Mini Language

1 .Language Definition:

1.1 Alphabet:

a. Upper (A-Z) and lowercase letters (a-z) of the English alphabet

b. Underline character '_';

c. Decimal digits (0-9);

1.2 Lexic:

a. Special symbols, representing:

Operators:

```
+    &     +=    &=     &&    ==    !=
-    |     -=    |=     ||    <     <=
*    ^     *=    ^=     <-    >     >=
/    <<    /=    <<=    ++    =     :=
%    >>    %=    >>=    --    !     ...
```

Separators: `() [] {} ; : , space`

Reserved words:

```
break        continue     func        int      Println
case         map          struct      float64  Print
else         package      switch      char     Scan
const        return       type        bool     Printf
if           for          import      var
```

b. Identifiers:

- a sequence of letters and digits, such that the first character is a letter; the rule is:

Identifier ::= Letter| {Letter| digit}

IdentifierList ::= Identifier {“,” Identifier}

Letter::= "A" | "B" | . ..| "Z" | "a" | "b" | . ..| "z"

Digit::= "0" |”1"|2"|"3"|4"|"5"|"6"|"7"|"8"|"9"

Number ::= ["+" | "-"]Digit{Digit}

Escape_char :: = `\` ("n")

c. constants

1. integer

int_const ::= ["+" | "-"] Digit | Number

bool_const ::= "true" | "false"

2. float

float_const ::= ["+" | "-"] Digit{ Digit} "."Digit{Digit}

3. character

char_const:= "Letter" | "Digit" | ”Number”

4. string

string ::= “”” {character{string}} “”””

character ::= Letter| Digit

2. Syntax:

The words - predefined tokens are specified between " and ":

2.1 Syntactic rules:

2.2 Types

Type ::= "bool" | "int" | "float64" | "char"

2.2.1 Array Type

ArrayType ::= "[" ArrayLength "]" Type
ArrayLength ::= Digit{Digit}

2.2.2 Function Type

FunctionType = "func" Signature .
Signature = Parameters [Result] .
Result = Parameters | Type .
Parameters = "(" [ParameterList [","]] ")" .
ParameterList = ParameterDecl { "," ParameterDecl } .
ParameterDecl = [IdentifierList] ["..."] Type .

2.2.3 Struct Type

```
StructType    = "struct" "{" { FieldDecl ";" } "}" .
FieldDecl     = (IdentifierList Type) [ Tag ] .
Tag           = string .
```

2.3 Blocks

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

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

2.3 Declarations and scope

```
SourceFile       = PackageClause { ImportDecl } { Declaration }
PackageClause  = "package" PackageName .
PackageName    = Identifier .
ImportDecl       = "import" ( ImportSpec | "(" { ImportSpec ";" } ")" ) .
ImportSpec       = [ PackageName ] ImportPath .
ImportPath       = string.
Declaration = VarDecl | FunctionDecl .
```

2.3.1 Expressions

```
ExpressionList = Expression { "," Expression }
Expression ::= UnaryExpr | Expression binary_op Expression
UnaryExpr  ::= PrimaryExpr | unary_op UnaryExpr .
PrimaryExpr ::= Identifier
binary_op  = "||" | "&&" | rel_op | add_op | mul_op .
rel_op     = "==" | "!=" | "<" | "<=" | ">" | ">=" .
add_op     = "+" | "-" .
mul_op     = "*" | "/" .
unary_op   = "+" | "-" | "!" | "&"|   .
```

2.3.2 VarDecl

```
VarDecl     = "var" ( VarSpec | "(" { VarSpec ";" } ")" ) .
VarSpec     = IdentifierList ( Type [ "=" ExpressionList ] | "="
ExpressionList )
ShortVarDecl = IdentifierList ":=" ExpressionList (ex:i, j := 0, 1)
```

2.3.3 FunctionDecl

FunctionDecl = "func" FunctionName [“(“ { Identifier type} {“, ” Identifier type} “)”] Signature [FunctionBody] .
FunctionName = Identifier .
FunctionBody = Block

2.4 Expressions

QualifiedIdent = PackageName "." Identifier . (ex: math.Sin)

Statement ::= Assignstmt | Iostmt | Ifstmt | Forstmt

SimpleStmt = Expression | IncDecStmt | assignment | ShortVarDecl

assignstmt ::= Identifier ":=" Expression

IncDecStmt = Expression ("++" | "--")

WriteStmt ::== “fmt.Print” | “fmt.Printf” | “fmt.Println”

Iostmt ::= "fmt.Scan" "(" Identifier ")" | WriteStmt "(" Identifier ")"

IfStmt = "if" [SimpleStmt ";"] Expression Block .

Condition ::= Expression

ForStmt = "for" [Condition | ForClause | RangeClause] Block .

ForClause = [InitStmt] ";" [condition] ";" [PostStmt] .

InitStmt = SimpleStmt.

PostStmt = SimpleStmt.

RangeClause = [ExpressionList "=" | IentifierList ":="] "range" Expression

ReturnStmt = "return" [ExpressionList]

//Program

```
package main

import (

    "fmt"
```

```
    "math"
)

func main() {
    var k int
    fmt.Print("Enter the value of k: ")
    fmt.Scan(&k)
    fmt.Printf("Prime numbers less than %d are:\n", k)
    for i := 2; i < k; i++ {
        if isPrime(i) {
            fmt.Println(i)
        }
    }
}
func isPrime(n int) bool {
    if n < 2 {
        return false
    }
    sqrtN := int(math.Sqrt(float64(n)))
    for i := 2; i <= sqrtN; i++ {
        if n%i == 0 {
            return false
        }
    }
```

```
    return true

}
```