GOMscript User's Guide

B. Ban
IBM Zurich Research Laboratory
8803 Rueschlikon
email: bba@zurich.ibm.com

September 8, 1998

Contents

Chapter 1

Introduction

GOMscript is an interpreter for a simple C++-like language. It contains the usual control flow statements such as `if-then-else`, `while` and `for`, and allows to define functions and classes. GOMscript may be easily extended by adding functions and classes located in a shared library (an *extension*).
Contrary to C++, GOMscript is untyped. This means that values will only be assigned an (implicit) type at runtime and not at compile time as in C++. This gets rid of the necessity of a compiler, resulting in faster development, but also adds some overhead to the running program. However, GOMscript was not written to compete with C++, but to produce a tool that allows to interactively put together small applications and to write simple scripts.

1.1 Design

GOmscript was written using bison and flex[1] The runtime was written using a number of C++ classes representing expressions, statements, the symbol table and so on (see file `Grammar.h`).
The basic idea is that for each production a corresponding syntax tree consisting of instances of these C++ classes is constructed and subsequently traversed, interpreting the instances along the way. For this purpose, each class has a virtual method `Interpret`. A `BinExpr` for example has two children: a left and a right subexpression. When `Interpret` is called on an instance, it will call `Interpret` on both its left and right subexpression, add the two results and return the total. When done interpreting the tree, it is deleted and the tree for the next production is created.
All variables and function definitions are stored in the *symbol table* which records bindings between identifiers (e.g. variable or function names) and values (of variables, or definition of functions). Each new *scope* (c.f. 2.5) creates a separate nested symbol table in which local bindings are stored. A nested symbol table has a reference to its predecessor so that, if a symbol is not found in the local scope, it can be searched in the enclosing scope[2]. When a scope is deleted, its assciated symbol table is deleted as well.
There is one global symbol table which is the father of all scopes and which does not have a predecessor. This table is only deleted when the interpreter exists.
All built-in types have the ability to be saved to and re-read from a file. The method `Dump` stores the contents of an instance to a stream and `Read` reconstructs the contents from a stream. If user-defined classes need to be persistent as well, these two methods have to be overridden.
Persistence of values allows to stream the contents of the symbol table to a stream (e.g. a file or UNIX socket) and to reconstruct the contents at a later time. This enables the interpreter to save its state[3] and to continue from this state at a later time. An application of this capability is shown in section 4.

[1] Actually, the first version used lex and yacc, but bison and flex are more flexible when a parser has to be written that can be called recursively.
[2] For a more detailed description of scopes refer to section 2.5
[3] In this respect, GOMscript is similar to Smalltalk which saves its state in an *image* file.

1.2 Installation

The GOMscript executable and suppport files (extensions) is tarred and gzipped. To install it,

1. Unzip the file: `gzip -d gomscript-linux-2.0.tar.gz` and
2. Untar the resulting tar file: `tar -xvf gomscript-linux-2.0.tar`

The directory `./gomscript` will be created which contains all the necessary files. Directory `vbx` contains the extensions. To start the interpreter, type `main`. GOMscript will try to load all the extensions in subdirectory `vbx` that end in `*.dll`. After doing so, a prompt signals that GOMscript is now ready to accept input. A very simple interaction with the interpreter is shown below:

```
> a=23;
> a;
> 23
> println "Hello world";
Hello world
> for(i=0 to 3) {println i;}
0
1
2
3
> quit
This is the end.

VBXs were unloaded.
```

Typing `main -help` shows the command line parameters the interpreter accepts.

Chapter 2

Syntax

The syntax of GOMscript is similar to C++ (or Java) and was deliberately kept simple.

2.1 Types

GOMscript knows the following built-in types:

Null Represents 'no' value. Is often returned in the case of a failure.

Number Both real and inteer number values can be represented through this type. Example: `3`, `3.145`.

String Example: `"Hello world"`

List A collection of elements, e.g. `#(1, 2, 3)`, `#("Bela", "Janet", 3.23, null)` or `#(1, 2, #(-34, 4.3), 56)`. Lists can also be created using the `new` operator: `a=new List(1,2,3);`

Struct Represents a struct which contains other named values. Each member has a name and a value: `a=new Struct("name", "Bela", "age", 31);`.

Union Represents a C-like union with a name and a value: `a=new Union("age", 31);`.

The types `List`, `Struct` and `Union` are examples of builtin *classes* with methods and instance members. The `inspect` statement lists all the methods implemented by an instance's class: `inspect a;`. All builtin classes are defined in the file `BuiltinClasses.h`.

2.2 Identifiers

Identifiers denote variables, function and class names and have to consist of the characters a-zA-Z plus the underscore ('_') and period ('.') characters, and digits. Digits may however not be in the first position. Valid identifiers are e.g. `a`, `number_of_bytes`, `a.Length` and `person.age`.

2.3 Expressions

The following expressions are present in GOMscript:

Expression	Description
expr + expr	Addition
expr - expr	Subtraction
expr * expr	Multiplication
expr / expr	Division
expr MOD expr	Modulo
- expr	Unmary minus
(expr)	Parentheses
identifier	Value of identifier
NUMBER	Number
STRING	String
NULL	Null
List	List
Function call	Value of function call
NEW	New expression

2.3.1 Boolean Expressions

The following boolean expressions are present in GOMscript:

Expression	Description
true	Value for true
false	Value for false
(expr)	Grouping
not expr	Inversion of expr
expr and expr	Boolean AND
expr or expr	Boolean OR
expr < expr	Less than
expr <= expr	Less than or equal
expr > expr	Greater
expr >= expr	Greater than or equal
expr == expr	Equal
expr != expr	Not equal
expr <> expr	Not equal

2.4 Statements

Each program consists of a (possibly empty) set of statements. Each statement is terminated by a semicolon ';' character. GOMscript starts at the first statement and processes all statements in the order they are read until no more statements are left (or the user terminates the program). The following statements are available in GOMscript:

Statement	Description
ident = expr	Assigment
block	Block statement (2.4.1)
expr	Expression
vars	Definition of variables in class or function scope
function def	Function definition (2.4.9)
class def	Class definition (2.4.10)
`quit`	Exits the interpreter
`symbols`	Shows all symbols that are currently defined in the symbol table (2.4.2)
`classes`	Shows all the classes currently defined
`inspect` expr	Shows variables and methods of an instance
`reload`	Reloads all extensions
`load` string	Interpretes the contents of the string
`print(ln)`	Prints a variable or value
`dump` string	Dumps the contents of the symbol table to file string
`dump`	Dumps the contents of the symbol table to the file "symtab.db"
`read` string	Populates the symbol table with the values stored in file string
`read`	Populates the symbol table with the values stored in file "symtab.db"
while	while statement (2.4.3)
if	if statement (2.4.4)
for	for statement (2.4.5)
return	return statement (2.4.6)
break	break statement (2.4.7)
`eval` string	Interprets the contents of string (2.4.8)

2.4.1 `Block` Statement

A block is a sequence of statements separated by semicolons.

2.4.2 `Symbols` Statement

This statement prints the contents of the symbol table, i.e. all bindings between identifiers and their values. Identifiers may denote both variables and functions. Classes are *not* stored in the symbol table, but are located in a separate table. The contents of this table can be printed using the `classes` statement.
Note that both the `symbols`- and the `classes`-statements do *not* have a terminating semicolon !

2.4.3 `While` Statement

```
while( <boolean expression> ) { <block> }
```

The `while` statement evaluates the boolean expression as long as it is true. When it evaluates to false, processing continues after the `while` statement.
Example:

```
a=1;
while(a < 10) {
    println a;
    a=a+1;
}
```

2.4.4 If Statement

```
if( <boolean expression> ) { <block> }
[ else { <block> } ]
```

The boolean expression is evaluated. If it is true, the block after the `if` statement is interpreted, otherwise the block after the `else` statement is interpreted. The `else` part is optional.
Example:

```
if(number_of_bytes == -1) {
    println "Error occurred: Number of bytes is -1";
}
else {
    println "Number of bytes read: " + number_of_bytes;
}
```

2.4.5 For Statement

```
for( <identifier> = <expr> (to | downto) <expr> { <block> }
```

Used to iterate through a sequence. The identifier is assigned the result of the first expression which must evaluate to a number (start number). The second expression evaluates to another number which identifies the stop number. The block is evaluated n times with identifier being incremented each time.
Example:

```
for(i=0 to l.Length()-1) {
    println "Element at " + i + " is: " + l.At(i);
}
```

2.4.6 Return statement

This statement returns from a function or method.
Example:

```
define sqr(x) {
    return x*x;
}
```

2.4.7 Break Statement

Breaks out of an enclosing `for`- or `while` loop and continues processing at the statement following the enclosing statement.
Example:

```
a=1;
while(true) {
    println a;
    a=a+1;
    if(a == 10) {
        break;
    }
}
```

2.4.8 `Eval` Statement

Evaluates the string that is the argument of the `eval` statement.
Example:

```
a="println \"Hello world\"; ";
eval a;
```

2.4.9 `Function` Definition

```
define <funcname> ( <parmlist> ) { <block> }
```

Allows to define functions. A function has a parameter list and a (possibly empty) seqeuence of statements (a block). An (optional) `vars` statament allows to define function-local variables. A function-local variable will never refer to the global symbol table to resolve a variable name (an identifier), but rather try to resolve the name in the function-local symbol table.
Example:

```
define Size(my_list) {
    vars: length;  // local to function
    length=my_list.Length();
    return length;
}
```

The variable `length` will *not* be looked up in the global symbol table if not found in the local one (even if a variable named `length` was present in the global symbol table). If the `vars` statement was absent, the value of `length` would first be looked up in the local scope and, if not present, subsequently be looked up using the global symbol table.
Functions are values and can therefore also be used as arguments to a function call, e.g.:

```
define sqr(x) {return x*x;}
define apply(function, arg) {return function(arg);}

> apply(sqr, 34);
> 1156
```

To avoid side effects, arguments of a function may be modified inside the function scope, but they will not have any effect outside of the function (call-by-value). Call-by-reference is not possible in GOMscript. Therefore semantics of call-by-reference have to be achived through function return values.

2.4.10 Class Definition

```
class <classname> [inherits from <classname>]
{ [<vardecls>] [<methods>] }
```

GOMscript allows to define classes which may inherit member variables and methods from superclasses. GOMscript only implements single inheritance. As with functions *class-local* variables can be defined using the `vars` statement. All variable defined with this statement will only be resolved in class scope and will never consult the global symbol table.
Methods inside class definitions are defined as normal functions with the exception that an additional scope (class scope) is present (c.f. section 2.5 for a detailed discussion of scopes.)
Example:

```
class Person {
    vars:
        _name, _age;
    methods:
        define Show() {
            print "Name: "; println _name;
            print "Age:  "; println _age;
        }
}

class Bela inherits from Person {
    vars:
        _salary;
    methods:
        define Bela(name, age, salary) {
            _name=name;
            _age=age;
            _salary=salary;
        }
        define Show() {
            super.Show();
            print "Salary: "; println _salary;
        }
}

> bela=new Bela("Bela Ban", 30, 48000);
> bela.Show();
Name: Bela Ban
Age:  30
Salary: 480000
>
```

Methods in GOMscript classes cannot be overloaded, that is there cannot be more than one method of the same name in a class. A subclass, however, may override a method of the same name in the superclass. If a subclass wants to invoke a method with the same name in the superclass, the keyword `super` can be used (as shown in the above example) as receiver of the message.
An instance of a class can be created using the `NEW` statement (as shown above). Arguments to initialize a new instance can be given to a `constructor`. A constructor is a method that has the same name as the class and a number of parameters. This method will be invoked immediately after the new instance has been created, and before it is returned from the `NEW` call. There can only be one constructor per class.

2.5 Scope of Identifiers

A *scope* is the area within which an identifier has a value (a *binding*). Scopes are always implemented through symbol tables which have a number of bindings between an identifier and a value. An identifier can be a variable- or function.

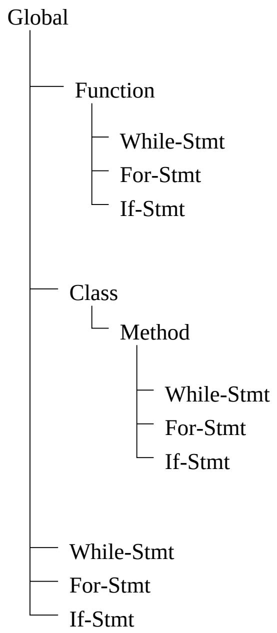

Figure 2.1: Scopes in GOMscript

Fig. 2.1 shows how scopes can nest in GOMscript. The highest scope with a lifetime equal to that of the interpreter itself is the global scope. It contains all global variables plus all functions[1].
Functions, classes and `If-`, `For-` and `While` statements open another nested scope. Identifiers will be tried to resolved in this scope. If this fails, the global scope is used for resolution[2].
A class has methods which open another scope. Each method contains *method-local* variables that only exist for the duration of the method call.
Identifiers from a certain scope can always refer to identifiers of their enclosing scopes (unless otherwise specified).

[1] There is another global symbol table that contains all the classes, but is not shown in the figure.
[2] Unless the `vars` statement is used in function- or class definitions.

Chapter 3

Extensions

GOMscript allows to be extended in an easy way through user-defined functions and classes located in shared libraries. All extensions have to obey a certain format and are read by GOMscript at startup time. Upon successful reading of the user-defined extensions, all functions and classes contained therein will be accessible from within the interpreter.

3.1 Functions

3.2 Classes

3.3 Examples

3.3.1 Implementing Sockets as Extensions

3.4 Interacting with CMIP Agents

3.5 Interacting with DSOM Instances

Chapter 4

Roaming Agents

Chapter 5

Appendix

5.1 Definitions

5.1.1 Common Definitions for Extensions

```
#ifndef DEFS_H
#define DEFS_H

#include "Grammar.h"

typedef CL_List<Expr*> Elist;
typedef NativeInstance* (*Constructor)(Elist&);
typedef void (*FuncPtr)();

struct DLL_Info {
  void*    dll_handle;
  FuncPtr  unload_func;
};

struct FuncDescr {
  char*         name;
  void*         addr;
  int           number_of_args;
  BType*        parm_types;     // Array of BTypes...
};

struct MethodDescr {
  char*         name;
  MethodPtr     mptr;
  int           number_of_formal_parms;
  BType*        parm_types;     // Array of BTypes
};

struct ClassDescr {
  char*         name;
  char*         superclass;
  Constructor   ctor;
  MethodDescr*  methods;
};

struct Contents  {
  FuncDescr*    functions;
  ClassDescr*   classes;
  FuncPtr       load_func;    /* called when DLL is loaded   */
```

```
  FuncPtr       unload_func;  /* called when DLL is unloaded */
};

#endif
```