GoLF Specification
Computer Science 411

John Aycock

Table of Contents

Overview

GoLF stands for “Go Language Fragment” and is a subset of the Go programming language, with minor changes to make it more amenable to implementation in a one-semester course. If you've never used Go previously, don't panic: it draws heavily from C and will be fairly familiar as a result. You also have access to a reference implementation of GoLF that you can run test inputs through to see the result.

For CPSC 411, **do not use the original Go specification**; use this GoLF specification instead.

Introduction

This is the reference manual for the GoLF programming language, a general-purpose language that is strongly typed. The grammar is compact and simple to parse, allowing for easy analysis by automatic tools such as integrated development environments.

A few simple examples, starting with Hello, world:

```
func main() {
    prints("Hello, world!\n")
}
```

Fibonacci series (recursive):

```
func main() {
    var i int
    i = 0   // not strictly needed: see "zero value"

    // Anything larger than 47 overflows a 32-bit int...
    //
    for i <= 47 {
        prints("fib(")
        printi(i)
        prints(") = ")
        printi(fib(i))
        prints("\n")
        i = i + 1
    }
}

func fib(n int) int {
    if n == 0 { return 0 }
    if n == 1 { return 1 }
    return fib(n-1) + fib(n-2)
}
```

Notation

The syntax is specified using Extended Backus-Naur Form (EBNF):

```
Production  = production_name "=" [ Expression ] "." .
Expression  = Alternative { "|" Alternative } .
Alternative = Term { Term } .
Term        = production_name | token [ "…" token ] | Group | Option | Repetition .
Group       = "(" Expression ")" .
Option      = "[" Expression "]" .
Repetition  = "{" Expression "}" .
```

Productions are expressions constructed from terms and the following operators, in increasing precedence:

```
|   alternation
()  grouping
[]  option (0 or 1 times)
{}  repetition (0 to n times)
```

Lower-case production names are used to identify lexical tokens. Non-terminals are in CamelCase. Lexical tokens are enclosed in double quotes `""` or back quotes ``` `` ```.

The form `a … b` represents the set of characters from `a` through `b` as alternatives. The horizontal ellipsis … is also used elsewhere in the spec to informally denote various enumerations or code snippets that are not further specified. The character … is not a token of the GoLF language.

Source code representation

Source code is 7-bit ASCII text, and input characters outside that range may not appear in GoLF programs. That being said, character values in this specification will be given in Unicode notation for consistency with the original Go specification. Each input character is distinct; for instance, upper and lower case letters are different characters.

Implementation restriction: for compatibility with other tools, a compiler may disallow the NUL character (U+0000) in the source text.

Characters

The following terms are used to denote specific ASCII character classes:

```
newline        = /* the character U+000A */ .
ascii_char     = /* an arbitrary ASCII character except newline */ .
ascii_letter   = "A" … "Z" | "a" … "z" .
ascii_digit    = "0" … "9" .
```

Letters and digits

The underscore character _ (U+005F) is considered a letter.

```
letter        = ascii_letter | "_" .
decimal_digit = "0" … "9" .
```

Lexical elements

Comments

Comments serve as program documentation. GoLF has one form of comment, line comments that start with the character sequence `//` and stop at the end of the line. A line comment acts like a newline.

A comment cannot start inside a string literal, or inside a comment.

Tokens

Tokens form the vocabulary of the GoLF language. There are four classes: *identifiers*, *keywords*, *operators and punctuation*, and *literals*. White space, formed from spaces (U+0020), horizontal tabs (U+0009), carriage returns (U+000D), and newlines (U+000A), is ignored except as it separates tokens that would otherwise combine into a single token. Also, a newline or end of file may trigger the insertion of a semicolon. While breaking the input into tokens, the next token is the longest sequence of characters that form a valid token.

Semicolons

The formal grammar uses semicolons `";"` as terminators in a number of productions. GoLF programs may omit most of these semicolons using the following two rules.

1. When the input is broken into tokens, a semicolon is automatically inserted into the token stream immediately after a line's final token if that token is
    - an identifier

- an integer or string literal
- one of the keywords `break` or `return`
- one of the punctuation symbols `)` or `}`

2. To allow complex statements to occupy a single line, a semicolon may be omitted before a closing "}".

To reflect idiomatic use, code examples in this document elide semicolons using these rules. Also note that either a newline *or* end of file may follow a line's final token.

Fun fact: the full Go compiler sets the semicolon token's lexeme to ";" if the semicolon was actually in the input, and "\n" if the semicolon was inserted by the compiler.

Identifiers

Identifiers name program entities such as variables. An identifier is a sequence of one or more letters and digits. The first character in an identifier must be a letter.

```
identifier = letter { letter | ascii_digit } .
```

Examples:

```
_
a
_x9
```

Some identifiers are predeclared.

Keywords

The following keywords are reserved and may not be used as identifiers.

```
break
else
for
func
if
return
var
```

Operators and punctuation

The following character sequences represent operators (including assignment operators) and punctuation:

```
+    &&    ==    !=    (    )
-    ||    <     <=    {    }
*          >     >=    ,    ;
/          =
%          !
```

Integer literals

An integer literal is a sequence of digits representing an integer constant.

```
int_lit        = decimal_lit .
decimal_lit    = decimal_digits .
decimal_digits = decimal_digit { decimal_digit } .
```

For example:

```
42
0600        // this would be interpreted as the base 10 number 600, unlike Go
170141183460469231731687303715884105727

_42         // an identifier, not an integer literal
```

String literals

A string literal represents a string constant obtained from concatenating a sequence of characters. There is one form of these in GoLF: interpreted string literals.

Interpreted string literals are character sequences between double quotes, as in `"bar"`. Within the quotes, any character may appear except newline and an unescaped double quote. The text between the quotes forms the value of the literal, with backslash escapes interpreted as follows:

```
\b   U+0008 backspace
\f   U+000C form feed
\n   U+000A line feed or newline
```

```
\r   U+000D carriage return
\t   U+0009 horizontal tab
\\   U+005C backslash
\"   U+0022 double quote
```

All other sequences starting with a backslash are illegal.

```
string_lit             = interpreted_string_lit .
interpreted_string_lit = `"` { ascii_value } `"` .

ascii_value    = ascii_char | escaped_char .
escaped_char   = `\` ( "b" | "f" | "n" | "r" | "t" | `\` | `"` ) .
```

Constants

There are *boolean constants*, *integer constants*, and *string constants*. A constant value is represented by an integer or string literal, or an identifier denoting a constant. The boolean truth values are represented by the predeclared constants `true` and `false`.

Implementation restriction: Although numeric constants have arbitrary precision in the language, a compiler may implement them using an internal representation with limited precision (although no less precision than the `int` type). That said, every implementation must give an error if unable to represent an integer constant precisely.

Variables

A variable is a storage location for holding a value. The set of permissible values is determined by the variable's *type*. A variable declaration reserves storage for a named variable; for function parameters, the signature of a function declaration reserves storage for those parameters as named variables.

The type of a variable is the type given in its declaration.

A variable's value is retrieved by referring to the variable in an expression; it is the most recent value assigned to the variable. If a variable has not yet been assigned a value, its value is the zero value for its type: `false` for booleans, `0` for numeric types, `""` for strings.

Types

A type determines a set of values together with operations and methods specific to those values. A type is denoted by a type name.

The language predeclares certain type names.

```
Type      = TypeName .
TypeName  = identifier .
```

Boolean types

A boolean type represents the set of Boolean truth values denoted by the predeclared constants `true` and `false`. The predeclared boolean type is `bool`.

Numeric types

An *integer* type represents the set of integer values. It is the only numeric type present in GoLF.

There is a predeclared integer type:

```
int    the set of all signed integers
```

In GoLF, integers are 32 bits in size (-2147483648 to 2147483647) and are represented using two's complement arithmetic.

String types

A string type represents the set of string values. A string value is a (possibly empty) sequence of bytes. Strings are immutable: once created, it is impossible to change the contents of a string, although a string variable may have a different (immutable) string assigned to it. The predeclared string type is `string`.

Blocks

A *block* is a possibly empty sequence of declarations and statements within matching brace brackets.

```
Block = "{" StatementList "}" .
StatementList = { Statement ";" } .
```

In addition to explicit blocks in the source code, there are implicit blocks:

1. The *universe block* encompasses all GoLF source text.
2. The source input file has a file block containing all GoLF source text in that file.

Blocks nest and influence scoping.

Declarations and scope

A *declaration* binds a non-blank identifier to a variable or function. Every identifier in a program must be declared. No identifier may be declared twice in the same block.

```
Declaration   = VarDecl .
TopLevelDecl  = Declaration | FunctionDecl .
```

The *scope* of a declared identifier is the extent of source text in which the identifier denotes the specified constant, type, variable, or function.

GoLF is lexically scoped using blocks:

1. The scope of a predeclared identifier is the universe block.
2. The scope of an identifier denoting a variable or function declared at top level (outside any function) is the file block.
3. The scope of an identifier denoting a function parameter is the function body.
4. The scope of a variable identifier declared inside a function begins at the end of the VarSpec and ends at the end of the innermost containing block.

An identifier declared in a block may be redeclared in an inner block. While the identifier of the inner declaration is in scope, it denotes the entity declared by the inner declaration.

Predeclared identifiers

The following identifiers are implicitly declared in the universe block:

```
Types:
    bool int string

Constants:
    true false

Functions:
    getchar halt len printb printc printi prints
```

Variable declarations

A variable declaration creates a variable, binds an identifier to it, and gives it a type and an initial (zero) value.

```
VarDecl     = "var" VarSpec .
VarSpec     = identifier Type .
```

Each variable is initialized to its zero value.

Function declarations

A function declaration binds an identifier, the *function name*, to a function.

```
FunctionDecl = "func" FunctionName Signature FunctionBody .
FunctionName = identifier .
FunctionBody = Block .

Signature      = Parameters [ Result ] .
Result         = Type .
Parameters     = "(" [ ParameterList [ "," ] ] ")" .
ParameterList  = ParameterDecl { "," ParameterDecl } .
ParameterDecl  = identifier Type .
```

If the function's signature declares a result parameter, the last statement *executed* in the function body must be a `return` statement. It is a compile-time error for a function with a result parameter to have no `return` statements in the function body.

Functions may be recursive.

Expressions

An expression specifies the computation of a value by applying operators and functions to operands.

Operands

Operands denote the elementary values in an expression. An operand may be a literal, an identifier denoting a constant, variable, or function, or a parenthesized expression.

```
Operand     = Literal | OperandName | "(" Expression ")" .
Literal     = BasicLit .
BasicLit    = int_lit | string_lit .
OperandName = identifier .
```

Primary expressions

Primary expressions are the operands for unary and binary expressions.

```
PrimaryExpr = Operand | PrimaryExpr Arguments .

Arguments      = "(" [ ExpressionList [ "," ] ] ")" .
ExpressionList = Expression { "," Expression } .
```

Examples:

```
x
2
f(314159, true)
foo()
```

Calls

Given an expression f with function type,

```
f(a1, a2, … an)
```

calls f with arguments `a1`, `a2`, … `an`. Each argument must be an expression whose type matches the type of the corresponding parameter of `f`. All arguments are evaluated before the function is called. The type of the expression is the result type of `f`.

In a function call, the function value and arguments are evaluated in the usual lexical left-to-right order. After they are evaluated, the parameters of the call are passed by value to the function and the called function begins execution. The return parameters of the function are passed by value back to the caller when the function returns.

Operators

Operators combine operands into expressions.

```
Expression = UnaryExpr | Expression binary_op Expression .
UnaryExpr  = PrimaryExpr | unary_op UnaryExpr .

binary_op  = "||" | "&&" | rel_op | add_op | mul_op .
rel_op     = "==" | "!=" | "<" | "<=" | ">" | ">=" .
add_op     = "+" | "-" .
mul_op     = "*" | "/" | "%" .

unary_op   = "-" | "!" .
```

Comparisons are discussed elsewhere. For binary operators, the operand types must be identical.

Operator precedence

Unary operators have the highest precedence. There are five precedence levels for binary operators. Multiplication operators bind strongest, followed by addition operators, comparison operators, `&&` (logical AND), and finally `||` (logical OR):

```
Precedence    Operator
    5             *  /  %
    4             +  -
    3             ==  !=  <  <=  >  >=
    2             &&
    1             ||
```

Binary operators of the same precedence associate from left to right. For instance, `x / y * z` is the same as `(x / y) * z`.

Arithmetic operators

Arithmetic operators apply to numeric values and yield a result of the same type as the first operand. The arithmetic operators (`+`, `-`, `*`, `/`, `%`) apply to integer types only.

Integer operators

For two integer values `x` and `y`, the integer quotient `q = x / y` and remainder `r = x % y` satisfy the following relationships:

```
x = q*y + r  and  |r| < |y|
```

with x / y truncated towards zero ("truncated division").

```
 x     y     x / y     x % y
 5     3       1         2
-5     3      -1        -2
 5    -3      -1         2
-5    -3       1        -2
```

The one exception to this rule is that if the dividend `x` is the most negative value for the int type of `x`, the quotient `q = x / -1` is equal to `x` (and `r = 0`) due to two's-complement integer overflow.

If the divisor is zero at run time, a run-time error occurs.

Integer overflow

For signed integers, the operations `+`, `-`, `*`, `/`, may legally overflow and the resulting value exists and is deterministically defined by the signed integer representation, the operation, and its operands. Overflow does not cause a run-time error. A compiler may not optimize code under the assumption that overflow does not occur. For instance, it may not assume that `x < x + 1` is always true.

Comparison operators

Comparison operators compare two operands and yield a boolean value.

```
==    equal
!=    not equal
<     less
<=    less or equal
>     greater
>=    greater or equal
```

The equality operators `==` and `!=` apply to operands that are *comparable*. The ordering operators `<`, `<=`, `>`, and `>=` apply to operands that are *ordered*. These terms and the result of the comparisons are defined as follows:

- Boolean values are comparable. Two boolean values are equal if they are either both `true` or both `false`.
- Integer values are comparable and ordered, in the usual way.
- String values are comparable and ordered, lexically byte-wise.

Logical operators

Logical operators apply to boolean values and yield a result of the same type as the operands. The right operand is evaluated conditionally, i.e., they are short-circuiting.

```
&&    conditional AND    p && q  is  "if p then q else false"
||    conditional OR     p || q  is  "if p then true else q"
!     NOT                !p      is  "not p"
```

Statements

Statements control execution.

```
Statement =
    Declaration | SimpleStmt |
    ReturnStmt | BreakStmt |
    Block | IfStmt | ForStmt .

SimpleStmt = EmptyStmt | ExpressionStmt | Assignment .
```

Empty statements

The empty statement does nothing.

```
EmptyStmt = .
```

Expression statements

Function calls can appear in statement context. Such statements may be parenthesized.

```
ExpressionStmt = Expression .
```

Assignments

```
Assignment = Expression assign_op Expression .
```

```
assign_op = "=" .
```

The left-hand side operand must be the name of a variable; other types of names, like those of functions and constants, may not be assigned to. Operands may be parenthesized. The type of the value on the right-hand side must be identical to the variable's type on the left-hand side.

If statements

"If" statements specify the conditional execution of two branches according to the value of a boolean expression. If the expression evaluates to true, the "if" branch is executed, otherwise, if present, the "else" branch is executed.

```
IfStmt = "if" Expression Block [ "else" ( IfStmt | Block ) ] .
```

The type of an `if` expression must be boolean.

For statements

A "for" statement specifies repeated execution of a block. There are two forms: with and without a condition.

```
ForStmt = "for" [ Condition ] Block .
Condition = Expression .
```

For statements with a single condition

In its simplest form, a "for" statement specifies the repeated execution of a block as long as a boolean condition evaluates to true. The condition is evaluated before each iteration.

```
for a < b {
        a = a * 2
}
```

The type of a `for` condition must be boolean.

For statements with no condition

If the condition is absent, it is equivalent to the boolean value `true`.

Return statements

A "return" statement in a function F terminates the execution of F, and optionally provides a result value.

```
ReturnStmt = "return" [ Expression ] .
```

In a function without a result type, a "return" statement must not specify a result value.

```
func noResult() {
    return
}
```

For a function with a result type, the type of the return expression must be the same as the function's result type.

Break statements

A "break" statement terminates execution of the innermost "for" statement within the same function.

```
BreakStmt = "break" .
```

It is an error for a `break` to occur without an enclosing `for` loop.

Built-in functions

Built-in functions are predeclared. They are called like any other function.

Although a GoLF user never sees declarations for these routines, some are shown in the table below to illustrate how they should be called:

```
Name      Signature               Description

getchar   func getchar() int      Returns an ASCII input character as an integer, or -1 if end of file has been reached.
halt      func halt()             Halts execution of the program.
len       func len(s string) int  Returns the string length in bytes.
printb    func printb(b bool)     Prints a boolean as “true” or “false”.
printc    func printc(c int)      Prints an integer as an ASCII character. The output of values outside the range of 7-bit ASCII is undefined.
printi    func printi(i int)      Prints an integer.
prints    func prints(s string)   Prints a string.
```

None of the printing functions automatically output a newline.

Input source file and program execution

A GoLF program consists of a single file declaring variables and functions belonging to the program.

```
SourceFile       = { TopLevelDecl ";" } .
```

A complete program declares a function `main` that takes no arguments and returns no value. It is an error for `main` to be absent from an input source file.

```
func main() { … }
```

Program execution begins by invoking the function `main`. When that function invocation returns, the program exits.

Appendix: Collected grammar rules

Lexical tokens:

```
newline        = /* the character U+000A */ .
ascii_char     = /* an arbitrary ASCII character except newline */ .
ascii_letter   = "A" … "Z" | "a" … "z" .
ascii_digit    = "0" … "9" .

letter        = ascii_letter | "_" .
decimal_digit = "0" … "9" .

identifier = letter { letter | ascii_digit } .

int_lit        = decimal_lit .
decimal_lit    = decimal_digits .
decimal_digits = decimal_digit { decimal_digit } .

string_lit             = interpreted_string_lit .
interpreted_string_lit = `"` { ascii_value } `"` .

ascii_value    = ascii_char | escaped_char .
escaped_char   = `\` ( "b" | "f" | "n" | "r" | "t" | `\` | `"` ) .

assign_op = "=" .

binary_op  = "||" | "&&" | rel_op | add_op | mul_op .
rel_op     = "==" | "!=" | "<" | "<=" | ">" | ">=" .
add_op     = "+" | "-" .
mul_op     = "*" | "/" | "%" .
unary_op   = "-" | "!" .
```

Syntax rules (SourceFile is the start symbol):

```
Type      = TypeName .
TypeName  = identifier .

Block = "{" StatementList "}" .

StatementList = { Statement ";" } .
Declaration   = VarDecl .
TopLevelDecl  = Declaration | FunctionDecl .

VarDecl     = "var" VarSpec .
VarSpec     = identifier Type .

FunctionDecl = "func" FunctionName Signature FunctionBody .
FunctionName = identifier .
FunctionBody = Block .

Signature      = Parameters [ Result ] .
Result         = Type .
Parameters     = "(" [ ParameterList [ "," ] ] ")" .
ParameterList  = ParameterDecl { "," ParameterDecl } .
ParameterDecl  = identifier Type .

Operand     = Literal | OperandName | "(" Expression ")" .
Literal     = BasicLit .
BasicLit    = int_lit | string_lit .
OperandName = identifier .
PrimaryExpr = Operand | PrimaryExpr Arguments .

Arguments      = "(" [ ExpressionList [ "," ] ] ")" .
ExpressionList = Expression { "," Expression } .

Expression = UnaryExpr | Expression binary_op Expression .
UnaryExpr  = PrimaryExpr | unary_op UnaryExpr .
```

```
Statement =
    Declaration | SimpleStmt |
    ReturnStmt | BreakStmt |
    Block | IfStmt | ForStmt .

SimpleStmt = EmptyStmt | ExpressionStmt | Assignment .

EmptyStmt = .

ExpressionStmt = Expression .

Assignment = Expression assign_op Expression .

IfStmt = "if" Expression Block [ "else" ( IfStmt | Block ) ] .

ForStmt = "for" [ Condition ] Block .

Condition = Expression .

ReturnStmt = "return" [ Expression ] .

BreakStmt = "break" .

SourceFile       = { TopLevelDecl ";" } .
```

John Aycock, 15 March 2023