SLANG FOR SWIFT-4

The Art of Compiler Construction using SWIFT-4

CHAPTER – 3

STATEMENT

The crux of the “SLANG For swift4” can be summed up in two sentences

*** Expression is what you evaluate for it's value**

*** Statement is what you execute for it's effect (on variables)**

The above two maxims can be converted into a computational structure as follows

A) Expression is what you evaluate for its value

```
public class Expression {

     public func evaluate( _ iContext:RuntimeContext?) ->Double {
          return 0.0
     }
}
```

B) Expression is what you evaluate for its value

```
public  class Statement{

     public  func execute( _ iContext:RuntimeContext?)->Bool{
          return false
     }
}
```

Let us implement a Print statement for the “SLANG For Swift4” compiler. The basic idea is as follows you add a class to model a statement and since the class has to inherit from the “Statement” (abstract class), it ought to implement Execute Method.

```
public class PrintStatement:Statement{

    var exp:Expression? = nil

    init(_ iExp:Expression){ self.exp = iExp }

    public func execute( _ iContext:RuntimeContext?)->Bool{

        let val  = exp?.evaluate(iContext)
        print("\(val)")

        return true
```

```
    }
}
```

Let us add a “PrintLine” statement as well. “PrintLine” implementation is not different from Print statement. The only difference is it emits a new line after the expression value.

```
public class PrintLineStatement:Statement{

    var exp:Expression?

    init(_ iExp:Expression){ self.exp = iExp }

    public func execute( _ iContext:RuntimeContext?)->Bool{

        let val  = exp?.evaluate(iContext)
        print("\(val)\n")

        return true
    }
}
```

Once we have created to classes to implement Print and “PrintLine” statement, we need to modify our parser (frontend) to support the statement in the language.

We are going to add few more tokens to support the Statements in the SLANG.

```
public enum Token{

    case illegal
    case plus
    case minus
    case times
    case divide
    case oParen
    case cParen
    case double
    case null
    case print
    case println
    case unquotedString
    case semi
}
```

In the “Lexer”, we add a new data structure to be used for Keyword lookup.

```
public struct ValueTable{
    public var token:Token = .illegal
    public var value:String = ""

    init( _ iToken:Token , _ iValue:String){

        self.token = iToken
        self.value = iValue
    }
}
```

In the Lexer class, we will populate an array of ValueTables with Token and it's textual representation as given below.

```
private var valueTables = [ValueTable(
.print,"PRINT"),ValueTable(.println,"PRINTLINE") ]
```

In the Lexer class, getToken function has to be modified as follows

```
public func getToken() throws ->Token{

        var token:Token = .illegal

        var reStart = true
        while reStart{
            reStart = false
        // Skip the white space
            while index < length && (expStr[index] == "\t" ||
expStr[index] == " "){
                index += 1
            }

            // End of the string? return null
            if index == length {
                return .null
            }

            switch expStr[index] {
                case "\n":
                    token = .illegal
                    index += 1
                    reStart = true
                    continue
                case "+":
                    token = .plus
                    index += 1
```

```
                case "-":
                    token = .minus
                    index += 1
                case "*":
                    token = .times
                    index += 1
                case "/":
                    token = .divide
                    index += 1
                case "(":
                    token = .oParen
                    index += 1
                case ")":
                    token = .cParen
                    index += 1
                case ";":
                    token = .semi
                    index += 1
                case "0","1","2","3","4","5","6","7","8","9":
                    var str:String = ""
                    while index < length && (
                                                expStr[index] == "0" ||
                                                expStr[index] == "1" ||
                                                expStr[index] == "2" ||
                                                expStr[index] == "3" ||
                                                expStr[index] == "4" ||
                                                expStr[index] == "5" ||
                                                expStr[index] == "6" ||
                                                expStr[index] == "7" ||
                                                expStr[index] == "8" ||
                                                expStr[index] == "9"
                                            ){
                        str += expStr[index]
                        index += 1
                    }
                    number = Double(str)!
                    token = .double
                default:
                    if expStr[index].isLetter {
                        var temp = expStr[index]
                        index += 1
                        while index < length &&
(expStr[index].isLetterOrDigit ||  expStr[index] == "_"){
                            temp += expStr[index]
                            index += 1
                        }
                        temp = temp.uppercased()
                        for val in valueTables{
                            if val.value == temp{
                                token = val.token
                                return token
                            }
                        }
                        lastStr = temp
```

```
                    return .unquotedString
                }else{
                    print("Error \(expStr[index])")
                    throw SlangError.illegalToken
                }
            }
        }
        return token
    }
```

We need to add a new entry point into the RDParser class to support statements. The grammar for the SLANG at this point of time (to support statement) is as given below.

<stmtlist> := { statement }+ {statement} := <printstmt> | <printlinestmt>
<printstmt> := print <expr >;
<printlinestmt>:= printline <expr>;
<Expr> ::= <Term> | Term { + | - } <Expr>
<Term> ::= <Factor> | <Factor> {*|/} <Term>
<Factor>::= <number> | (<expr>) | {+|-} <factor>

The new entry point to the parser is as follows...

```
public func parse()->[Statement]{

        getNext()
        return getStatementList()
}
```

The getStatementList method implements the grammar given above. The BNF to source code translation is very easy and without much explanation it is given below

<stmtlist> := { <statement> }+
{<statement> := <printstmt> | <printlinestmt>
<printstmt> := print <expr >
<printlinestmt>:= printline <expr>;
<Expr> ::= <Term> | <Term> { + | - } <Expr>
<Term> ::= <Factor> | <Factor> {*|/} <Term>
<Factor>::= <number> | (<expr>) | {+|-} <factor>

```
private func getStatementList()->[Statement]{

        var retStatements:[Statement] = [Statement]()
        while currentToken != .null{
            do{
                let temStmt = try getStatement()
                retStatements.append(temStmt!)
            }catch SlangError.invalidExpression{
                print(SlangError.invalidExpression.discription)
            }catch{
                print("Unknown error")
            }
        }

        return retStatements
}
```

The method getStatement queries the statement type and parses the rest of the statement.

```
private func getStatement() throws ->Statement?{

        var retVal:Statement? = nil
        switch currentToken {
            case .print:
                retVal = try parsePrintStatement()
                getNext()
            case .println:
                retVal = try parsePrintlnStatement()
                getNext()
            default:
                print("Invalid statement")
                throw SlangError.invalidExpression
       }


        return retVal
    }

private func parsePrintStatement() throws ->Statement{

        getNext()
        let exp = try getExpression()
        if currentToken != .semi{
            throw SlangError.invalidExpression
        }

        return PrintStatement(exp!)
}
```

```
private func parsePrintlnStatement() throws ->Statement{

        getNext()
        let exp = try getExpression()
        if currentToken != .semi{
            throw SlangError.invalidExpression
        }

        return PrintLineStatement(exp!)
}
```

Finally I have invoked these routines to demonstrate how everything is put together.

```
import Foundation

var str:String = "PRINTLINE 2*10;" + "\n" + "PRINTLINE 10;\n PRINT
2*10;\n"
var parser = RDParser(str)
var list = parser.parse()
for stmt in list{
    _ = stmt.execute(nil)
}

str = "PRINTLINE -2*10;" + "\n" + "PRINTLINE -10*-1;\n PRINT 2*10;\n"
parser = RDParser(str)
list = parser.parse()
for stmt in list{
    _ = stmt.execute(nil)
}
```