COMP520 - GoLite Syntax Specification

Vincent Foley

February 10, 2019

1 Scanner

1.1 Source code representation

Go programs are UTF-8 encoded (https://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:

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

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.

`print` `println` `append` `len` `cap`

https://golang.org/ref/spec#Keywords

1.3 Operators

Your scanner must recognize all the Go operators:

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

https://golang.org/ref/spec#Operators_and_Delimiters

1.4 Comments

Your scanner must recognize both block comment and line comments. 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 above comment */
   This is an error */
```

https://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. 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 must 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. Interpreted string literals must support the same escape sequences as runes. In addition, you must 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 '\' followed by 'n'
```

- https://golang.org/ref/spec#Integer_literals
- https://golang.org/ref/spec#Floating-point_literals
- https://golang.org/ref/spec#Rune_literals
- https://golang.org/ref/spec#escaped_char
- https://golang.org/ref/spec#String_literals

1.6 Identifiers

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

Hint: You should have a weeding pass to check the correct use of the blank identifier.

- https://golang.org/ref/spec#Identifiers
- https://golang.org/ref/spec#Blank_identifier

1.7 Semicolons

Your scanner must 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.

You do not have to implement semicolon insertion rule 2.

https://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 programs to be used with a Go compiler.

https://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.

https://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.

https://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 examples above, we declared only one identifier. A variable declaration can also declare multiple identifiers are once.

```
var x1, x2 int
var 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 both forms of variable declaration (individual and distributed) and all three forms of variable specification.

https://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 definition
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.

Note that in GoLite we do not support type aliasing.

https://golang.org/ref/spec#TypeDecl

2.6 Function declaration

Go supports several ways to declare functions (see the link below for more information). To simpify 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 parameter name is accompanied with a type. In the short form, we can list multiple consecutive parameter names followed by the type that all these identifiers should have.

```
// 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
}
```

https://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`.

https://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)
```

https://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
```

https://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
```

https://golang.org/ref/spec#Struct_types

2.8 Statements

2.8.1 Empty statements

Your parser must support empty statements.

https://golang.org/ref/spec#Empty_statements

2.8.2 Block statements

You parser must support block statements.

```
block_stmt   ::= '{' stmt_list '}'
```

2.8.3 Expression statements

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

https://golang.org/ref/spec#Expression_statements

2.8.4 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 do 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.

(Note: the reference says that the left-hand size is an expression list. For this milestone you may ignore lvalue checks.)

https://golang.org/ref/spec#Assignments

2.8.5 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.

https://golang.org/ref/spec#Declaration

2.8.6 Short declaration statements

Your parser must support short declaration statements.

https://golang.org/ref/spec#ShortVarDecl

2.8.7 Increment and decrement statements

Your parser must support the post-increment and post-decrement statements. In Go, only the postfix form is allowed. Beware that, unlike in C, increment and decrement are statements, not expressions.

https://golang.org/ref/spec#IncDecStmt

2.8.8 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
               | 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.

2.8.9 Return statements

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

https://golang.org/ref/spec#Return_statements

2.8.10 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
}
```

https://golang.org/ref/spec#If_statements

2.8.11 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.

https://golang.org/ref/spec#ExprSwitchStmt

2.8.12 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.

https://golang.org/ref/spec#For_statements

2.8.13 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 withing a loop (or `break` also within a `switch` case), however this is not something that can be expressed (easily) in a context-free grammar. You will need a weeding pass to make sure all breaks and continues are properly used.

- https://golang.org/ref/spec#Break_statements
- https://golang.org/ref/spec#Continue_statements

2.9 Expressions

2.9.1 Parenthesized expressions

Your parser must support parenthesized expressions.

https://golang.org/ref/spec#Operands

2.9.2 Identifiers

Your parser must support identifiers as expressions.

- https://golang.org/ref/spec#Operands

- https://golang.org/ref/spec#identifier

2.9.3 Literals

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

https://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:

```
+   -   !   ^
```

https://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.

- https://golang.org/ref/spec#Operators
- https://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 argument, and we cannot use the ellipses to pass variadic paramaters (since we don't support those functions).

https://golang.org/ref/spec#Calls

2.9.7 Builtins

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' '(' expr ',' expr ')'
```

The first argument to `append` is a slice expression to append to, and the second expression is the element to add. In GoLite, we don't support appending multiple elements in the same expression.

To help you write interesting programs, we will also have builtin functions `len` and `cap` which each take a single slice or array argument. `len` also supports string types.

```
len_expr ::= 'len' '(' expr ')'
cap_expr ::= 'cap' '(' expr ')'
```

- https://golang.org/ref/spec#Appending_and_copying_slices
- https://golang.org/ref/spec#Length_and_capacity

2.9.8 Type casts

In GoLite, we'll support a very primitive form of type casting: only base types and types whose underlying type is a base type 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 will (likely) appear as function calls in your AST. It will be important in a later phase of the compiler to convert them to the proper cast nodes. The full specification for conversions are described in the link below, but beware of functionality we do not support.

https://golang.org/ref/spec#Conversions