CS335 Milestone 1
Specifications & Language Manual

BY

GROUP 11

ABHAY MISHRA, 190017
ASHOK KUMAR SAINI, 190195
ASHUTOSH SHUKLA, 170171
GAGAN ARYAN, 190327

SIT Triple

Source : Golang

Implementation : C++

Target : MIPS

Contents

Chapter 1

Source Code Representation

Source code is a sequence of ASCII Characters each of size 1 byte. Each character is distinct; for instance, upper and lower case letters are different characters.

1.1 Characters

The following terms are used to denote specific character classes:

```
newline          = /* ASCII Code 0x0A , Unicode code point U+000A*/
ascii_char       = /* an arbitrary ASCII character except newline */
ascii_letter     = /* An ASCII character whose Unicode code point
                      is classified as "Letter" */
ascii_digit      = /* An ASCII character whose Unicode code point is
                      classified as "Number, decimal" */
```

1.2 Letters and digits

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

```
letter        = ascii_letter | "_" .
decimal_digit = "0" ... "9" .
binary_digit  = "0" | "1" .
octal_digit   = "0" ... "7" .
hex_digit     = "0" ... "9" | "A" ... "F" | "a" ... "f" .
```

Chapter 2

Lexical Elements

2.1 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 */.

2.2 Tokens

Tokens form the vocabulary of the Go language. There are four classes:

1. Identifiers
2. Keywords
3. Operators and Punctuations
4. Literals

2.2.1 Identifiers

Identifiers name program entities such as variables and types. An identifier is a sequence of one or more letters and digits. The first character in an identifier must be a letter.

```
identifier = letter { letter | ascii_digit } .
```

2.2.2 Keywords

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

```
break       func        var
case        struct      return
else        goto        package
const       if          range
continue    for         import
```

2.2.3 Operators and punctuations

The following character sequences represent operators (including assignment operators) and punctuation:

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

2.2.4 Literals

2.2.4.1 Integer Literals

An integer literal is a sequence of digits representing an integer constant. An optional prefix sets a non-decimal base: 0b or 0B for binary, and 0x or 0X for hexadecimal. A single 0 is considered a decimal zero. In hexadecimal literals, letters a through f and A through F represent values 10 through 15.

```
int_lit        = decimal_lit | binary_lit | hex_lit .
decimal_lit    = "0" | ( "1" ... "9" ) [ decimal_digits ] .
binary_lit     = "0" ( "b" | "B" ) binary_digits .
hex_lit        = "0" ( "x" | "X" ) hex_digits .

decimal_digits = decimal_digit { decimal_digit } .
binary_digits  = binary_digit { binary_digit } .
hex_digits     = hex_digit { hex_digit } .
```

2.2.4.2 Floating-point literals

A floating-point literal is a decimal representation of a floating-point constant.

A decimal floating-point literal consists of an integer part (decimal digits), a decimal point, a fractional part (decimal digits), and an exponent part (e or E followed by an optional sign and decimal digits). One of the integer part or the fractional part may be elided; one of the decimal point or the exponent part may be elided. An exponent value exp scales the mantissa (integer and fractional part) by 10exp.

```
float_lit         = decimal_digits "." [ decimal_digits ] [ decimal_exponent ] |
                    decimal_digits decimal_exponent |
                    "." decimal_digits [ decimal_exponent ] .
decimal_exponent  = ( "e" | "E" ) [ "+" | "-" ] decimal_digits .
```

2.2.4.3 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.

Raw string literals are character sequences between back quotes, as in ‘foo‘. Within the quotes, any character may appear except back quote. The value of a raw string literal is the string composed of the uninterpreted characters between the quotes; in particular, backslashes have no special meaning and the string may contain newlines.

Interpreted string literals are character sequences between double quotes, as in ”bar”. Within the quotes, any character may appear except newline and unescaped double quote.

```
string_lit             = raw_string_lit | interpreted_string_lit .
raw_string_lit         = "‘" { ascii_char | newline } "‘" .
interpreted_string_lit = ‘"‘ { ascii_value | byte_value } ‘"‘ .
```

While breaking the input into tokens, the next token is the longest sequence of characters that form a valid token.

2.3 Separators

White space, formed from spaces (U+0020), horizontal tabs (U+0009), and newlines (U+000A), is ignored except as it separates tokens that would otherwise combine into a single token.

2.4 Semicolons

The formal grammar uses semicolons ";" as terminators in a number of productions.

Chapter 3

Constants and Variables

3.1 Constants

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

A constant value is represented by an integer, floating-point or string literal, an identifier denoting a constant, a constant expression, or the result value of some built-in functions such as len applied to some expressions. The boolean truth values are represented by the predeclared constants true and false.

3.1.1 Declaration and Scope

A constant declaration binds a list of identifiers (the names of the constants) to the values of a list of constant expressions. The number of identifiers must be equal to the number of expressions, and the nth identifier on the left is bound to the value of the nth expression on the right.

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

IdentifierList = identifier { "," identifier } .
ExpressionList = Expression { "," Expression } .
```

If the type is present, all constants take the type specified, and the expressions must be assignable to that type. If the type is omitted, the constants take the individual types of the corresponding expressions. If the expression values are untyped constants, the declared constants remain untyped and the constant identifiers denote the constant values. For instance, if the expression is a floating-point literal, the constant identifier denotes a floating-point constant, even if the literal's fractional part is zero.

```
const Pi float64 = 3.14159265358979323846
const zero = 0.0         // untyped floating-point constant
const (
 size int64 = 1024
 eof        = -1  // untyped integer constant
)
const a,b,c=3,4,"foo" //a=3,b=4,c="foo", untyped integer and string constants
const u, v float32 = 0, 3    // u = 0.0, v = 3.0
```

3.2 Variables

A variable is a storage location for holding a value. The set of permissible values is determined by the variable's type.

A variable declaration or, for function parameters and results, the signature of a function declaration reserves storage for a named variable.

Structured variables of array and struct types have elements and fields that may be addressed individually. Each such element acts like a variable.

The static type (or just type) of a variable is the type given in its declaration, the type provided in the new call or the type of an element of a structured variable.

A variable's value is retrieved by referring to the variable in an expression; it is the most recent value assigned to the variable.

3.2.1 Declaration and Scope

A declaration binds a non-blank identifier to a constant, type, variable, function, label, or package. Every identifier in a program must be declared. No identifier may be declared twice in the same block, and no identifier may be declared in both the file and package block.

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 ) .

    var i int
    var U, V, W float64
    var k = 0
    var x, y float32 = -1, -2
```

The declaration of variables of pointer, array and string type is similar to that of numeric types. e.g.

```
var p *int // initialized to NULL by default
var a [2]string
var b [3]int
```

Chapter 4

Data Types

A type determines a set of values together with operations 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 | QualifiedIdent .
TypeLit   = ArrayType | StructType | PointerType | FunctionType
```

4.1 Numeric Types

A numeric type represents sets of integer or floating-point values. The predeclared architecture-independent numeric types are:

```
uint8       the set of all unsigned  8-bit integers (0 to 255)
uint16      the set of all unsigned 16-bit integers (0 to 65535)
uint32      the set of all unsigned 32-bit integers (0 to 4294967295)
uint64      the set of all unsigned 64-bit integers (0 to 18446744073709551615)

int8        the set of all signed  8-bit integers (-128 to 127)
int16       the set of all signed 16-bit integers (-32768 to 32767)
int32       the set of all signed 32-bit integers (-2147483648 to 2147483647)
int64       the set of all signed 64-bit integers (-9223372036854775808 to 922337203685477758

float32     the set of all IEEE-754 32-bit floating-point numbers
float64     the set of all IEEE-754 64-bit floating-point numbers
```

```
byte        alias for uint8
```

The value of an n-bit integer is n bits wide and represented using two's complement arithmetic.

Note: Golang doesn't have a char data type. The byte data type represents ASCII characters.

4.2 Boolean Types

A boolean type represents the set of Boolean truth values denoted by the predeclared constants true and false. The predeclared boolean type is bool;

4.3 String Types

A string type represents the set of string values. A string value is a (possibly empty) sequence of bytes. The number of bytes is called the length of the string and is never negative.

The length of a string s can be discovered using the built-in function `len`. The length is a compile-time constant if the string is a constant. A string's bytes can be accessed by integer indices 0 through `len(s)-1`.

4.4 Pointers

A pointer type denotes the set of all pointers to variables of a given type, called the base type of the pointer. The value of an uninitialized pointer is `nil`.

```
    PointerType = *BaseType .
    BaseType    = Type .
    *Point
    *[4]int
```

Chapter 5

Expressions and operators

An expression specifies the computation of a value by applying operators and functions to operands.

5.1 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.

5.2 Qualified identifiers

A qualified identifier is an identifier qualified with a package name prefix. Both the package name and the identifier must not be blank.

```
QualifiedIdent = PackageName "." identifier .
```

A qualified identifier accesses an identifier in a different package, which must be imported. The identifier must be exported and declared in the package block of that package.

```
math.Ceil // denotes the Ceil function in package math
```

5.3 Primary expressions

Primary expressions are the operands for unary and binary expressions.

```
 PrimaryExpr =
Operand |
PrimaryExpr Selector |
PrimaryExpr Index |
PrimaryExpr Arguments .

Selector       = "." identifier .
Index          = "[" Expression "]" .
Arguments      = "(" [ ( ExpressionList | Type [ "," ExpressionList ]
```

5.4 Selectors

For a primary expression x that is not a package name, the selector expression `x.f` denotes the field or method f of the value x (or sometimes *x). The identifier f is called the (field or method) selector; it must not be the blank identifier. The type of the selector expression is the type of f. If x is a package name, see the section on qualified identifiers.

5.5 Index expressions

A primary expression of the form `a[x]` denotes the element of the array, pointer to array and string indexed by x. The value x is called the index. The following rules apply in general:

- The index x must be of integer type
- A constant index must be non-negative and representable by a value of type int
- The index x is in range if $0 \leq x < len(a)$, otherwise it is out of range

For a of array type A:

- A constant index must be in range
- A[x] is the array element at index x and the type of a[x] is the element type of A

For a of pointer to array type:

- A[x] is shorthand for (*a)[x]

For a of string type:

- A constant index must be in range if the string a is also constant
- a[x] may not be assigned to

Otherwise a[x] is illegal.

5.6 Calls

Given an expression f of function type F,

```
            f(a1, a2, ... an)
```

calls f with arguments a1, a2, ... an. Arguments must be single-valued expressions assignable to the parameter types of F and are evaluated before the function is called. The type of the expression is the result type of F.

5.7 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   = "+" | "-" | "!" | "^" | "*" | "&" .
```

5.7.1 Operator precedence

Unary operators have the highest precedence. As the ++ and −− operators form statements, not expressions, they fall outside the operator hierarchy. As a consequence, statement $*p++$ is the same as $(*p)++$.

There are five precedence levels for binary operators. Multiplication operators bind strongest, followed by addition operators, comparison operators, && (`logical AND`), and finally || (`logical OR`):

Precedence	Operator
5	`* / % << >> &`
4	`+ - | ^`
3	`== != < <= > >=`
2	`&&`
1	`||`

Binary operators of the same precedence associate from left to right. For instance, $x/y*z$ is the same as $(x/y)*z$.

5.8 Airthmetic operators

Arithmetic operators apply to numeric values and yield a result of the same type as the first operand. The four standard arithmetic operators `(+, -, *, /)` apply to integer, floating-point, and complex types; + also applies to strings. The bitwise logical and shift operators apply to integers only.

`+`	`sum`	`integers, floats, complex values, strings`
`-`	`difference`	`integers, floats, complex values`
`*`	`product`	`integers, floats, complex values`
`/`	`quotient`	`integers, floats, complex values`
`%`	`remainder`	`integers`
`&`	`bitwise AND`	`integers`
`|`	`bitwise OR`	`integers`
`^`	`bitwise XOR`	`integers`
`<<`	`left shift`	`integer << integer >= 0`
`>>`	`right shift`	`integer >> integer >= 0`

5.8.1 Integer operators

For two integer values x and y, the integer quotient $q = x/y$ and remainder $r = x\%y$ satisfy the following relationships:

```
        x = q*y + r  and  |r| < |y|
```

with x/y truncated towards zero (”truncated division”).

If the divisor is a constant, it must not be zero.

For integer operands, the unary operators $+, -,$ and $\wedge$ are defined as follows:

```
+x                          is 0 + x
-x    negation              is 0 - x
^x    bitwise complement    is m ^ x  with m = "all bits set to 1" for unsigned x
                                      and  m = -1 for signed x
```

5.8.2 Floating-point operators

For floating-point and complex numbers, $+x$ is the same as x, while $-x$ is the negation of x. The result of a floating-point by zero is not specified beyond the IEEE-754 standard;

5.8.3 String Concatenation

Strings can be concatenated using the + operator or the += assignment operator:

s := ”hi” + string(c) s += ” and good bye”

String addition creates a new string by concatenating the operands.

5.9 Comparison operators

Comparison operators compare two operands and yield an untyped boolean value.

```
    ==    equal
    !=    not equal
    <     less
    <=    less or equal
    >     greater
    >=    greater or equal
```

In any comparison, the first operand must be assignable to the type of the second operand, or vice versa.

The equality operators == and != apply to operands that are comparable. The ordering operators <, <=, > and >= apply to operands that are ordered. These terms and the result of the comparisons are defined as follows:

- Boolean values are comparable. Two boolean values are equal if they are either both true or both false.
- Integer values are comparable and ordered, in the usual way.
- Floating-point values are comparable and ordered, as defined by the IEEE-754 standard.
- String values are comparable and ordered, lexically byte-wise.
- Pointer values are comparable. Two pointer values are equal if they point to the same variable or if both have value nil. Pointers to distinct zero-size variables may or may not be equal.

5.10 Logical operators

Logical operators apply to boolean values and yield a result of the same type as the operands. The right operand is evaluated conditionally.

```
&&    conditional AND    p && q  is  "if p then q else false"
||    conditional OR     p || q  is  "if p then true else q"
!     NOT                !p      is  "not p"
```

5.11 Address operators

For an operand x of type T, the address operation &x generates a pointer of type *T to x. The operand must be addressable, that is, either a variable, pointer indirection; or a field selector of an addressable struct operand; or an array indexing operation of an addressable array.

For an operand x of pointer type *T, the pointer indirection *x denotes the variable of type T pointed to by x. If x is nil, an attempt to evaluate *x will cause a run-time panic.

Chapter 6

Array and Structs

6.1 Built in Function

6.1.1 len

If would take the input as reference to the array and returns the length represented as a non-negative int. We would take of length in case of different data types.

6.2 Array

An array is a numbered sequence of elements of a single type, called the element type. The number of elements is called the length of the array and is never negative.

```
ArrayType   = "[" ArrayLength "]" ElementType .
ArrayLength = Expression .
ElementType = Type .
```

The length is part of the array's type; it must evaluate to a non-negative constant representable by a value of type int. The length of array a can be discovered using the built-in function `len`. The elements can be addressed by integer indices 0 through `len(a)-1`. Array types are always one-dimensional but may be composed to form multi-dimensional types.

```
[32]byte
[2*N] struct [ x, y int32 ]
```

```
[1000]*float64
[3][5]int
[2][2][2]float64  // same as [2]([2]([2]float64))
```

6.3 Struct

Struct types A struct is a sequence of named elements, called fields, each of which has a name and a type. Field names may be specified explicitly (IdentifierList) or implicitly (EmbeddedField). Within a struct, non-blank field names must be unique.

```
StructType    = "struct" "{" { FieldDecl ";" } "}" .
FieldDecl     = (IdentifierList Type | EmbeddedField ) [ Tag ] .
EmbeddedField = [ " * " ] TypeName .
Tag           = string_lit .
// An empty struct.
struct { }

// A struct with 6 fields.
struct {
    x, y int
    u float32
    _ float32  // padding
    A *[ ]int
    F func()
}
```

Chapter 7

Control Structures

7.1 Conditionals

7.1.1 If statements

"If" statements specify the conditional execution of two branches according to the value of a boolean expression. If the expression evaluates to true, the "if" branch is executed, otherwise, if present, the "else" branch is executed.

```
IfStmt = "if" [ SimpleStmt ";" ] Expression Block [ "else" ( IfStmt | Block ) ] .
if x > max {
    x = max
}
```

The expression may be preceded by a simple statement, which executes before the expression is evaluated.

```
if x := f(); x < y {
     return x
} else if x > z {
     return z
} else {
     return y
}
```

7.2 Loops

7.2.1 For Statements

A ”for” statement specifies repeated execution of a block. There are three forms: The iteration may be controlled by a single condition, a ”for” clause, or a ”range” clause.

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

7.2.1.1 For statements with single condition

In its simplest form, a ”for” statement specifies the repeated execution of a block as long as a boolean condition evaluates to true. The condition is evaluated before each iteration. If the condition is absent, it is equivalent to the boolean value true.

```
for a < b {
    a *= 2
}
```

7.2.1.2 For statements with for clause

A ”for” statement with a ForClause is also controlled by its condition, but additionally it may specify an init and a post statement, such as an assignment, an increment or decrement statement. The init statement may be a short variable declaration, but the post statement must not. Variables declared by the init statement are re-used in each iteration.

```
ForClause = [ InitStmt ] ";" [ Condition ] ";" [ PostStmt ] .
InitStmt = SimpleStmt .
PostStmt = SimpleStmt .
for i := 0; i < 10; i++ {
     f(i)
}
```

If non-empty, the init statement is executed once before evaluating the condition for the first iteration; the post statement is executed after each execution of the block (and only if the block was executed). Any element of the ForClause may be empty but the semicolons are required unless there is only a condition. If the condition is absent, it is equivalent to the boolean value true.

```
for cond { S() }    is the same as    for ; cond ; { S() }
for      { S() }    is the same as    for true     { S() }
```

Chapter 8

Function and Function Pointers

8.1 Functions

8.1.1 Function Type

A function type denotes the set of all functions with the same parameter and result types. The value of an uninitialized variable of function type is `nil`. Format:

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

Here `func` is keyword. Usage example:

```
func()
func(x int) int
func(a, _ int, z float32) bool
func(a, b int, z float32) (bool)
```

Note that parameter names are optional so `func(x int) int` is same as `func(int) int`. The parameter names must either be all present or all absent. We currently look to implement functions with only one return value.
Two function types are identical if they have the same number of parameters and result values, corresponding parameter and result types are identical. Parameter and result names are not required to match.

8.1.2 Function Declaration

Function declarations binds a function name (i.e. an identifier) to a function.

```
    FunctionDecl = func FunctionName Signature [ FunctionBody ] .
```

If the function's signature declares result parameters, the function body's statement list must end in a terminating statement - return or goto.

8.2 Function Pointers

Function Pointers are achieved in go by use of function variables.
eg. `f func(func(int,int) int, int) int` declares a function with first input parameter also a function, which is analogous to C/C++ function pointers. Like any other type, function types can be assigned to function variables,
eg: assuming function `func sub (num1 int, num2 int) int` { `*body*` } is declared we can have, in main(), `var fn1 func(int,int) int = sub`.

Chapter 9

Input/Output statements

Package fmt implements formatted I/O with functions analogous to C's printf and scanf. The format 'verbs' are derived from C's but are simpler.

9.0.1 Printing

The verbs:

General:

```
    %v the value in a default format when printing structs, the plus flag (%+v) adds field r
    %#v a Go-syntax representation of the value
    %T a Go-syntax representation of the type of the value
    %% a literal percent sign; consumes no value
```

Boolean:

```
    %t the word true or false
```

Integer:

```
    %b base 2
    %c the character represented by the corresponding Unicode code point
    %d base 10
    %o base 8
    %O base 8 with 0o prefix
```

```
%q a single-quoted character literal safely escaped with Go syntax.
%x base 16, with lower-case letters for a-f
%X base 16, with upper-case letters for A-F
%U Unicode format: U+1234; same as "U+%04X"
```

Floating-point:

```
%e scientific notation, e.g. -1.234456e+78
%E scientific notation, e.g. -1.234456E+78
%f decimal point but no exponent, e.g. 123.456
%F synonym for %f
```

String:

```
%s the uninterpreted bytes of the string
%q a double-quoted string safely escaped with Go syntax
```

Pointer:

```
%p base 16 notation, with leading 0x
The %b, %d, %o, %x and %X verbs also work with pointers,
formatting the value exactly as if it were an integer.
```

9.0.2 Scanning

Scanf scans formatted text to yield values.Scanf parse the arguments according to a format string, analogous to that of Printf. In the text that follows, 'space' means any Unicode whitespace character except newline.

The verbs behave analogously to those of Printf. For example, $\%x$ will scan an integer as a hexadecimal number, and $\%v$ will scan the default representation format for the value. The Printf verbs $\%p$ and $\%T$ and the flags # and are not implemented. For floating-point and complex values, all valid formatting verbs ($\%b$ $\%e$ $\%E$ $\%f$ $\%F$ $\%g$ $\%G$ $\%x$ $\%X$ and $\%v$) are equivalent and accept both decimal and hexadecimal notation (for example: "2.3e+7", "0x4.5p-8") and digit-separating underscores (for example: "3.14159_26535_89793").

Chapter 10

Libraries

10.1 Math Library

10.1.1 Ceil

`double Ceil(float x )`: returns the smallest integer value greater than or equal to x.
Syntax - double Ceil(float x)
We use double so that in case the integral part of float excedes the int range

10.1.2 Floor

`double Floor(double x)` : returns the largest integer value less than or equal to x.
Syntax - double Floor(float x)
We use double so that in case the integral part of float excedes the int range

10.1.3 Abs

`double Abs(double x)` : returns the absolute value of x.
Syntax : double Abs(double x)

10.1.4 Log

`double Log(double x)` : returns the natural logarithm (base-e logarithm) of x.
Syntax : double Log(double x)

10.1.5 Sqrt

`double Sqrt(double x)]` : returns the square root of x.
Syntax : double Sqrt(double x)

10.1.6 Pow

`double Pow(double x, double y)` : returns x raised to the power of y i.e. x^y.
Syntax : double Pow(double x, double y)

10.1.7 Pow_z

`double Pow_z(double x, double y, double z)` : returns x raised to power of y and divided by z i.e. (x^y)%y.
Syntax : double Pow_z(double x, double y, double z)

10.2 String

Let str_pac be the string package of Go and a, b be two strings

10.2.1 Contains

`func Contains(s, substr string) bool`
Contains reports whether substr is within s.
Ex - str_pac.Contains(”test” , ”es”) return true

10.2.2 Count

`func Count(s, substr string) int`
Count counts the number of non-overlapping instances of substr in s. If substr is an empty string, Count returns 1 + the number of Unicode code points in s.
Ex - str_pac.Count(”test” , ”t”) returns 2

10.2.3 HasPrefix

`func HasPrefix(s, prefix string) bool`
HasPrefix tests whether the string s begins with prefix.
Ex - str_pac.HasPrefix(”test”, ”te”) returns true

10.2.4 HasSuffix

`func HasSuffix(s, suffix string) bool`
HasSuffix tests whether the string s ends with suffix.
Ex - str_pac.HasSuffix(”test”, ”st”) return true

10.2.5 Index

`func Index(s, substr string) int`
Index returns the index of the first instance of substr in s, or -1 if substr is not present in s.
Ex - str_pac.Index(”test”, ”e”) returns 1

10.2.6 ToLower

`func ToLower(s string) string`
ToLower returns s with all Unicode letters mapped to their lower case.
Ex - str_pac.ToLower(”TEST”) returns test

10.2.7 ToUpper

`func ToUpper(s string) string`
ToUpper returns s with all Unicode letters mapped to their upper case.
Ex - str_pac.ToUpper(”test”) returns TEST

Chapter 11

Code Optimizations

Compiler would support a few optimizations that would improve the intermediate code and produce better target code. Our primary goal would be to introduce high level optimizations to the program initially and we will be looking into low level optimizations only after completing all the other features already mentioned in this manual. A few of the optimizations that we plan to introduce are sketched below.

11.1 Constant Propagation

Here we will be evaluating the expressions in compiler time by replacing the constants in the expressions with their values. For instance the below expression

```
a = 11.4
b = a/4.4
```

will be evaluated as b = 11.4/4.4 at compile time.

11.2 Dead Code Elimination

A variable is live at a particular point in the program if its value can be used after that point. If its value cannot be used subsequently, that particular variable is said to be dead at that point. We plan on optimizing the compilation such that the dead code is removed in the intermediate code. Example -

```
int i = 0;
for (i) {
    i = i + 1;
}
```

11.3 Constant Expression Evaluation

If there is any expression in the program consisting of only constant values, then the compiler would calculate the value of that expression at compile time. Example -

```
a=4.50/2 is replace by
a=2.25
```