

# COMP520 - GoLite Syntax Specification

Vincent Foley

February 6, 2015

## 1 Scanner

### 1.1 Source code representation

Go programs are UTF-8 encoded ([http://golang.org/ref/spec#Source\\_code\\_representation](http://golang.org/ref/spec#Source_code_representation)), and Go allows Unicode characters to be used in identifiers, string literals, etc. In GoLite, we'll only ask that you support the ASCII charset.

### 1.2 Keywords

Your scanner must recognize all the Go keywords:

<code>break</code>	<code>default</code>	<code>func</code>	<code>interface</code>	<code>select</code>
<code>case</code>	<code>defer</code>	<code>go</code>	<code>map</code>	<code>struct</code>
<code>chan</code>	<code>else</code>	<code>goto</code>	<code>package</code>	<code>switch</code>
<code>const</code>	<code>fallthrough</code>	<code>if</code>	<code>range</code>	<code>type</code>
<code>continue</code>	<code>for</code>	<code>import</code>	<code>return</code>	<code>var</code>

Although we won't implement the semantics associated with all the keywords (e.g. `interface` and `select`), we will make them reserved to prevent users from using them as identifiers and thus having invalid Go code.

In addition to the Go keywords, we will add a few of our own. These GoLite keywords will make the implementation of some parser rules easier.

<code>int</code>	<code>float64</code>	<code>bool</code>	<code>rune</code>	<code>string</code>
<code>print</code>	<code>println</code>	<code>append</code>		

- <http://golang.org/ref/spec#Keywords>

### 1.3 Operators

Your scanner must recognize all the Go operators:

+	&	+=	&=	&&	==	!=	( )
-		-=	=		<	<=	[ ]
*	^	*=	^=	<-	>	>=	{ }
/	<<	/=	<<=	++	=	:=	, ;
%	>>	%=	>>=	-	!	...	. :
	&^		&^=				

- [http://golang.org/ref/spec#Operators\\_and\\_Delimiters](http://golang.org/ref/spec#Operators_and_Delimiters)

## 1.4 Comments

Your scanner must recognize both block comment and line comments. Note that block comments do not nest. It is an error if a block comment is opened, and never closed (i.e. reaching the end of file).

```
// This is a line comment

/* This is a
   block comment */

/* Block comments do not nest
   /* The star-slash on the right ends the while comment */
   This is an error */
```

- <http://golang.org/ref/spec#Comments>

## 1.5 Literals

Your scanner must recognize literals for integers in decimal, octal and hexadecimal.

Your scanner must recognize literals for floating-point numbers. Note that, unlike in Minilang, the integral part or the decimal part of the number can be left off. Your scanner does not need to support scientific notation.

Your scanner must recognize literals for runes (chars). You should support the single-character escape sequences listed below. You do not need to support escape sequences of character codes (e.g. '\xa8').

```
\a  \b  \f  \n  \r
\t  \v  \\  \'
```

Your scanner must recognize literals for interpreted strings and raw strings. Your string literals should support the same escape sequences as runes. In addition, you should be able to escape a double quote in an interpreted string by using \". In a raw string, all characters stand for themselves: there are no escape sequences and there is no way to include a back-quote. It is an error if a string is opened, and never closed (i.e. reaching the end of file).

```
/* Integer literals */
255    // decimal
0377   // octal
```

```

0xff    // hexadecimal

/* Floating-point literals */
0.12    // Integral and decimal parts
.12     // Decimal part only
12.     // Integral part only

/* Rune literals */
'L'     // Single character
'\n'    // Escaped character

/* String literals */
"hello\n" // Interpreted string, \n is transformed into newline
'hello\n' // Raw string, \n appears as a '\n' followed by 'n'

• http://golang.org/ref/spec#Integer\_literals
• http://golang.org/ref/spec#Floating-point\_literals
• http://golang.org/ref/spec#Rune\_literals
• http://golang.org/ref/spec#escaped\_char
• http://golang.org/ref/spec#String\_literals

```

## 1.6 Identifiers

Your scanner must recognize identifiers. Identifiers in GoLite only use the ASCII charset.

- <http://golang.org/ref/spec#Identifiers>

## 1.7 Semicolons

Your scanner should insert semicolon tokens in the token streams according to semicolon insertion rule 1. Rule 1 covers more than 95% of all semicolon insertions in Go, and 100% of cases in GoLite.

Because we have added several keywords that are identifiers in Go (i.e. the base types), you'll need to extend the list of tokens that trigger the insertion of a semicolon token.

You do not have to implement semicolon insertion rule 2.

- <http://golang.org/ref/spec#Semicolons>

## 2 Parser

### 2.1 Program structure

In Go, a program is divided up in three parts: a package declaration, a list of import statements, and a list of top-level declarations. GoLite will not support packages, so we won't implement the import statements. We will however keep the package declaration; this will allow GoLite programs to be used with a Go compiler.

- [http://golang.org/ref/spec#Source\\_file\\_organization](http://golang.org/ref/spec#Source_file_organization)

### 2.2 Package declaration

A package declaration is the keywords `package` followed by an identifier. There must be exactly one in a GoLite program.

- <http://golang.org/ref/spec#PackageClause>

### 2.3 Top-level declarations

Your parser must support the following top-level declarations:

- variable declarations
- type declarations
- function declarations

You do not have to support constant declarations nor method declarations.

- <http://golang.org/ref/spec#TopLevelDecl>

### 2.4 Variable declaration

Go has three forms of variable specifications:

- Specifying the type, but leaving out an expression
- Specifying an expression, but leaving out the type
- Specifying both the type and an expression

```
var x int           // type only
var y = 42          // expression only
var z int = 1       // type and expression
```

In the above example, we declared only one identifier. A variable declaration can also declare multiple identifiers at once.

```
var x1, x2 int
varnn y1, y2 = 42, 43
var z1, z2 int = 1, 2
```

Finally, it is possible to "distribute" the `var` keyword to a list of variable specifications:

```
var (
    x1, x2 int
    y1, y2 = 42, 43
    z1, z2 int = 1, 2
)
```

Your parser must support all both forms of variable declaration and all three forms of variable specification.

- <http://golang.org/ref/spec#VarDecl>

## 2.5 Type declaration

A type declaration in Go is the keyword `type` followed by an identifier (the new name for a type) followed by an existing type.

```
type num int          // simple type alias
type point struct {   // point is a struct
    x, y float64
}
```

Like variable declarations, it is possible to "distribute" the `type` keyword to a list of type specifications.

```
type (
    num int
    point struct {
        x, y float64
    }
)
```

Your parser must support both forms of type declarations.

- <http://golang.org/ref/spec#TypeDecl>

## 2.6 Function declaration

Go supports several ways to declare functions (see the link below for more information). To simplify your parser, in GoLite we will support only two forms of function definitions. The following table describes the major differences between Go and GoLite.

	Go	GoLite
Arbitrary number of return values	YES	NO
Named return values	YES	NO
Optional body	YES	NO
Unnamed input parameters	YES	NO
Optional comma after the last parameter	YES	NO
Variadic input parameters	YES	NO

In GoLite, we have two ways of specifying input parameters that we'll call long form and short form. In the long form, the input parameters are a list of id+type pair. In the short form, multiple consecutive parameters of the same type can be accumulated in a list.

```
// Long form, one id with one type
func f(a int, b int, c string, d int) {
    return
}

// Short form, many ids with one type
func f(a, b int, c string, d int) string {
    return c
}
```

- [http://golang.org/ref/spec#Function\\_declarations](http://golang.org/ref/spec#Function_declarations)

## 2.7 Types

### 2.7.1 Basic types

Go supports several basic types. To keep our compiler simple, GoLite supports only five basic types: int, float64, bool, rune, string.

<http://golang.org/ref/spec#Types>

### 2.7.2 Slices

Your parser must support slices. You do not need to support slice literals.

```
// A slice literal, not supported in GoLite
var x []int = []int{ 1, 2, 3 }

// In GoLite, we'll use append()
var x []int
x = append(x, 1)
x = append(x, 2)
x = append(x, 3)
```

[http://golang.org/ref/spec#Slice\\_types](http://golang.org/ref/spec#Slice_types)

### 2.7.3 Arrays

Your parser must support arrays. In Go, the size can be an expression, in GoLite the size must be an integer literal. You do not need to support array literals.

```
// An array literal, not supported in GoLite
var x [3]int = [3]int{ 1, 2, 3 }
```

```
// In GoLite, we'll use indexing
var x [3]int
x[0] = 1
x[1] = 2
x[2] = 3
```

[http://golang.org/ref/spec#Array\\_types](http://golang.org/ref/spec#Array_types)

### 2.7.4 Structs

Your parser must support structs. You do not need to support anonymous members. You do not need to support tags. You do not need to support struct literals.

```
type point struct {
    x, y, z int
}
```

```
// A struct literal, not supported in GoLite
var p point = point{ 1, 2, 3 }
```

```
// In GoLite, we'll use field access
var p point
p.x = 1
p.y = 2
p.z = 3
```

[http://golang.org/ref/spec#Struct\\_types](http://golang.org/ref/spec#Struct_types)

## 2.8 Statements

### 2.8.1 Empty statements

Your parser must support empty statements.

[http://golang.org/ref/spec#Empty\\_statements](http://golang.org/ref/spec#Empty_statements)

### 2.8.2 Expression statements

Your parser must support expression statements. (See next section to know which expressions are supported.)

[http://golang.org/ref/spec#Expression\\_statements](http://golang.org/ref/spec#Expression_statements)

### 2.8.3 Assignment statements

In Go and GoLite, assignment is not an expression, but a statement. Your parser must support assignment statements.

As described in the specification, if the assignment operator is the equal sign (=), the left-hand side and right-hand side can be lists. Because we will not support multiple return values, in GoLite the number of identifiers must match the number of expressions.

For op-assign statements, only a single id and single expression can be on either side of the assignment operator.

Hint: the reference says that the left-hand side is an expression list, however you should make sure that the expression on the left is a lvalue (i.e. an identifier, an array indexing expression, etc.)

<http://golang.org/ref/spec#Assignments>

### 2.8.4 Declaration statements

Variable declarations and type declarations are statements and can occur in a block. Function declarations are **not** statements and cannot occur inside a block. Your parser must support declaration statements.

<http://golang.org/ref/spec#Declaration>

### 2.8.5 Short declaration statements

Your parser must support short declaration statements.

<http://golang.org/ref/spec#ShortVarDecl>

### 2.8.6 Increment and decrement statements

Your parser must support the post-increment and post-decrement statements. In Go, only the postfix form is allowed, and an increment/decrement is a statement, not an expression like C.

Hint 1: you can reuse the += and -= nodes of your AST to support these statements. This will reduce the number of cases you need to handle when traversing the AST.

Hint 2: like with assignment, you should make sure that the left-hand side is a lvalue.

<http://golang.org/ref/spec#IncDecStmt>



### 2.8.7 Print and println statements

These two statements are specific to GoLite, and allow the programmer to display a list of expressions.

```
print_stmt    ::= 'print' ' (' expr_list ')'
println_stmt  ::= 'println' ' (' expr_list ')'
expr_list     ::= /* empty */
                | expr_list ',' expr
```

`print` should print all its argument on a single line, with no space separating the elements. `println` prints its arguments separated by one space, and also prints a newline character after the last expression.

### 2.8.8 Return statements

Your parser must support return statements. In GoLite, `return` only accepts zero or one expression.

[http://golang.org/ref/spec#Return\\_statements](http://golang.org/ref/spec#Return_statements)

### 2.8.9 If statements

Your parser must support if statements. You should be able to parse an if statement with no else part, an if/else statement and if/else if/else statements. You should also be able to parse the optional initialization statement that can come before the condition expression.

```
// single if
if expr {
    // statements
}

// if/else
if expr {
    // statements
} else {
    // statements
}

// if/else if/else
// note that no brace comes between the else and the if
if expr1 {
    // statements
} else if expr2 {
    // statements
} else {
    // statements
}
```

[http://golang.org/ref/spec#If\\_statements](http://golang.org/ref/spec#If_statements)

### 2.8.10 Switch statements

Your parser must support expression switch statements.

Hint: you'll need a weeding phase to make sure that there is only one default case.

GoLite does not support type switch statements.

<http://golang.org/ref/spec#ExprSwitchStmt>

### 2.8.11 For statements

Your parser must support the for statement, but not under all its forms. In GoLite, we will support the infinite loop for, the while loop and the three-part loop.

```
// Infinite loop
for {
}

// "While" loop
for x < 10 {
}

// Three-part loop
for i := 0; i < 10; i++ {
}
```

GoLite does not support the *for/range* statements.

[http://golang.org/ref/spec#For\\_statements](http://golang.org/ref/spec#For_statements)

### 2.8.12 Break and continue statements

Your parser must support the break and the continue statements. In GoLite, we will only support the unlabelled form of these statements.

Hint: `break` and `continue` must only appear within a loop, however this is not something that can be expressed in a context-free grammar. You will need a weeding pass to make sure all breaks and continues are properly used.

- [http://golang.org/ref/spec#Break\\_statements](http://golang.org/ref/spec#Break_statements)
- [http://golang.org/ref/spec#Continue\\_statements](http://golang.org/ref/spec#Continue_statements)

## 2.9 Expressions

### 2.9.1 Parenthesized expressions

Your parser must support parenthesized expressions.

<http://golang.org/ref/spec#Operands>

### 2.9.2 Identifiers

Your parser must support identifiers as expressions.

- <http://golang.org/ref/spec#Operands>
- <http://golang.org/ref/spec#identifier>

### 2.9.3 Literals

Your parser must support literals (integers, floating-point, runes and strings) as expressions.

<http://golang.org/ref/spec#BasicLit>

### 2.9.4 Unary operations

Your parser must support unary operations as expressions. All unary operators have the same operator precedence.

GoLite supports the following four unary operators:

+ - ! ^

<http://golang.org/ref/spec#Operators>

### 2.9.5 Binary operations

Your parser must support binary operations as expressions. Your parser must also support the correct precedence and associativity of the operators.

GoLite supports all the Go binary operators.

- <http://golang.org/ref/spec#Operators>
- [http://golang.org/ref/spec#Operator\\_precedence](http://golang.org/ref/spec#Operator_precedence)

### 2.9.6 Function calls

Your parser must support function calls as expressions. The syntax of function calls in GoLite is simpler than the one in Go; in particular, we will not allow a trailing comma after the last, and we cannot use the ellipses to pass variadic parameters (since we don't support those functions).

<http://golang.org/ref/spec#Calls>

### 2.9.7 Append

In Go, `append` is a builtin because its semantics cannot be implemented under the Go type system. In GoLite, we'll implement a simpler version of `append`.

```
append_expr ::= 'append' '(' ID ',' expr ')'
```

The first argument to `append` is the name of a slice to append to, and the second expression is the element to add. In GoLite, we don't allow the first argument to be a subslice.

[http://golang.org/ref/spec#Appending\\_and\\_copying\\_slices](http://golang.org/ref/spec#Appending_and_copying_slices)

### 2.9.8 Type casts

In GoLite, we'll support a very primitive form of type casting: only base types (except strings) and aliases of base types (except aliases of strings) can be used for type casting.

The syntax of type casting looks like the syntax for function calls.

```
type num int
var x int = 3
var y num = num(x)
var z float64 = float64(x)
```

Hint: casts to base types (e.g. `float64(x)`) can be syntactically distinguished because they use a reserved keyword and have their own AST node. However, aliases (e.g. `num(x)`) will likely be represented as function calls in the AST after parsing. We can fix the AST (i.e. switch from a function call to a type cast node) in a later phase of the compiler.