GOLD Language Reference Manual

Language Guru: Timothy E. Chung (tec2123)
System Architect: Aidan Rivera (ar3441)
Manager: Zeke Reyna (eer2138)
Tester: Dennis Guzman (drg2156)

October 16th, 2017

1 Introduction

GOLD is a language very similar in syntax to C. We aim to create a language in which the description of command-line games may be made easier. In order to fulfill our goal we have placed special importance a form of structure which we have appropriately named a “type-struct”. This associates a name to the type of struct which can hold named fields to describe more abstract concepts. Functions, too, have an alteration in which they can be associated with a particular type of type-struct, and can access its members through the keyword “self”. We hope that these two additions into a C-like language, bringing an association of actions with some form of entities, can aid in the creation of command line games.
We supply default functions with which to interact with strings and the command line interface, and in the future we will include a package which would supply a default or suggested interface of functions.

2 Lexical Conventions/Tokens

2.1 Comments

Comments in GOLD occur between the token “//” and the end of the line.

2.2 Identifiers

An identifier in GOLD consists of an alphabetic character or underscore followed by a sequence of alphanumeric characters and/or underscores “_”.

2.3 Keywords

These identifiers are reserved and can only be used as keywords:

`if`	`elif`	`else`	`func`	`return`	`for`
`int`	`string`	`float`	`true`	`false`	`type`
`struct`	`break`	`continue`	`null`	`print`	`input`
`bool`	`self`	`void`			

2.4 Constants

2.4.1 Integer Literals

An integer consists of a sequence of digits and will be represented as that string's decimal value.

2.4.2 Floating Point Literals

A float will be represented by a sequence of one or more digits, a decimal point, and a sequence of one or more digits. At least the integer part must be present in addition to the decimal point, but can take a value of 0.

2.4.3 Character Literals

A character literal will consist of a single character, or a backslash followed by one of {`"`, `r`, `n`, `t`,} or a sequence of digits whose decimal value falls between 0 and 127, inclusively, forming a character's ASCII representation. " `\"` " , " `\r` ", " `\n` ", and " `\t` " have their conventional meaning.

2.4.4 Boolean Literals

A boolean literal can be either of the keywords `true` and `false`.

2.4.5 String Literals

String literals consist of a sequence of characters surrounded by **“”**. Any of the escapes in 2.4.3 can be included in the string.
The creation of new strings from other variables is covered in the library functions section (8.2).

3 Types

All types are null if declared but undefined.

3.1 Primitive Types

bool: a boolean value, can take either a true, false, or null value
int: an integer numerical value
float: a floating point numerical value, consisting of a decimal point and at least one of either an integer or fractional part
string: a sequence of characters as defined in (2.4.5).
func: a construct taking in arguments and returning a value of predefined type

3.2 Derived

Arrays: represented as [], a 0-indexed list of predefined-typed variables of fixed size
Pointers: an address location of any other type in memory

3.3 Type-Structs

A GOLD specific concept, these consist of a set of capitalized, named strings, which map to values of predefined type. All are required at definition, but can be instantiated to null.
Ex.

```
type User struct {
      Name string;
      Age int;
}
User U = {Name=”Stephen”; Age=null;};
```

The fields and type-name of the type-struct must be capitalized at definition.

4 Expressions

If an undefined identifier is accessed in an expression, an Error is raised.

4.1 Unary Operators

`-expr (num)`, evaluates to the negation of either a floating point or integer value; -0 -> 0
`!expr (boolean, boolean expression),` evaluates to the negation of a boolean value,

```
!true == false // returns true
!false == true // returns true
```

`&expr (identifier)`, evaluates to the address of the identifier in memory if the expression is an identifier, otherwise return an error, the expression can also be a function
`*expr (identifier)`, evaluates to an identifier to the object or literal that identifier was pointing to. If the identifier doesn't reference a pointer, then a dereference error will be thrown.

4.2 Numeric Binary Operators

`expr + expr (num)`, adds either float to float or int to int
`expr - expr (num)`, subtracts the right numerical value from the left numerical value
`expr / expr (num),` divides the right value from the left, returning its integer value if integers are used
`expr * expr (num),` multiplies the numerical values
`expr % expr (num)`, takes the remainder of division of the left value by the right

These group from left to right and do not have side-effects on the variables involved unless they evaluate to the right hand expression of an assignment. These operations can not be crossed between ints and floats, as GOLD is a strongly typed language.

4.3 Relational Operators

`expr < expr`, less than (num)
`expr > expr`, greater than (num)
`expr >= expr`, greater than or equal to (num)
`expr <= expr`, less than or equal to (num)
These operators return true if the relation is true and the types are the same, and false otherwise. If types are mismatched an error is thrown.

4.4 Equality Operators

`expr == expr (num)`, returns true if the numbers are equal to each other and the same type, false if they are not equal, and an error for a type mismatch.
`expr != expr (num)`, returns true if the numbers are not equal, false if they are, and an error for a type mismatch.

4.5 Logical Operators

`expr && expr (boolean)`, returns true if both the expressions are true, and false otherwise.
`expr || expr (boolean)`, returns true if either expr is true, and false otherwise.

4.6 Assignment Operator

`lvalue = expr` replaces the left-hand value with the expression if their types match, otherwise it throws up an error. If also declaring a variable whilst assigning a value to it, the type of the variable must be prepended to it.
Ex. `<variable-type> <variable-name> = expr`

4.7 Subset Operator

`<Array>[<index value>]` This operator requires an identifier in the <Array> position, and an int i that satisfies 0 < i < length_of_array for the <index value>. Expression would evaluate to the 0-indexed value of the list.

4.8 Precedence of Operators

Order of Precedence is from to to bottom,

<Operator Type>	<Operator>	<Associativity>
Subset Operator	[ ]	left-to-right
Unary Operators	*, &	right-to-left
	!, -(negation)	
Numeric Binary Operators	+, -(subtraction)	left-to-right
	*, /, %	

Relational Operators	<, >, <=, >=	left-to-right
	==, !=	
Logical Operators	&&, ||	left-to-right
Assignment Operator	=	right-to-left

5 Declarations

5.1 Type specifiers

`int, string, float, bool, void, type <name> struct`

"`void`" should be specified as the return type of a function if no value is to be returned from it.
If a struct is declared, a type name must be specified as a sequence of alphanumeric digits or "_" that begins with a capitalized alphabetic character. If assigning a variable as a struct, it is proper convention to capitalize the beginning of the variable's name, but is not required.

These type specifiers define types of variable, parameter types and return types. "void" is exclusively a return type.

5.2 Object Declarators

Each object declaration statement takes the form of

```
<type> <variable-name>;
```

Where the variable-name is the identifier of the object. This holds true of all types but "`void`".

5.3 Function Declarators

```
func [type-struct type] <function-name>(arg-name arg-type, ... ) <return-type>
{... }
```

A function declaration consists of the func keyword followed by an optional type-struct association, the function name, parentheses optionally filled by a sequence of argument names and types, separated by commas. The return type must be specified after, with statements in the function surrounded by braces.
If the function has a type-struct association, the function is to be used by calling the defined variable's identifier of the type-struct, appended with a ".", and then appended with the function name, parentheses, and any required arguments; see (7.2 for example).

5.4 Array Declarator

Ex. `<type-name>[<size>] <identifier>;`
Or
Ex. `[<type-name>[] ] <identifier> = <type-name>[<value>, ….];`

The first example shows the declaration of an array of type “type-name”; the size (in number of values) must be specified. The second illustrates the declaration and definition of an array. If the array identifier is already instantiated as the right size and type, the <type-name>[] is optional.

6 Statements

6.1 Assignment Statement

An assignment statement can take the form of
Ex. [<identifier-type>] <identifier> = expr
Where if the <identifier> is already declared, the <identifier-type> can be omitted, but the expression must be of the correct type. If the <identifier-type> is specified, the expression’s type can overwrite the previous type of the identifier.

6.2 Function-Call Statement

If a function is defined as having a type-struct association, a function is called in the format
Ex. <type-struct identifier>.<function-name>([correctly typed arguments, ….]);
Or
Ex. [identifier type] <identifier> = ...(see above)
and the type-struct variable and variable members will be accessible by the keyword “self” in the function.

If a function does not have a type-struct association, then a function-call statement takes the form of
Ex. <function-name>([correctly typed argument, correctly typed argument, ...]);
Or
Ex. [identifier type(optional)] <identifier> = <function-name>([args...]);

The latter function-call in each case will assign the result of the function to the identifier, or declare an identifier of the correct type if the types match and the type is specified. If the identifier is already being used, the type specification will overwrite the variable; if no type is given and the return type of the function does not match that of the existing variable, an Error will be raised.

6.3 Sequence Statement

Statements occur in sequence or in singular fashion, separated by semicolons. Alternately, a semicolon can be used by itself in absence of a statement

Ex. statement ; statement; statement ….; or ;

6.4 Control-Flow Statement

- if, elif, else

The control flow statements evaluate expressions to determine which block of statement(s) to execute. There must be at least one statement, or a semicolon in absence of statements.

Ex. if (expr) {statement}

Ex. if (expr) {statement} else {statement}

Ex. if (expr) {statement} elif (expr) {statement} ….

Ex. if (expr) {statement} elif (expr) {statement} …. else {statement}

6.5 Loop Statement

- for

for (initial expr; conditional expr; iterative expr) { statement }

GOLD has no while loop, only a “for” loop. A for loop begins with its initial expression, which can be blank, can set an existing variable to some value, or can instantiate a new variable, in a C-like style. A for loop without any conditional expression will run forever like a while loop that always evaluates to true. At the end of each statement block, the conditional expr is evaluated, which if false will cause the end of the loop. Otherwise, the iterative expression is evaluated and the statement block begins again.

7 Library Functions

7.1 print

Prints the string value to stdout, without a newline character.

7.2 println

Prints the string value to stdout, with a newline character.

7.3 sprint

Returns a string from a formatted string and variable input
Ex. string s = sprint("%s am %i feet tall", "I", 6);

Signifier	Variable Type
%s	Replaces with a string
%i	Replaces with an integer
%f	Replaces with a float
%p	Replaces with an address

7.4 input

This function waits until the enter key is pressed to return the string representation of user input (purely string of characters, no escapes)

Ex. string s = input(); //takes user input

7.5 len

When performed on an array, returns the integer size of the array.
When performed on a string, returns the integer size of the string.
Ex. len("Hi"); ---> 2
Ex. int[] a = [0, 1, 2]; len(a); ----> 3

8 Scope

Local Variables - Local variables are declared inside a function or block and can only be accessed within that function or block
Actual Parameters - These are the parameters declared in the function or method signature and can be accessed anywhere within the function/method
Global Variables - These are the variables declared outside any function and can be accessed anywhere within the package or by external packages that import the package in which they exist

9 Examples

```
------------------------------------------------------------
public func main() int {
	int a = 99;
	string b69 = “ bottles of beer”;
	string not_vans = “ on the wall”;
	string sharing = “take one down, pass it around,”

	for (int i = a; i > 0; i = i - 1) {
		string temp = sprint(“%i”, i);
		println(sprint(“%i%s%s”, i, b69, not_vans));
		println(sprint(“%i%s,”, ib69));
		println(sprint(“%s”, sharing));
		println(sprint(“%i%s%s\n”));
	}
	string s = sprint(“%s %s\n”, s1, s2);
	print(sprintf(“%s %s\n”, s1, s2));
	return 0;
}
------------------------------------------------------------

type Account struct {
	Owner string;
	Balance float;
	DateCreated string;
}

//Valid declaration inside and/or outside of main function
//Account Primary = {
		//Owner = “Timothy”;
		//Balance = 99999.99;
		//DateCreated = “09:07:1997T13:22.046Z”;
	//};

func Account deposit_10000() int {
	//this.float += 10000; //Would throw error because is an int, not a float
	this.float += 10000.;
	return 0;
}
```

```
public func main() int {
	Account AccountDeTim = {
		Owner =  “Timothy”;
		Balance = 99999.99;
		DateCreated = “09:07:1997T13:22.046Z”;
	};
	if (AccountDeTim.deposit_10000() != 0) {
		return 1;
	} else {
		println(sprint(“%s’s new balance is %f”, AccountDeTim.Owner,
	AccountDeTim.Balance));
	}

	return 0;
}
```

10 Recapitulation

GOLD aims to be a simple language that takes advantage of the concept of struct, pointers, and easy pointer manipulation by removing the complex rules of pointer manipulation one sees in C or C++. It allows for creation of type struct and at the same time, struct function/method that one can use for a specific struct. This allows us to take advantage of object oriented concept using structs. GOLD aims to keep the standard library as thin as possible like C so it is easy to learn and start with.