**SLANG FOR SWIFT-4**

The Art of Compiler Construction using SWIFT-4

**CHAPTER – 5**

**Compilation**

Till now, the Slang Compiler was interpreting the statements and expressions. In this step we will try to compile the source into LLVM IR (LLVM Intermediate Representation)

In this step, we are not going to make any change into the SLANG grammar. With Future (support for Logical Expression, Relational Expression, Control structures and Procedures ) in mind , we will re-factor the code.

LLVM IR is a high-level assembly language with a suite of backend that can generate machine code for many different target architectures. High-level assembly language, while oxymoronic, means that LLVM is semantically similar to assembly while adding higher-level abstractions. These abstractions make it possible to generate code without having to consider things like calling conventions, nuances of pointer arithmetic, register allocation, and stack management.

IR is also strongly typed, which means you can think about structured types and don't have to worry much about manipulating raw data and thinking in terms of bits and bytes.

**Generating IR in Swift**

Almost all of the LLVM project is implemented in C++, including all the libraries and APIs. Unfortunately, Swift does not interoperate directly with C++ like it does Objective-C. Because many more languages can interoperate with C than C++, the LLVM project includes a comprehensive set of C bindings into the LLVM libraries.

However, as with any C interop, it's cumbersome and easy to introduce subtle bugs from Swift. Using the C API, you frequently have to make use of the UnsafeMutableBufferPointer APIs, which (while convenient) are less than ideal. So that Slang For Swift-4 is using a third-party library that wraps the LLVM API in a much more friendly, Swifty interface. It does away with all the unsafe code and exposes native Swift types that represent the underlying LLVM types. It's called LLVMSwift (https://github.com/llvm-swift/LLVMSwift)

*To give a quick feel of what is involved*

Let's say we want to emit Hello, World as LLVM IR.

First, we want to create a Module. Think of an LLVM module as everything that would reside in a single .o file from clang or gcc. It includes all the function declarations, global variables, and metadata. You can give it whatever name you'd like.

**import** **LLVM**

**let** module **=** **Module**(name: "main")

Next, we'll want to make an IRBuilder that will handle generating all the individual IR statements, functions, and global variables. A builder builds code into a specific Module, so you need to create it with the particular module you just created.

**let** mainType **=** **FunctionType**(args: [],returnType: **VoidType**())

**let** main **=** builder**.addFunction**("main", type: mainType)

*// declare void @main()*

In this case, main is a reference to an LLVM Function. You can pass this Functionobject into the IRBuilder's buildCall method to build a call instruction.

For detailed explanation about LLVMSwift : <https://harlanhaskins.com/2017/01/11/building-a-compiler-with-swift-in-llvm-part-3-code-generation-to-llvm-ir.html>

How to include ‘LLVMSwift’ in ‘SLANG For Swift4’

* Install LLVM 6.0+ using brew : brew install llvm
* Check-out LLVMSwift from : <https://github.com/llvm-swift/LLVMSwift.git>
* Add the files in ‘Sources/’ to your SLANG for Swift4 XCode project
* Under ‘Library Search Paths’ add the output of ‘llvm-config –libdir’
  + /usr/local/Cellar/llvm/6.0.0/lib
* Under Header Search Paths add the output of llvm-config –includedir
  + /usr/local/Cellar/llvm/6.0.0/include
* Under Link Target with Libraries drag in /path/to/your/llvm/6.0.0/lib/libLLVM.dylib
  + /usr/local/Cellar/llvm/6.0.0/lib

Now you good to go