SLANG FOR SWIFT-4

The Art of Compiler Construction using SWIFT-4

CHAPTER – 1

Compilers are some of the most fascinating programs because they touch every aspect of Computer Science, from CLI and API design to calling conventions and platform-specific optimizations. Yet, many developers have a fearful reverence for them that makes them seem unapproachable. I think this is unnecessary.

Difficulties in compiler development arise mainly because of human concerns: semantic correctness and following a strict standard. Those concerns are easier to satisfy when designing a new language, as the standard is usually malleable.

Thanks to the availability of information and better tools writing a compiler has become just an exercise in software engineering. The Compilers are not difficult programs to write. The various phases of compilers are easy to understand in an independent manner. The relationship is not purely sequential. It takes some time to put phases in perspective in the job of compilation of programs.

The task of writing a compiler can be viewed in a top down fashion as follows

Parsing => Creation of Abstract Syntax Tree => Tree Traversal to generate the Object code or Recursive interpretation.

Abstract Syntax Tree

In computer science, an abstract syntax tree (AST), or just syntax tree, is a tree representation of the abstract (simplified) syntactic structure of source code written in a certain programming language. Each node of the tree denotes a construct occurring in the source code. The syntax is abstract in the sense that it does not represent every detail that appears in the real syntax. For instance, grouping parentheses is implicit in the tree structure, and a syntactic construct such as “ if ‘cond’ then ‘expr’ ” may be denoted by a single node with two branches. Most of you might not be aware of the facts that, programming languages are hierarchical in nature. We can model programming language constructs as classes. Trees are a natural data structure to represent most things hierarchical.

As a case in the point, let us look a simple expression evaluator. The expression evaluator will support double precision floating point value as the operands. The Operators supported are addition (+), subtraction (-), multiplication (*) and division. The Object model support Unary operators (+, -) as well. We are planning to use a composition model for modeling an expression.

In most imperative programming languages, an expression is something which u evaluate for its value. Where as statements are something which you executes for it's effect.

Let us design a class for Expression

```
class Expression{
    init(){}
    func evaluate( _ iContext:RuntimeContext?){ }
}
```

For the time being the ‘RuntimeContext’ is an empty class

```
class RuntimeContext {
    public init() {}
}
```

Modeling Expression

Once you have declared the interface and it's parameters, we can create a hierarchy of classes to model an expression.

class Expression // Base class for Expression

class NumericConstant // Numeric Value

class Binary Expression // Binary Expression

class Unary Expression // Unary Expression

Take a look at the listing of NumericConstant class

```
class NumericExpression:Expression{

    private var value:Double = 0

    init( _ iValue:Double){
        super.init()
        value = iValue
    }

    override func evaluate( _ iContext:RuntimeContext?)->Double {

        return value
    }
}
```

Since the class is derived from Expression, it ought to implement the evaluate method. In the Numeric Constant node, we will store a IEEE 754 double precision value. While evaluating the tree, the node will return the value stored inside the object.

Binary Expression

In a Binary Expression, one will have two Operands (Which are themselves expressions of arbitrary complexity) and an Operator.

```
class BinaryExpression:Expression{

    private var exp1:Expression?
    private var exp2:Expression?
    private var op :Operator?

    init(_ iExp1:Expression , _ iExp2:Expression, _ iOp:Operator){

        self.exp1 = iExp1
        self.exp2 = iExp2
        self.op = iOp
    }

    override func evaluate( _ iContext:RuntimeContext?)->Double {

        switch self.op! {
          case .plus:
            return exp1!.evaluate(iContext) + exp2!.evaluate(iContext)
          case .minus:
            return exp1!.evaluate(iContext) - exp2!.evaluate(iContext)
          case .times:
            return exp1!.evaluate(iContext) * exp2!.evaluate(iContext)
          case .divide:
            return exp1!.evaluate(iContext) / exp2!.evaluate(iContext)
        }
    }
}
```

Unary Expression

In a unary expression, one will have an Operand (which can be an expression of arbitrary complexity) and an Operator, which can be applied on the Operand.

```
public class UnaryExpression:Expression{

    private var exp:Expression?
    private var op :Operator?

    init(_ iExp:Expression , _ iOp:Operator){

        self.exp = iExp
        self.op = iOp
    }

    override func evaluate( _ iContext:RuntimeContext?)->Double {

        switch self.op! {
            case .plus:
                return exp!.evaluate(iContext)
            case .minus:
                return -(exp!.evaluate(iContext))
            default:
                return exp!.evaluate(iContext)
        }
    }
}
```

We will include the LLVM IR Generator before composing the expression.

```
var exp:Expression = BinaryExpression(NumericConstant(5) ,
NumericConstant(10), .plus)
print("\(exp.evaluate(nil))")

exp = UnaryExpression( BinaryExpression(exp , NumericConstant(20) ,
.times ) , .minus)
print("\(exp.evaluate(nil))")
```