number identifying the `gobc` file format; it has the value `0xCAFEDEAD`.

functionCount

The `functionCount` item specifies the number of functions defined in the file in the global scope.

functions

The `functions` item is an array where each element is a `functionInfo` structure giving a complete description of the function.

1.2 `functionInfo` structure

Each Golang function is described by a `functionInfo` structure.

TODO: How to handle anonymous and first-class functions?

```
functionInfo {  
    u8          accessFlags  
    u64         nameLength  
    u8          name[nameLength]  
    u64         descriptorLength  
    u8          descriptor[descriptorLength]  
    u32         attributesCount  
    attributesInfo attributes[attributesCount]  
}
```

The items that appear in the `functionInfo` structure are defined below:

`accessFlags`

The value of the `accessFlags` item is a mask of flags used to denote access permission to and properties of this function. The interpretation of each flag, when set, is shown below.

Flag	Value	Description
EXPORTED	0x01	Exported function; can be accessed outside the package

`nameLength`

The `nameLength` item specifies the number of bytes in `name`.

`name`

The `name` item is an array of bytes that comprise the function name. It can be evaluated as a string.

descriptorLength

The `descriptorLength` item specifies the number of bytes in `descriptor`.

descriptor

The `descriptor` item is an array of bytes that comprise the function descriptor. The descriptor describes the type of the parameters and the return types. It should be evaluated as a string. See [Function Descriptor](#) for more details.

attributesCount

The `attributesCount` item specifies the number of attributes of this function.

attributes

The `attributes` item is an array where each element is an `attributesInfo` structure specifying an attribute.

The `functionInfo` structure can contain the following attribute structures:

- [codeAttribute](#)

1.2.1 Function Descriptor

The function descriptor represents the parameters that the function takes and the values that it returns. It should be evaluated as a string. It is described by the following grammar:

```
functionDescriptor -> "(" parameterDesc* ")" returnDesc*
parameterDesc -> Type
returnDesc -> Type
Type -> I64      // int64
      | F64      // float64
      | S        // string
      | V        // void
```

For example, the descriptor `(I64S)SI64` takes two arguments, an int64 and a string, and returns two values, a string and an int64.

1.3 attributeInfo structures

Attributes are attached to various structures to provide more detailed information. All attributes share the following general structure:

```
attributeInfo {  
    u8      attributeType  
    u64      attributeLength  
    u8      data[attributeLength]  
}
```

The items that appear in the `attributeInfo` structure are defined below:

attributeType

The `attributeType` item specifies which attribute the structure represents.

attributeLength

The `attributeLength` item specifies the length of the subsequent information in bytes. The length does not include the initial nine bytes that contain the `attributeType` and `attributeLength` items.

data

The `data` item contains the data of the attribute. Each attribute will have a different structure for this data.

1.3.1 codeAttribute structure

The code attribute is a variable-length attribute in the attributes table of a `functionInfo` structure. A code attribute contains the instructions and auxiliary information for a single function.

```
codeAttribute {  
    u8      attributeType  
    u64      attributeLength
```

	u64	codeLength
}	u8	code[codeLength]

The items that appear in the `codeAttribute` structure are defined below:

attributeType

The `attributeType` item specifies which attribute the structure represents. The value for `codeAttribute` is `0x01`

attributeLength

The `attributeLength` item specifies the length of the subsequent information in bytes. The length does not include the initial nine bytes that contain the `attributeType` and `attributeLength` items.

codeLength

The `codeLength` item specifies the length of the `code` item.

code

The `code` item is an array which contains the instructions of the function. Each instruction must be read 2 bytes at a time. Each instruction has a deterministic length that can be read, which is given in the instruction specification.

2 The Interpreter

A stack-based VM to execute the `gobc` instructions.

2.1 Operand Stack

When a context for execution is created (i.e. when a function is called), a LIFO stack is created for it, called the operand stack. The operand stack is used to store intermediate values to operate on with bytecode instructions, parameters passed to functions, and return values of functions.

Bytecode instructions will push and/or pop onto the operand stack, depending on the instruction.

2.2 Local Variable Table

When a function is called, a local variable table needs to be instantiated to hold the local variables. This will be an array that is zero-indexed. When a function returns from execution, the local variable table should be deallocated.

Instructions that use the local variable table include instructions that contain 'store' and 'load' in the name, such as `i64store` and `i64load`.