SLANG FOR SWIFT-4

The Art of Compiler Construction using SWIFT-4

CHAPTER – 2

INPUT Analysis

Compilers are programs, which translate source language to a target language. The Source language can be a language like C,C++ or Lisp. The potential target languages are assembly languages, object code for the microprocessors like “intel x86”, “itanium” or “power pc”. There are programs, which translate java to C++ and Lisp to C. In such case, target language is another programming language.

Any compiler has to understand the input. Once it has analyzed the input characters, it should convert the input into a form, which is suitable for further processing.
Any input has to be parsed before the object code translation. To parse means to understand. The Parsing process works as follows

The characters are grouped together to find a token (or a word). Some examples of the tokens are '+','*', while, for, if etc. The module, which reads character at a time and looks for legal token is called a lexical analyzer or “Lexer”. The input from the “Lexer” is passed into a module, which identifies whether a group of tokens form a valid expression or a statement in the program. The module that determines the validity of expressions is called a parser. Rather than doing a lexical scan for the entire input, the parser requests the next token from the lexical analyzer. They act as if they are co-routines.

To put everything together let us write a small program which acts a four function calculator The calculator is capable of evaluating mathematical expressions which contains four basic arithmetical operators, parenthesis to group the expression and unary operators.

Given below is the Lexical Specifications of the calculator.

OK_PLUS – ‘+’

TOK_MUL - ‘*’

TOK_SUB - ‘-’

TOK_DIV - '/'

TOK_OPAREN – ‘(‘

TOK_CPAREN – ‘)’

TOK_DOUBLE – [0-9]+

The above can be converted in Swift-4 as follows

```
public enum Token{

    case illegal
    case plus
    case minus
    case times
    case divide
    case oParen
    case cParen
    case double
    case null
}
```

The Lexical Analysis Algorithm scans through the input and returns the token associated with the operator. If it has found out a number, returns the token associated with the number. There should be another mechanism to retrieve the actual number identified.

Following pseudo code shows the schema of the lexical analyzer

```
while ( there is input ) {
        switch(currentchar) {
                case Operands:
                        advance input pointer
                        return TOK_XXXX;
                case Number:
                         Extract the number( Advance the input )
                        return TOK_DOUBLE;
                default: error
       }
 }
```

The following Swift-4 code is a literal translation of the above algorithm.

```
import Foundation

/*
 String extension for accessing characters as String by subscript
 */
extension String{

    subscript(index:Int) -> String{

        let chars = Array(self)
        let str:String = String(chars[index])

        return str
    }
}

public enum Token{

    case illegal
    case plus
    case minus
    case times
    case divide
    case oParen
    case cParen
    case double
    case null
}

/*
     A naive Lexical analyzer which looks for operators , Parenthesis
     and number. All numbers are treated as IEEE doubles. Only numbers
     without decimals can be entered. Feel free to modify the code
     to accomodate LONG and Double values
 */

public class Lexer{

    private var expStr:String = ""
    private var index:Int = 0
    private var length:Int = 0
    public var number:Double = 0

    init(_ iExpStr:String){
        self.expStr = iExpStr
        length = iExpStr.count
    }

    public func getToken()->Token{

        var token:Token = .illegal
```

```
        // 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 "+":
                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 "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:
                print("Error While Analyzing Tokens")
        }

        return token
    }
}
```

The Grammar

In computer science, a formal grammar (or grammar) is sets of formation rules (grammar) that describe which strings formed from the alphabet of a formal language are syntactically valid within the language. A grammar only addresses the location and manipulation of the strings of the language. It does not describe anything else about a language, such as its semantics (i.e. what the strings mean).

A context-free grammar is a grammar in which the left-hand side of each production rule consists of only a single nonterminal symbol. This restriction is non-trivial; not all languages can be generated by context-free grammars. Those that can are called context-free languages.

The "Backus Naur Form" (BNF) notation is used to specify grammars for programming languages, command line tools, file formats to name a few. The semantics of BNF is beyond the scope of this book.

Grammar of the expression evaluator

<Expr> ::= <Term> | Term { + | - } <Expr>

<Term> ::= <Factor> | <Factor> {*|/} <Term>

<Factor>::= <number> | (<expr>) | {+|-} <factor>

There are two types of tokens in any grammar specifications. They are terminal tokens (terminals) or non-terminals. In the above grammar, operators and <number> are the terminals. <Expr>, <Term>, <Factor> are non-terminals. Non-terminals will have at least one entry on the left side.

Conversion of Expression to the pseudo code

```
// <Expr> ::= <Term> { + | - } <Expr>
Void Expr() {
        Term();
        if ( Token == TOK_PLUS || Token == TOK_SUB ) {
                        // Emit instructions // and perform semantic operations
                        Expr(); // recurse
         }
}
```

Conversion of Term to the pseudo code

```
// <Term> ::= <Factor> { * | / } <Term>
Void Term() {
        Factor();
        if ( Token == TOK_MUL || Token == TOK_DIV ) {
                // Emit instructions // and perform semantic operations
                Term(); // recurse
        }
}
```

The following pseudo code demonstrates how to map <Factor> into code

```
// <Factor> ::= <TOK_DOUBLE> | ( <expr> ) | { + |- } <Factor> //
Void Factor() {
        switch(Token)
                case TOK_DOUBLE:
                        // push token to IL operand stack return
                case TOK_OPAREN:
                         Expr(); //recurse
                        // check for closing parenthesis and return
                case UNARYOP:
                        Factor(); //recurse
                default:
                        //Error
}
```

The class “RDParser” is derived from the “Lexer” class. By using an algorithm by the

name Recursive descent parsing, we will evaluate the expression. A recursive descent parser is a top-down parser built from a set of mutually-recursive procedures where each such procedure usually implements one of the production rules of the grammar. Thus the structure of the resulting program closely mirrors that of the grammar it recognizes.

```
enum SlangError:Error{

    case runtimeError
    case missingParenthessis
    case illegalToken

    var discription:String{
        get{
            switch self {
            case .runtimeError :
                return "Runtume error"
            case .missingParenthessis :
                return "Missing parenthesis error"
            case .illegalToken :
                return "Illegal token error"
            }
        }
    }
}

public class RDParser:Lexer{

    private var currentToken:Token = .illegal

    override init(_ iExpStr:String){
        super.init(iExpStr)
    }

    public func callExpression()->Expression?{

        currentToken = getToken()
        do{
            let tempExp = try getExpression()
            return tempExp
        }catch SlangError.runtimeError{
            print(SlangError.runtimeError.discription)
        }catch SlangError.missingParenthessis{
            print(SlangError.missingParenthessis.discription)
        }catch SlangError.illegalToken{
            print(SlangError.illegalToken.discription)
        }catch{
            print("Unknow error")
        }

        return nil
    }
```

```
    private func getExpression() throws ->Expression?{

        var tempToken:Token
        var retVal:Expression? = try getTerm()

        while currentToken == .plus || currentToken == .minus{
            tempToken = currentToken
            currentToken = getToken()
            let tempExp = try getExpression()
            let op = tempToken == .plus ? Operator.plus : Operator.minus
            retVal = BinaryExpression(retVal!,tempExp!,op)
        }

        return retVal
    }

    private func getTerm() throws ->Expression?{
        var tempToken:Token
        var retVal:Expression? = try getFactor()
        while currentToken == .times || currentToken == .divide{
            tempToken = currentToken
            currentToken = getToken()
            let tempExp = try getTerm()
            let op = tempToken==.times?Operator.times : Operator.divide
            retVal = BinaryExpression(retVal!,tempExp!,op)
        }

        return retVal
    }

    private func getFactor() throws -> Expression?{

        var tempToken:Token
        var retVal:Expression? = nil

        if  currentToken == .double {
            retVal = NumericConstant(self.number)
            currentToken = getToken()
        }else if currentToken == .oParen{
            currentToken = getToken()
            retVal = try getExpression()
            if currentToken != .cParen{
                print("Missing Closing Parenthesis")
                throw SlangError.missingParenthessis
            }
            currentToken = getToken()
        }else if currentToken == .plus || currentToken == .minus {
            tempToken = currentToken
            currentToken = getToken()
            retVal = try getFactor()
            let op = tempToken == .plus ? Operator.plus : Operator.minus
            retVal = UnaryExpression(retVal!,op)
        }else{
```

```
            print("Illegal Token")
            throw SlangError.illegalToken
        }

        return retVal
    }
}
```

Using the Builder Pattern, we will encapsulate the “Parser”, “Lexer” class activities

```
import Foundation

public class AbstractBuilder{

}

public class ExpressionBuilder:AbstractBuilder{

    private var expStr:String = ""

    init( _ iExpStr:String){
        self.expStr = iExpStr
    }

    public var expression:Expression?{

        get{
            let rdParser = RDParser(self.expStr)
            return rdParser.callExpression()
        }
    }
}
```

The expression compiler is invoked as follows

```
let expBldr = ExpressionBuilder("-2*(3+3)")
let exp = expBldr.expression
print("\(exp?.evaluate(nil) ?? 0)")
```