GoGo
A Go compiler written in Go

Michael Lippautz
`michael.lippautz@sbg.ac.at`

Andreas Unterweger
`andreas.unterweger@sbg.ac.at`

June 9, 2010

Contents

1 Introduction

Lorem ipsum dolor sit amet...

2 Input Language

Go is a programming language developed by Google, based on a C like syntax and fully specified in [Goo10]. The input language follows the one defined by Go. This results in programs being able to be compiled by the official Go compilers and GoGo.

2.1 Differences to Go

1. GoGo only provides only a **very** basic featureset. Expect every advanced and interesting feature to be missing.
2. GoGo forces the usage of semicolons at the end of statements. This restriction was made to make parsing easier.
3. Go is fully Unicode compatible, while GoGo uses ASCII characters only.
4. Simplified expressions, following Wirth's [Wir96] defintions.

2.1.1 EBNF

Lorem ipsum dolor sit amet...

Atoms

The following listing described the basic atoms that are possible in GoGo programs.

Listing 2.1: Atoms

```
single_char = CHR(32)|...|CHR(127).
char = "'" single_char "'".
string = """ {single_char} """.

digit = "0"|...|"9".
integer = digit {digit}.

letter = "a"|...|"z"|"A"|...|"Z"|"_".
identifier = letter { letter | digit }.
selector = { "." identifier
    | "[" (integer | identifier selector) "]" }.
```

Expressions

Although not as expressive as the ones from Go, these rules define expressions that have comparisons, relations and arithmetical terms.

Listing 2.2: Expressions

```
cmp_op = ">" | "<" | ">=" | "<=" | "==" | "!=".
unary_arith_op = "+" | "-".
binary_arith_op = "*" | "/" .

factor = identifier selector | integer | char | string
    | "(" expression ")" | "!" factor.
term = factor { (binary_arith_op | "&&") factor}.
simple_expression = [ unary_arith_op ] term
    { (unary_arith_op | "||") term }.
expression = "&" identifier selector
    | simple_expression [ cmp_op simple_expression ].
```

Types and Variable Declarations

Listing 2.3: Types

```
type = ([ "[" integer "]" ] identifier | "uint64" | "byte")
    | "string".
var_decl = "var" identifier type [ "=" expression ] ";".
var_decl_list = { var_decl }
```

Structs

Listing 2.4: Structs

```
struct_var_decl = identifier type ";".
struct_var_decl_list = { struct_var_decl }.
struct_decl = "type" identifier "struct" "{"
    struct_var_decl_list "}" ";".
struct_decl_list = { struct_decl }.
```

Statements

Listing 2.5: Statements

```
package_stmt = "package" identifier ";".
import_stmt = "import" string.
import_stmt_list = { import_stmt }.

stmt_sequence = { stmt }
stmt = assignment ";" | function_call_stmt ";" | if_stmt
```

```
    | for_stmt | ";".

assignment = identifier selector "=" expression
if_stmt = "if" expression "{" stmt_sequence "}" [ else_stmt ].
else_stmt = "else" "{" stmt_sequence "}".
for_stmt = "for" [assignment] ";" [expression] ";" [assignment]
    "{" stmt_sequence "}".
```

Functions

Listing 2.6: Functions

```
expression_list = expression { "," expression }.
function_call = "(" [expression_list] ")".
function_call_stmt = identifier selector function_call.

identifier_type = identifier [ "*" ] type.
identifier_type_list = [ identifier_type
    { "," identifier_type } ].
func_decl_head = "func" identifier "(" identifier_type_list ")"  [type].
func_decl = "{" var_decl_list stmt_sequence
    ["return" expression ";"] "}".
func_decl_raw = ";".
func_decl_list = { func_decl_head (func_decl | func_decl_raw)
```

The GoGo Program

Finally, the main program structure is defined by `go_program`. The sequence of the various program parts has been forced to the following to make parsing easier.

Listing 2.7: GoGo Program

```
go_program = package_stmt import_stmt_list struct_decl_list
    var_decl_list func_decl_list.
```

3 Output Language

The output language is Plan-9 assembler [Pik00]. It is a modified version of 64 bit assembly for Intel x86 processors with AT&T syntax that has been created by Bell Labs to be used in their compiler and assembler collection.

4 Scanner

Lorem ipsum dolor sit amet...

5 Parser

Lorem ipsum dolor sit amet...

6 Symbol table

Lorem ipsum dolor sit amet...

6.1 Supported data types

Lorem ipsum dolor sit amet...

6.2 Local variables and offset calculations

Lorem ipsum dolor sit amet...

7 The code generator

Lorem ipsum dolor sit amet...

7.1 Assembly output

Lorem ipsum dolor sit amet...

7.2 Register allocation

The target architecture provides 8 general purpose registers (`R8-R15`) as well as the registers `RAX`, `RBX`, `RCX` and `RDX`[Int09]. The latter are not being used by GoGo to store variables as their values may change when performing arithmetical operations (p.e. `RAX` and `RDX` are always used as the destination registers for multiplications), thus possibly overwriting values previously stored there.
GoGo stores a list for every one of the 8 registers currently free, returning the first free register if required by the code generator. Whenever a register is no longer required (freed), it will be reinserted into the "free" list in order to make it available for future use. Due to the limited amout of registers, the list described is implemented in form of a bit array in the compiler.

7.3 The generation of arithmetical expressions

Lorem ipsum dolor sit amet...

7.4 The generation of assignments

Lorem ipsum dolor sit amet...

7.4.1 The generation of conditional expressions

Lorem ipsum dolor sit amet...

7.5 The generation of loops

Lorem ipsum dolor sit amet...

7.6 The generation of functions

Lorem ipsum dolor sit amet...

7.7 Global variable initialization

Besides the compiled functions from the input file, the code segment contains a function called main.init which performs the initialization of global variables. In contrast to local variables which can be initialized directly at point of their declaration, global variables need to be initialized before any other methods are called, thus requiring the main.init function.
Global variables in general are stored in the data segment, called `data` in the output. They are addressed by their corresponding symbol table offsets relative to the beginning of the data segment. For the special treatment of string constants, please refer to the next section.

7.8 String constants

Strings in Go are 16 bytes in size, containing an 8 byte address (pointer) to its character buffer and an 8 byte length (of which only 4 bytes are used). This makes string length calculations unnecessary and also explains why strings in Go are read-only and have to be reallocated when being changed.
Whenever a string constant is found in the input code, a new byte array with the string's length is declared and initialized with the string's characters. Next, another 16 bytes are allocated which represent the actual string. Using the main.init function as described in the previous section, the string's length and the previously allocated byte array address are assigned to the according offsets of the string in the data segment. When assigning the string constant (or using it as a parameter), an item representing a data segment variable with the string's address is used, therefore eliminating the need for any further special treatment.

8 Library and run time

In order to be able to perform I/O operations and memory management, a library called libgogo is implemented which wraps Linux syscalls and provides an easy to use interface to the GoGo compiler. As GoGo generates assembly code for 64 bit Linux operating systems and the Go compiler allows to mix assembly and Go code, the operating system's built-in functions can be used via syscalls in order to provide the functionality described above.

8.1 I/O syscalls

Besides read and write operations to files (and the console), exiting the program as well as opening and closing files requires the use of syscalls. On Linux 64 bit operating systems with Intel architecture, these syscalls can be invoked by the assembly mnemonic `SYSCALL` where the register `RAX` contains the syscall number defining the syscall, and the registers `RDI`, `RSI` and `RDX` contain the first, second and third parameter respectively[Var08].
The following table lists the syscalls used by libgogo, together with the value of `RAX` representing the syscall number. The latter were derived from the constants defined in `/usr/src/linux-headers-2.6.32-22/arch/x86/include/asm/unistd_64.h` of the current Linux kernel source[Var10]. The syscall function prototypes (for semantics and formal parameters) were derived from the corresponding Linux man pages (see [Var97] and others).

Syscall number (`RAX`)	Syscall function	Purpose
0	`sys_read`	Reads from a file
1	`sys_write`	Writes to a file
2	`sys_open`	Opens a file
3	`sys_close`	Closes a file
12	`sys_brk`	See next section
60	`sys_exit`	Exits the program

8.2 The memory manager

Libgogo provides a very simple memory manager using a bump pointer which can allocate, but not free memory. By using the `sys_brk`[Var97] function, the memory manager expands the data segment of the running program in steps of 10 KB if necessary in order to deal with subsequent allocations.
As the Go compiler used for boot strapping uses a custom memory manager in its run time environment, the libgogo memory manager and the GoGo compiler take measures

to avoid conflicts with the former. First and foremost, all implicit and explicit memory allocations in the GoGo compiler rely on the libgogo memory manager in order to keep the amount of memory allocated by the Go run time constant. Additionally, the memory manager does not allocate any memory in the original data segment to not overwrite any string constants or other information stored there by the Go run time. This is achieved by directly expanding the data segment during the initialization of the libgogo memory manager which also allows to store the first address allocated, thus being able to distinguish between memory allocated by the libgogo memory manager and memory allocated by the Go run time.

8.3 String memory management

Based on the memory manager described above, functions to copy and append strings are implemented in libgogo. As strings in Go are read-only once they are created (or appended), subsequent appending operands require the string to be entirely copied to a new memory location first. In order to avoid this in most cases, the functions handling string appending in libgogo allocate more memory than needed for the current operation in order to be able to reuse this memory in subsequent append operations if the string appended is short enough to fit in the memory already allocated.

Empirical testing showed that it is most convenient to allocate the next power of two of the string length required (p.e. 16 if 9 bytes are initially required). This reduces the number of copy operations and thus memory consumption by more than a power of 10 for very large strings and small appended string (which is common appending code output).

The string manager also has to take care of strings which have been allocated by the Go run time as those strings don't have any "spare" bytes left. Thus, a reallocation of memory for these strings is necessary and cannot be avoided. Subsequent allocations (after the reallocation) can then be dealt with as described above, leading to the performance gains mentioned. The distinction between strings allocated by the Go run time and the libgogo memory manager can be performed easily by comparing the string's address with the first address available to the libgogo memory manager as described in the previous section. If the string's address is smaller, the string is not yet managed by the libgogo memory manager.

8.4 Program parameter determination

Usually, a program's parameters are accessible through `argc` and `argv`, positioned on the stack as function parameters of the main (entry) function. As the Go compiler used for bootstrapping adds run time routines which are invoked before the main function, the parameters cannot be fetched from the stack as their position cannot be determined. The Go compiler allows to access these parameters using a separate package (library) which could not be used in order to remain independent of third party libraries.

The current approach to fetch parameters requires the activation of the `proc` file system in the Linux kernel which is enabled by default in most Linux distributions[Var06]. The `proc` file system allows (among other information) to access parameters of all processes running in the system, including the current process. The latter's parameters can be through the virtual file `/proc/self/cmdline` where `self` refers to the current process. The virtual file contains all parameters, separated by zero bytes and terminated by a sequence of two zero bytes. When parsed, this allows to access the program's parameters without having to access the stack.

9 Building

Lorem ipsum dolor sit amet...

10 Testing

In order to test the compiler, a test suite has been constructed that may be used to verify results against an already existing result set.

The test suite offers the following functions:

- **newvalids/ackvalids/fullclean** – These commands are used to create a new result set as reference for further tests. While `fullclean` deletes the old set, `newvalids` is used to create a new one. After verifying that the compiled output is correct (by manually checking it), the command `ackvalids` can be used to acknowledge the set (resulting in a checksum file).
- **test/clean** – `test` is used to perform a compilation and compare the results against the last valid result set. In order to do so, checksums of the tests are compared. If they are not equal, a `diff` is printed to the user.

Bibliography

[Goo10] Google Inc. The Go Programming Language Specification. `http://golang.org/doc/go_spec.html` (4.6.2010), 2010.

[Int09] Intel Inc. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M. `http://www.intel.com/products/processor/manuals/index.htm` (9.6.2010), 2009.

[Pik00] Pike, R. A Manual for the Plan 9 assembler. `http://doc.cat-v.org/plan_9/4th_edition/papers/asm` (4.6.2010), 2000.

[Var97] Various. brk(2) – Linux man page. `http://linux.die.net/man/2/brk` (9.6.2010), 1997.

[Var06] Various. proc(5) – Linux man page. `http://linux.die.net/man/5/proc` (9.6.2010), 2006.

[Var08] Various. syscalls(2) – Linux man page. `http://linux.die.net/man/2/syscalls` (9.6.2010), 2008.

[Var10] Various. The Linux Kernel. `http://www.kernel.org` (9.6.2010), 2010.

[Wir96] Wirth, N. *Compiler Construction.* Addison-Wesley, 1996.