

# CD LAB PROJECT SYNOPSIS

## *Compiler Design for Golang Language*

Team members :-

Nishchay Parashar - Reg. no. 160905298 | Roll no. 44

Manas Gupta - Reg. no. 160905406 | Roll no. 62

Ankur Sangwan - Reg. no. 160905412 | Roll no. 63

Go (often referred to as Golang) is a statically typed, compiled programming language designed at Google by Robert Griesemer, Rob Pike, and Ken Thompson. Go is syntactically similar to C, but with the added benefits.

Our project is a partial implementation of the Golang compiler covering the fundamental grammar of the language.

## The Go Programming Language Specification

### Characters

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

`newline` = /\* the Unicode code point U+000A \*/ .

`unicode_char` = /\* an arbitrary Unicode code point except newline \*/ .

`unicode_letter` = /\* a Unicode code point classified as "Letter" \*/ .

`unicode_digit` = /\* a Unicode code point classified as "Number, decimal digit" \*/ .

### Letters and digits

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

`letter` = `unicode_letter` | `"_"` .

`decimal_digit` = `"0"` ... `"9"` .

`octal_digit` = `"0"` ... `"7"` .

`hex_digit` = `"0"` ... `"9"` | `"A"` ... `"F"` | `"a"` ... `"f"` .

### Comments

Comments serve as program documentation. There are two forms:

1. *Line comments* start with the character sequence `//` and stop at the end of the line.

2. *General comments* start with the character sequence `/*` and stop with the first subsequent character sequence `*/`.

A comment cannot start inside a [string literal](#), or inside a comment. A general comment containing no newlines acts like a space. Any other comment acts like a newline.

## Tokens

Tokens form the vocabulary of the Go 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.

## Identifiers

decimal\_digit

identifier = letter { letter | unicode\_digit } .

## Keywords

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

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

## 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](#). An optional prefix sets a non-decimal base: 0 for octal, 0x or 0X for hexadecimal. In hexadecimal literals, letters a-f and A-F represent values 10 through 15.

```
int_lit = decimal_lit .  
decimal_lit = ( "1" ... "9" ) { decimal_digit } .
```

## Floating-point literals

A floating-point literal is a decimal representation of a [floating-point constant](#). It has an integer part, a decimal point, a fractional part, and an exponent part. The integer and fractional part comprise decimal digits; the exponent part is an e or E followed by an optionally signed decimal exponent. One of the integer part or the fractional part may be elided; one of the decimal point or the exponent may be elided.

```
float_lit = decimals "." [ decimals ] [ exponent ] |  
           decimals exponent |  
           "." decimals [ exponent ] .  
decimals = decimal_digit { decimal_digit } .  
exponent = ( "e" | "E" ) [ "+" | "-" ] decimals .
```

## Imaginary literals

An imaginary literal is a decimal representation of the imaginary part of a [complex constant](#). It consists of a [floating-point literal](#) or decimal integer followed by the lower-case letter i.

```
imaginary_lit = ( decimals | float_lit ) "i" .
```

## String literals

A string literal represents a [string constant](#) obtained from concatenating a sequence of characters. There are two forms: raw string literals and interpreted string literals.

```
string_lit = raw_string_lit | interpreted_string_lit .  
raw_string_lit = "" { unicode_char | newline } "" .  
interpreted_string_lit = "`" { unicode_value } "`" .
```

## Blocks

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

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

## Types

A type determines a set of values together with operations and methods specific to those values. A type may be denoted by a *type name*, if it has one, or specified using a *type literal*, which composes a type from existing types.

```
Type = TypeName | TypeLit | "(" Type ")" .  
TypeName = identifier .
```

```

TypeLit  = ArrayType | PointerType | FunctionType
ArrayType = "[" ArrayLength "]" ElementType .
ArrayLength = Expression .
ElementType = Type .
PointerType = "*" BaseType .
BaseType  = Type .
FunctionType = "func" Signature .
Signature   = Parameters [ Result ] .
Result      = Parameters | Type .
Parameters  = "(" [ ParameterList [ "," ] ] ")" .    Removed ", "
ParameterList = ParameterDecl { "," ParameterDecl } .
ParameterDecl = [ IdentifierList ] [ "..." ] Type .

```

## Variable declarations

A variable declaration creates one or more [variables](#), binds corresponding identifiers to them, and gives each a type and an initial value.

```

VarDecl  = "var" ( VarSpec | "(" { VarSpec ";" } ")" ) .
VarSpec  = IdentifierList ( Type [ "=" ExpressionList ] | "=" ExpressionList ) .

```

## Constants

There are *boolean constants*, *rune constants*, *integer constants*, *floating-point constants*, *complex constants*, and *string constants*. Integer, floating-point, and complex constants are collectively called *numeric constants*.

```

ConstDecl  = "const" ( ConstSpec | "(" { ConstSpec ";" } ")" ) .
ConstSpec  = IdentifierList [ [ Type ] "=" ExpressionList ] .
IdentifierList = identifier { "," identifier } .
ExpressionList = Expression { "," Expression } .

```

## Declarations and Scope

```

Declaration = ConstDecl | VarDecl .
TopLevelDecl = Declaration | FunctionDecl .

```

## Short variable declarations

A *short variable declaration* uses the syntax:

```

ShortVarDecl = IdentifierList "!=" ExpressionList .

```

## Function declarations

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

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

If the function's [signature](#) declares result parameters, the function body's statement list must end in a [terminating statement](#).

## 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 = "+" | "-" | "!" | "^" | "*" | "&" | "<-" .
```

## Operands

Operands denote the elementary values in an expression. An operand may be a literal, a (possibly [qualified](#)) non-[blank](#) identifier denoting a [constant](#), [variable](#), or [function](#), or a parenthesized expression.

The [blank identifier](#) may appear as an operand only on the left-hand side of an [assignment](#).

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

## Statements

Statements control execution.

```
Statement =  
    Declaration | LabeledStmt | SimpleStmt |  
    GoStmt | ReturnStmt | BreakStmt | ContinueStmt | GotoStmt |  
    FallthroughStmt | Block | IfStmt | SwitchStmt | SelectStmt | ForStmt |  
    DeferStmt .
```

```
SimpleStmt = EmptyStmt | ExpressionStmt | IncDecStmt | Assignment | ShortVarDecl .
```

```
Declaration = ConstDecl | VarDecl .  
LabeledStmt = Label ":" Statement .  
Label = identifier .
```

```

GoStmt = "go" Expression .
ReturnStmt = "return" [ ExpressionList ] .
BreakStmt = "break" [ Label ] .
ContinueStmt = "continue" [ Label ] .
GotoStmt = "goto" Label .
FallthroughStmt = "fallthrough" .
Block = "{" StatementList "}" .
StatementList = { Statement ";" } .
IfStmt = "if" [ SimpleStmt ";" ] Expression Block [ "else" ( IfStmt | Block ) ] .
SwitchStmt = ExprSwitchStmt | TypeSwitchStmt .
ExprSwitchStmt = "switch" [ SimpleStmt ";" ] [ Expression ] "{" { ExprCaseClause } "}" .
ExprCaseClause = ExprSwitchCase ":" StatementList .
ExprSwitchCase = "case" ExpressionList | "default" .
TypeSwitchStmt = "switch" [ SimpleStmt ";" ] TypeSwitchGuard "{" { TypeCaseClause } "}" .
TypeSwitchGuard = [ identifier "!=" ] PrimaryExpr "." "(" "type" ")" .
TypeCaseClause = TypeSwitchCase ":" StatementList .
TypeSwitchCase = "case" TypeList | "default" .
TypeList = Type { "," Type } .
ForStmt = "for" [ Condition | ForClause | RangeClause ] Block .
Condition = Expression .
ForClause = [ InitStmt ] ";" [ Condition ] ";" [ PostStmt ] .
InitStmt = SimpleStmt .
PostStmt = SimpleStmt .
RangeClause = [ ExpressionList "=" | IdentifierList "!=" ] "range" Expression .
DeferStmt = "defer" Expression .
EmptyStmt = .
ExpressionStmt = Expression .
IncDecStmt = Expression ( "++" | "--" ) .
Assignment = ExpressionList assign_op ExpressionList .
assign_op = [ add_op | mul_op ] "=" .
ShortVarDecl = IdentifierList "!=" ExpressionList .

```