VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

PRINCIPLES OF PROGRAMMING LANGUAGES - CO3005

MiniGo SPECIFICATION

Version 1.0.2

HO CHI MINH CITY, 01/2025

MiniGo's SPECIFICATION

Version 1.0.2

1 Introduction

Go, also known as Golang, is a statically typed, compiled programming language developed by Google. It was created in 2007 by Robert Griesemer, Rob Pike, and Ken Thompson, and was officially released to the public in 2009. The language was designed to address the shortcomings of other programming languages in terms of performance, scalability, and ease of use, particularly for large software systems. Go emphasizes simplicity, efficiency, and concurrency, making it ideal for modern, distributed applications.

One of the key reasons for Go's creation was to improve the development process at Google. The team wanted a language that would be faster to compile than C++ and more efficient in terms of concurrency management than languages like Java. Go introduces features such as goroutines (lightweight threads) and channels, which facilitate concurrent programming and make it highly effective for handling multiple tasks simultaneously.

Go has become increasingly popular, particularly in cloud computing, microservices, and DevOps. Its growing community and extensive standard library have contributed to its widespread adoption. Some of the most notable projects developed using Go include Docker, Kubernetes, Terraform, and the cloud infrastructure behind companies like Uber, Dropbox, and Netflix. The language's strong performance, ease of use, and scalability have made it a top choice for building large, reliable systems that require high concurrency and low-latency processing.

MiniGo is a simplified version of the Go programming language, designed specifically for students to practice building a compiler within a limited timeframe. It retains the core concepts of Go, such as basic data types, structs, and interfaces, but removes more complex features like goroutines, channels, and the extensive standard library. The goal of MiniGo is to provide a manageable subset of Go that allows students to focus on fundamental concepts of programming language implementation, including lexical analysis, parsing, semantics checking, and code generation. By working with MiniGo, students gain hands-on experience in implementing a programming language, which helps them understand the underlying principles of language design and compilers, while also giving them the opportunity to create a working language from scratch.

2 Program structure

A program in MiniGo consists of a single file that includes various declarations without strict ordering. The file may contain constant declarations, variable declarations, type declarations (such as structs and interfaces), and function declarations, all of which can appear in any order. The program is structured around a mandatory 'main' function, which serves as the entry point of execution. The 'main' function does not take parameters or return values, and it is where the program's execution begins.

3 Lexical structure

3.1 Characters set

A MiniGo program is a sequence of characters from the ASCII character set. Whitespace characters include blank spaces (`' '`), tabs (`'\t'`), form feeds (ASCII FF, `'\f'`), and carriage returns (ASCII CR, `'\r'`). Newline characters (ASCII LF, `'
n'`) are also treated as whitespace in most contexts, but they have a special role in terminating statements, automatically inserting a semicolon where needed. Additionally, newlines are used to determine line numbers in MiniGo, which helps to accurately report errors and provide context in the compiler's error messages.

3.2 Program comment

In MiniGo, there are two types of comments: single-line comments and multi-line comments. Single-line comments begin with // and extend to the end of the line, allowing developers to add brief explanations or notes within the code. These comments are typically used for inline documentation or to temporarily disable code during development. Multi-line comments begin with /* and end with */. They can span multiple lines, making them useful for longer descriptions or for commenting out large blocks of code. A notable feature of MiniGo's multi-line comments is that they support nesting, meaning you can place one /* ... */ comment inside another. Both types of comments are ignored by the compiler and do not affect the execution of the program, but they are important for improving code readability and maintaining clarity in complex code sections.

```
/* This is a /* nested
   multi-line
  comment. */
*/
```

3.3 Tokens set

A token is a sequence of one or more characters in the source code that, when grouped together, acts as a single atomic unit of the language. In the MiniGo programming language, tokens are the smallest meaningful elements in the code and are categorized into five types: identifiers, keywords, operators, separators, and literals. Each token type serves a distinct purpose, contributing to the syntactic and semantic structure of a MiniGo program.

3.3.1 Identifiers

In MiniGo, an **identifier** is the name used to identify variables, constants, types, functions, or other user-defined elements within a program. Identifiers must adhere to the following rules:

1. **Composition**: An identifier must start with a letter (`A-Z` or `a-z`) or an underscore (_). Subsequent characters can include letters, digits (`0-9`), or underscores.
2. **Case Sensitivity**: Identifiers are case-sensitive, meaning `myVariable` and `MyVariable` are treated as distinct identifiers.
3. **Length**: There is no explicit limit on the length of an identifier, but it is recommended to use concise yet descriptive names for clarity.
4. **Keywords Restriction**: Identifiers cannot be the same as any reserved keyword in MiniGo.

Examples of Valid Identifiers: `x`, `userName`, `_tempVar`, `count123`.

Examples of Invalid Identifiers: `123variable` (starts with a digit), `my-variable` (contains a hyphen), `if` (reserved keyword).

Identifiers are essential for making MiniGo programs readable and maintainable, enabling developers to assign meaningful names to program elements.

3.3.2 Keywords

In MiniGo, **keywords** are reserved words that have special meaning within the language. They are used to define the syntax and structure of a MiniGo program and cannot be used as identifiers (e.g., variable names, function names, etc.). The following are the reserved keywords in MiniGo:

`if`	`else`	`for`	`return`	`func`	`type`
`struct`	`interface`	`string`	`int`	`float`	`boolean`
`const`	`var`	`continue`	`break`	`range`	`nil`
`true`	`false`				

Rules for Keywords

- Keywords are case-sensitive. For example, `if` is a valid keyword, but `If` is not.
- Keywords must be used only in their predefined context and cannot be redefined or used as identifiers.

3.3.3 Operators

MiniGo supports the following operators:

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

These operators are essential for performing arithmetic, logical, relational, and other operations in MiniGo. Their specific behaviors and precedence rules will be explained in detail later.

3.3.4 Separators

MiniGo uses the following separators:

- `(`, `)`
- `{`, `}`
- `[`, `]`
- `,`
- `;`

3.3.5 Literals

Literals in MiniGo represent fixed values directly written in the source code. They are classified as follows:

- **Integer Literals:** Integer literals represent whole numbers and can be written in multiple numeric systems:
  - **Decimal Integers:** Written as a sequence of digits from `0` to `9`. Leading zeros are not allowed in decimal integers, except for the number zero (0). Examples: `0`, `42`, `12345`.
  - **Binary Integers:** Represented by a leading `0b` or `0B`, followed by a sequence of binary digits (`0` and `1`). Examples: `0b101`, `0B1101`.

  - **Octal Integers:** Represented by a leading `0o` or `0O`, followed by a sequence of octal digits (`0` through `7`). Examples: `0o12`, `0O77`.
  - **Hexadecimal Integers:** Represented by a leading `0x` or `0X`, followed by a sequence of hexadecimal digits (`0` through `9` and `a` through `f` or `A` through `F`). Examples: `0x1A`, `0XFF`.

- **Floating-point Literals:** Floating-point literals represent real numbers and include an optional fractional part. They consist of:
  - An integer part (`0-9` digits).
  - A mandatory decimal point (`.`) to indicate the fractional part, even if no digits follow it.
  - An optional fractional part (`0-9` digits).
  - Optionally, an exponent part introduced by `e` or `E`, followed by an optional sign (`+` or `-`) and an integer exponent.

  Examples of valid floating-point literals: `3.14`, `0.`, `2.0e10`.

- **String Literals:** String literals represent sequences of characters enclosed in double quotes (`"`). They may contain any ASCII character except for the double quote (`"`) and backslash (\) unless escaped. Escape sequences supported in MiniGo include:
  - \\`n`: Newline
  - \\`t`: Tab
  - \\`r`: Carriage return
  - \\`"`: Double quote
  - \\\\: Backslash

  Examples of valid string literals: `"hello"`, `"123"`, `"This is a string with a newline\n"`.

- **Boolean Literals:** Boolean literals represent the logical values `true` and `false`. They are case-sensitive and must be written in lowercase.

  Examples of valid boolean literals: `true`, `false`.

- **Nil Literal:** The nil literal, `nil`, represents the absence of a value. It is used in situations where a value is optional or non-initialized.

  Example of a valid nil literal: `nil`.

4 Type and Value

In MiniGo, the types of variables can be automatically inferred during compilation, allowing the programmer to omit explicit type annotations while ensuring type safety.

MiniGo supports several primitive (scalar) data types, including `int`, `float`, `boolean`, and `string`. In addition, MiniGo has composite types, such as `struct` and `interface`, which are used to model more complex data structures.

4.1 Boolean type

A value of boolean type can be `true` or `false`.

The following operators can be used on Boolean values:

```
!     &&     ||
```

4.2 Integer type

A value of type `int` can be a whole number, positive or negative.

The following operators are supported for `int` values:

```
+     -     *     /     %
==    !=    >     >=    <     <=
```

4.3 Float type

A value of type `float` can represent real numbers, including those with decimal points.

The following operators can be used with `float` values:

```
+     -     *     /     %
==    !=    >     >=    <     <=
```

4.4 String type

The `string` type represents a sequence of characters enclosed in double quotes `" "`. The following operations are supported for `string` values:

- `+`: Concatenation of two strings.
- `==`: Check if two strings are equal.

- `!=`: Check if two strings are not equal.
- `<`, `<=`, `>`, `>=`: Lexicographical comparison of strings.

Examples:

```
str1 := "Hello"
str2 := "World"
str3 := str1 + " " + str2   // str3 == "Hello World"

str4 := "apple"
str5 := "banana"
result := str4 == str5    // result == false
```

MiniGo supports the **array** type, which represents a collection of elements of the same type.

4.5 Array type

- An array type declaration begins with a list of dimensions followed by a type which is any of the primitive types (`int`, `float`, `boolean`, `string`) or composite types such as `struct`. A dimension is a integer literal or constant enclosed in a pair of squared brackets `[ ]`.
- Arrays are indexed from 0, and the size of the array is fixed once it is defined.

Example of an array declaration:

```
var arr [5]int; // defines an array of 5 integers.
var multi_arr [2][5]int; // defines an array of 2 x 5 integers.
```

4.6 Struct type

A `struct` in MiniGo is a composite data type that allows you to group different types of variables together into a single unit. It is useful for representing objects with different properties, especially when you need to combine various data types under one logical entity.

To define a struct type in MiniGo, you declare a `struct` type using the `type` keyword followed by the name of the struct and the `struct` keyword. The fields of the struct are enclosed within curly braces { } and are defined by a list of field names along with their associated types, each followed by a semicolon or a newline. The `struct` type declaration can end by a semicolon or a newline.

Each field in a struct represents a property of the object, and the type of the field can be any valid MiniGo type (including primitive types, arrays, other structs, and interfaces). You can define a struct with any number of fields, and these fields may be of different types.

Here is an example of how to define a struct in MiniGo:

```
type Person struct {
    name string ;
    age  int ;
}
```

In this example, the `Person` struct has two fields: `name` (of type `string`) and `age` (of type `int`). Notice that the fields are enclosed within curly braces { } and each field consists of a name and a type, separated by a space.

Once you have defined a struct, you can create instances of it by initializing the struct with values for its fields. For example,

```
p := Person{name: "Alice", age: 30}
```

Here, `p` is an instance of the `Person` struct with the field `name` set to `"Alice"` and the field `age` set to `30`. The values inside the curly braces { } are used to initialize the struct fields.

It is also possible to create an empty struct instance where the fields are initialized to their zero values:

```
p := Person{}
```

In this case, `p.name` will be an empty string (`""`) and `p.age` will be zero.

Accessing the fields of a struct can be done using the dot notation:

```
PutStringLn(p.name)  // Output: Alice
PutIntLn(p.age)    // Output: 30
```

You can also modify the fields of a struct instance:

```
p.age := 31
PutIntLn(p.age)    // Output: 31
```

A struct in MiniGo does not have methods by default, but you can define methods associated with a struct type. A method is a function that has a special receiver argument, which is the instance of the struct.

For example, you can define a method `Greet` for the `Person` struct to return a greeting message:

```
func (p Person) Greet() string {
    return "Hello, " + p.name
}
```

This method can be called on an instance of the `Person` struct:

```
PutStringLn(p.Greet())  // Output: Hello, Alice
```

4.7 Interface type

An `interface` defines a set of methods that a type must implement. Any type that implements the methods of an interface satisfies that interface.

To define an interface in MiniGo, you declare an `interface` type using the `type` keyword followed by the name of the interface and the `interface` keyword. The methods of the interface are enclosed within curly braces { }, and each method consists of the method name, a nullable list of parameters enclosed in parentheses, and an optional return type. Parameters can be defined as either a list of names sharing the same type (e.g., x, y int), or as individual name-type pairs separated by commas (e.g., x int, y float). Each method declaration ends with a semicolon or a newline. The interface declaration also ends by a semicolon or a newline.

Example of an interface declaration:

```
type Calculator interface {
    Add(x, y int) int;
    Subtract(a, b float, c int) float;
    Reset()
    SayHello(name string)
}
```

5 Variables, Constants and Function

In a MiniGo program, all variables and constants must be declared before usage. A name cannot be re-declared in the same scope, but it can be reused in other scopes. When a name is re-declared by another declaration in a nested scope, it is hidden in the nested scope.

5.1 Variables

There are three kinds of variables: **global variables**, **local variables**, and **function parameters**.

1. **Global variables**: Global variables are declared outside any function in the program. They are visible from the place where they are declared to the end of the program.

2. **Local variables**: Local variables are declared inside blocks(i.e., inside the body of functions). They are visible only within the block where they are declared and all nested blocks. A block is a list of statement enclosed by a pair of curly braces.

3. **Function parameters**: Unlike in C, formal parameters of a function in MiniGo have their own scope, which is limited to the enclosing function. Parameters will be hidden if redeclared within the body of the function.

In MiniGo, variables of type string, struct, array, and interface are passed by reference, while other types are passed by value.

In the case of passing by value, the callee function receives a copy of the argument's value in its parameters. Thus, the callee function cannot alter the argument's value in the caller function. When a function is called, each parameter is initialized with the corresponding argument's value passed from the caller.

In the case of passing by reference, the callee function receives the address of the argument. Therefore, any modification to the parameter inside the function affects the original argument in the calling function.

In MiniGo, a global or local variable is declared using the `var` keyword, followed by the variable name, an optional type, an optional initialization value and a semicolon or a new line. If the type is omitted, it is automatically inferred from the initialization expression. Initialization is done using an equals sign (`=`) followed by an expression, and the value of this expression must be computable at compile time.

Variables without an explicit initialization value are assigned the zero value of their type. For instance:

- Declaring with an explicit type and initialization: `var x int = 10;`.
- Declaring with inferred type: `var y = "Hello";` (type inferred as `string`).
- Declaring without initialization: `var z int;` (initialized to `0`).

5.2 Constants

In MiniGo, constants are values that cannot be changed once they are assigned. They are used to define fixed values that are immutable throughout the program. Constants must be assigned a value at the time of declaration, and their value cannot be altered later.

Constants can be divided into two types:

1. **Global constants**: These constants are declared outside of any function, and they are accessible throughout the entire program, from the point of declaration to the end of the program.
2. **Local constants**: These constants are declared inside a function or block. Their scope is limited to that function or block, and they are not accessible outside of it.

To declare a constant, you use the `const` keyword followed by the constant's name, its assigned value, and a semicolon or a newline. The value of a constant can either be a literal constant ~~such as a number, string or boolean)~~ or an expression that evaluates to a value.

Examples of constant declarations:

- A global constant with a literal value:

```
const Pi = 3.14;
```

- A local constant inside a function with a literal string:

```
const Greeting = "Hello, MiniGo!";
```

- A constant with a value derived from an expression:

```
const MaxSize = 100 + 50;
```

It is important to note that when declaring a constant, the value must be determined at compile time, meaning that the value can be an expression, but the expression must be fully evaluable before runtime (i.e., without any dynamic computation).

5.3 Functions and Methods

In MiniGo, functions and methods are key components for creating reusable, modular code. A function performs a specific task, while a method is a function associated with a particular type, typically a struct or interface.

Functions

A function in MiniGo is a named block of code that performs a specific task. To declare a function, use the `func` keyword followed by the function name, followed by a pair of parentheses `( )`, which may optionally contain a comma-separated list of parameters. Each parameter follows the same notation as in an Interface type. After the parameters (if any), you can optionally specify a return type. Finally, the function body is enclosed in curly braces

Example of a simple function:

```
func Add(x int, y int) int {
    return x + y;
}
```

Methods

In MiniGo, methods are a type of function that is associated with a specific type, usually a struct or an interface. Methods allow you to define behavior that operates on the fields or properties of an object.

To declare a method for a struct, write it similarly to a function, with the addition of a receiver placed between the `func` keyword and the method name. The receiver consists of a name and a type, enclosed in parentheses.

Example of a method in a struct:

```
type Calculator struct {
    value int;
}

func (c Calculator) Add(x int) int {
    c.value += x;
    return c.value;
}
```

In this example, the `Add` method is associated with the `Calculator` struct. The receiver `c` refers to a `Calculator` instance, allowing the method to modify the struct's `value` field.

Key Differences Between Functions and Methods

- A function is independent of any type and operates as a standalone entity, whereas a method is always tied to a specific type (typically a struct or interface).
- Functions do not have access to the fields of a struct, while methods can access and modify the struct's fields.
- Methods can be declared for structs or interfaces, but not for primitive types or basic data types in MiniGo.

6 Expressions

Expressions in MiniGo are constructs made up of operators and operands that work with data to produce new data. An expression can involve constants, variables, calls, or results from other operators.

In MiniGo, there are several types of operators, including arithmetic, relational, boolean and operators for accessing array and struct elements. Each of these operators is applied to operands of specific types and returns a result based on the operation.

6.1 Arithmetic operators

MiniGo supports the standard arithmetic operators for working with integers and floating-point numbers. In addition, operator + can be used to concat two strings. These operators include:

Operator	Operation	Operand's Type
+	Addition	int/float/string
-	Subtraction	int/float
*	Multiplication	int/float
/	Division	int/float
%	Modulo	int

Note:

- The `+` operator:
  - If both operands are of type `string`, the result is of type `string` (concatenation).
  - If both operands are of type `int` or `float`, the result is of the same type as the operands.
  - If one operand is `int` and the other is `float`, the result is `float`.
- The `-`, `*`, and `/` operators:
  - If both operands are of type `int` or `float`, the result is of the same type as the operands.
  - If one operand is `int` and the other is `float`, the result is `float`.
- The `%` operator:
  - Works only with operands of type `int`, and the result is also of type `int`.

6.2 Relational operators

Relational operators are used to compare values. In MiniGo, these operators require that the operands be of the same type, either both `int`, both `float`, or both `string`. The result of the relational operations is always a `boolean` type.

The following relational operators are supported:

Operator	Operation	Operand's Type
==	Equal	int/float/string
!=	Not equal	int/float/string
<	Less than	int/float/string
>	Greater than	int/float/string
<=	Less than or equal	int/float/string
>=	Greater than or equal	int/float/string

6.3 Boolean operators

Boolean operators are used to perform logical operations. These include:

Operator	Operation	Operand's Type
!	Negation	boolean
&&	Conjunction (AND)	boolean
||	Disjunction (OR)	boolean

6.4 Accessing array elements

To access an element of an array, MiniGo uses the index operator [], where the expression inside the brackets must be evaluated to an integer.

For example:

```
a[2][3] := b[2] + 1;
```

6.5 Accessing struct fields

To access the fields of a struct, MiniGo uses the dot operator .. The dot operator is used to reference a specific field of a struct.

For example:

```
person.name := "John";
person.age := 30;
```

6.6 Literal

In addition to the literals described in Section 3.3.5, MiniGo supports composite literals, such as array and struct literals.

6.6.1 Array Literal

An array literal begins with an array type, followed by a list of elements enclosed in curly braces.

For example:

```
arr := [3]int{10, 20, 30}
marr := [2][3]int{{1, 2, 3}, {4, 5, 6}}
```

6.6.2 Struct Literal

A struct literal begins with the name of the struct, followed by a pair of curly braces containing an optional, comma-separated list of elements. Each element consists of a field name, a colon, and an expression.

For example:

```
p := Person{name: "Alice", age: 30}
q := Person{}
```

6.7 Function and Method Call

In MiniGo, function and method calls are used to invoke the behavior defined by functions and methods, respectively. Both function and method calls are similar, but there are key differences based on whether they are associated with a function or a method of a type.

6.7.1 Function Call

To call a function in MiniGo, use the function name followed by a pair of parentheses `()` containing the actual arguments (if any). The arguments in the parentheses must match the parameters declared in the function's signature, both in number and type.

For example, to call a function named `add` that takes two integer arguments, the call would be written as:

```
add(3, 4)
```

If the function has no parameters, you can call it with empty parentheses:

```
reset()
```

6.7.2 Method Call

A method call in MiniGo is similar to a function call, but with the addition of the receiver type. Methods are associated with types (usually structs), and the receiver type is defined as part of the method declaration.

To call a method, use the expression representing the instance of the type, followed by a dot `.` and the method name, with the actual arguments in parentheses. The receiver type is implicitly passed when the method is called.

For example, if `calculator` is an instance of a struct `Calculator`, and the method `add` is defined for this struct, you would call it like this:

```
calculator.add(3, 4)
```

If the method has no parameters, it is called with empty parentheses:

```
calculator.reset()
```

6.8 Operator Precedence and Associativity

In MiniGo, operators are evaluated on the basis of their precedence and associativity. The following table summarizes the precedence and associativity of all operators:

Operator	Precedence	Associativity
`[ ]`, `.`	1 (Highest)	Left-to-right
`!`, `-` (unary)	2	Right-to-left
`*`, `/`, `%`	3	Left-to-right
`+`, `-` (binary)	4	Left-to-right
`==`, `!=`, `<`, `<=`, `>`, `>=`	5	Left-to-right
`&&`	6	Left-to-right
`||`	7 (Lowest)	Left-to-right

The expression in parentheses has highest precedence so the parentheses are used to change the precedence of operators.

7 Statements

A statement in MiniGo specifies an action for the program to perform. Each statement must end with a semicolon (`;`). However, the semicolon can be omitted if the statement ends with a newline.

There are many kinds of statements, as described as follows:

7.1 Variable and Constant Declaration Statement

Variable and constant declaration statement have been described in section 5.

7.2 Assignment Statement

An assignment statement in MiniGo consists of a left-hand side (LHS), an assignment operator, and a right-hand side (RHS). The LHS can be a scalar variable, an array element access (e.g., `arr[index]`), or a struct field access (e.g., `structName.fieldName`). MiniGo supports the assignment operators `:=`, `+=`, `-=`, `*=`, `/=`, and `%=`. The RHS is any valid expression, and its value must be compatible with the type of the LHS.

For example: `x := 5;` initializes `x` with 5, `x += 10;` adds 10 to `x`, and `arr[2] *= 3;` multiplies the third element of `arr` by 3.

When the left-hand side of an assignment statement is an undeclared scalar variable and the variable does not appear in the right-hand side expression, the variable is considered declared by this statement and remains in scope until the end of the block containing the statement.

7.3 If Statement

An **if statement** in MiniGo is used to execute a block of code conditionally, based on the evaluation of a boolean expression. The syntax consists of the `if` keyword, followed by a boolean expression enclosed in parentheses, and a block of code enclosed in curly braces. Optionally, an `else` clause may follow to define a block of code to be executed when the condition is false.

The structure of an `if` statement is as follows:

- The boolean expression inside the parentheses must evaluate to `true` or `false`.
- If the condition evaluates to `true`, the statements within the first block are executed.
- If the condition evaluates to `false`, the statements within the optional `else` block are executed.
- Additional conditions can be checked using the `else if` clause, which follows the same structure as the `if` clause.

For example,

```
if (x > 10) {
    println("x is greater than 10");
} else if (x == 10) {
    println("x is equal to 10");
} else {
    println("x is less than 10");
}
```

Note:

- Every block must be enclosed in curly braces, even if it contains only one statement.
- The boolean expression must be valid and not produce runtime errors. For instance, all variables within the expression must be declared and initialized.
- Nested `if` statements are allowed and can be used to create more complex decision-making logic.

7.4 For Statement in MiniGo

The `for` statement in MiniGo provides three common forms of iteration: basic form with only a logical expression, form with initialization, and form for iterating over an array. Below is a detailed description of each form.

7.4.1 Basic For Loop

In the basic form of a `for` loop, only a logical expression (condition) is used to control the loop execution. The syntax is as follows:

```
for condition {
    // statements
}
```

Here, the `condition` is a boolean expression that is evaluated before each iteration. The loop continues executing as long as the `condition` evaluates to `true`. If the condition evaluates to `false`, the loop exits.

Example:

```
for i < 10 {
    // loop body
}
```

In this case, the loop will run as long as the value of `i` is less than 10.

7.4.2 For Loop with Initialization, Condition, and Update

In this form, the `for` statement includes an initialization expression, a condition expression, and an update expression. The syntax is as follows:

```
for initialization; condition; update {
    // statements
}
```

- **Initialization**: This is executed only once before the loop begins and usually used to set the initial state of a loop scalar variable used in the loop. This is written in the form of assignment statement or a variable declaration with initialization.
- **Condition**: This is a boolean expression that is evaluated before each iteration. The loop continues as long as this expression evaluates to `true`. If the condition becomes `false`, the loop terminates.

- **Update**: This part is written as an assignment statement which is executed after each iteration. It is typically used to update ~~or increment~~ the loop scalar variable.

Example:

assign stmt (vì nếu là var Decl thì phải ghi i := 0)

```
for i := 0; i < 10; i += 1 {
    // loop body
}
```

In this example, `i` is initialized to 0, and the loop will continue as long as `i` is less than 10. After each iteration, `i` is incremented by 1.

7.4.3 For Loop with Range (Array Iteration)

In MiniGo, the `for` loop can also be used to iterate over the elements of an array using the `range` keyword. The syntax for iterating over an array is as follows:

```
for index, value := range array {
    // statements
}
```

- **index**: This is a scalar variable keeping the index of the current element in the array. It is of type `int`. In each iteration, `index` will take the value of the position of the element within the array (starting from 0).
- **value**: This is a scalar variable keeping the value of the current element in the array. It matches the type of the elements in the array. In each iteration, `value` will hold the element at the current `index`.
- **array**: This is the array that is being iterated over. The `array` is the collection of elements that the loop is iterating through. It must be of a fixed array type (since MiniGo does not support slices or maps).

The `range` keyword is used to iterate over arrays in MiniGo, which only supports arrays (not slices or maps). The loop iterates through each element, providing both the index and the value of the element during each iteration.

Example 1 (Array iteration):

```
arr := [3]int{10, 20, 30}
for index, value := range arr {
    // index: 0, 1, 2
    // value: 10, 20, 30
}
```

Example 2 (Array iteration without the `index`): If you do not need the index, you can omit it by using the blank identifier _.

```
arr := [3]int{10, 20, 30}
for _, value := range arr {
    // value: 10, 20, 30
}
```

In this example, only the `value` of each element is used, and the index is ignored.

7.5 Break Statement

The `break` statement in MiniGo is used to immediately terminate the execution of a `for` loop. When `break` is encountered, the control flow will jump to the first statement that follows the loop, effectively exiting the loop early.

The syntax for the `break` statement is:

```
break;
```

Note: The `break` statement does not require any condition. It immediately exits the nearest enclosing `for` loop, regardless of the current iteration or condition.

Example: Breaking out of a `for` loop when a certain condition is met:

```
for i := 0; i < 10; i+=1 {
    if (i == 5) {
        break;
    }
    // other statements
}
```

In this example, the `for` loop will terminate when the value of `i` reaches 5, and control will be transferred to the first statement following the loop.

Important: The `break` statement is used exclusively in loops such as `for`. It can only break out of the innermost loop in which it is placed.

7.6 Continue Statement

The `continue` statement in MiniGo is used to skip the remaining part of the current iteration of a loop and immediately proceed to the next iteration. When `continue` is encountered, the control flow jumps back to the loop's condition, effectively causing the next iteration to begin.

The syntax for the `continue` statement is:

```
continue;
```

Note: The `continue` statement does not terminate the loop; instead, it skips the rest of the current iteration and continues with the next iteration.

Example: Skipping over an iteration in a `for` loop when a certain condition is met:

```
for i := 0; i < 10; i+=1 {
    if (i == 5) {
        continue;
    }
    // statements that will not execute when i == 5
}
```

In this example, when the value of `i` is 5, the `continue` statement causes the loop to skip the remaining part of the current iteration and continue with the next iteration, i.e., `i` will be incremented and the next iteration will begin.

Important: The `continue` statement can only be used inside loops, such as `for` loops. It only affects the current iteration and does not terminate the loop itself.

7.7 Call statement

A **function or method call statement** is similar to a function or method call, except that it is not part of an expression and is always terminated by a semicolon or a newline.

For example:

```
foo(2 + x, 4 / y); m.goo();
```

7.8 Return statement

A **return statement** is used to transfer control and data back to the caller of the function in which it appears. The return statement begins with the keyword `return`, which may optionally be followed by an expression.

If the function or method has a return type, a `return` statement is required to return an appropriate value.

8 Scope

In MiniGo, there are three kinds of scopes: global, function/method, and local.

- **Global scope**: This scope applies to all declarations that are made outside of functions or methods. For variable and constant declarations, the scope extends from the point of declaration to the end of the program. For other kinds of declarations, such as type or function declarations, the scope applies throughout the entire program.
- **Function/method scope**: This scope applies to the parameters of a function or method. The scope begins from the point of parameter declaration and continues until the end of the enclosing function or method.
- **Local scope**: This scope applies to variables and constants declared within a block. The scope begins at the point of declaration and extends until the end of the enclosing block.

9 Built-in Functions

For convenience, MiniGo provides the following built-in functions:

`func getInt()int`: reads and returns an integer value from the standard input
`func putInt(i int)`: prints the value of the integer i to the standard output
`func putIntLn(i int)`: same as putInt except that it also prints a newline
`func getFloat() float`: reads and returns a floating-point value from the standard input
`func putFloat(f float)`: prints the value of the float f to the standard output
`func putFloatLn(f float)`: same as putFloat except that it also prints a newline
`func getBool()boolean`: reads and returns a boolean value from the standard input
`func putBool(b boolean)`: prints the value of the boolean b to the standard output
`func putBoolLn(b boolean)`: same as putBoolLn except that it also prints a new line
`func getString()string`: reads and returns a string value from the standard input
`func putString(s string)`: prints the value of the string to the standard output
`func putStringLn(s string)`: same as putStringLn except that it also prints a new line
`func putLn()`: prints a newline to the standard output

10 Change log

Version 1.0.1

- Add = and := into the list of operators in Section 3.3.3

Version 1.0.2

- Change True, False to true, false in Section 4.1
- Determine that the loop variable in for statement is scalar in Section 7.4.2

- Some modifications (with strike through or underline) in Section 5
- Fix the example (replacing i++ by i+=1) in Section 7.5 and 7.6