Compilers-II (CS3423)

Mini Assignment - I

An Introduction to the LLVM Infrastructure, AST, IR and Compiler Options

Sagar Jain
CS17BTECH11034

August 27, 2019

Contents

Clang AST

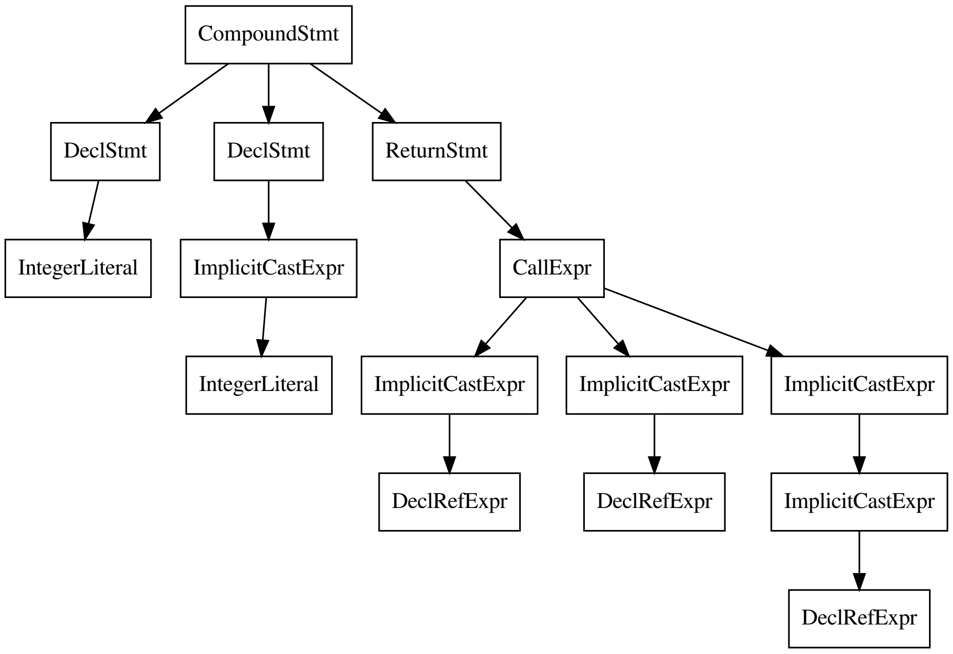
Clang is essentially a library that allows us to convert a C program into an Abstract Syntax Tree and then manipulte the tree in some ways. After viewing the ast for several non-trivial programs the following are a few observations:

1. The ast for any program can be printed using:
   `clang -Xclang -ast-dump -fsyntax-only <filename>`. Here `Xclang` option is used to pass the arguments to the clang compiler, `fsyntax-only` is used so that no object code is generated and `ast-dump` is used to print the abstract syntax tree.

2. The dump begins with a bunch of `TypedefDecl` nodes followed by the ast for the user code.

3. We can use the option `ast-list` to get all a list of all the *Declaration* nodes, this is an alternative option to look at the contents of the ast.

4. The `ast-view` option lets us look at an the entire ast graphically and is very easy to understand compared to the dump.Internally it creates a dot file which can be viewed by any dot file renderer. Example:



5. We can also use clang-check to filter and print only the ast of a subset of the Declaration nodes:
   `clang-check <filename> -ast-dump -ast-dump-filter=main --`

Observations about the AST structure

- In the tree that clang creates, every node is an instance of one of the two classes: **Decl** or **Stmt** class. Nodes usually have a name followed by a type. For example, the main function begins with a `FunctionDecl` node which would have a name *main* and a type(say `int` or `void`).

- Functions start with `FunctionDecl` node which has the function name and type along with it. These are always followed by the `ParamVarDecl` nodes which have information about the function parameters and their types.
- Most of the nodes in the body of the functions belongs to the Stmt class in the form of either of the following:
  - CompoundStmt
  - DeclStmt
  - ImplicitCastExpr
  - CallExpr
  - ReturnStmt
  - Literals, etc.
- Nodes like `ForStmt` have multiple children (5 for ForStmt) which are themselves nodes of different kind.
- Nodes like `IntegerLiteral` contain the value of the node as well along with the type.

Clang AST Traversal

The observations made about the AST traversal follows from this tutorial.

FrontendAction

The class ASTFrontendAction is provided to us to be able to interact with the ast while compilation. We must create a class which inherits from FrontendAction to be able to execute user specific actions on the ast. We must implement a virtual method `CreateASTConsumer` in this class which returns a type of `std::unique_ptr<clang::ASTConsumer>`, what we return is essentially our implementation of how we would like to consume the ast provided to us.

ASTConsumer

The clang class `ASTConsumer` is used to create the consumers for the ast. We must create a class that inherits from this class to be able to use the ast. We can have multiple different ways to enter the ast for example `HandleInlineFunctionDefinition`, `HandleTranslationUnit`, `HandleTopLevelDecl`, etc. We can override any of these methods to read the ast. This class must also have an implementation of RecursiveASTVisitor as its member, this implementation is defined in the following section. We can provide our consumer with `&Compiler.getASTContext()` to give it information like source locations which are not stored in the ast nodes themselves.

RecursiveASTVisitor

We must implement a class which inherits from RecursiveASTVisitor to be used by the consumer class during the traversal. In this class we have methods of the type `VisitNodeType(NodeType *)` for different types of ast nodes. These methods return bool where *true* implies that we want to continue the traversal and *false* stops the traversal.

LLVM Error Messages

Programmatic Errors

The observations on error messages have been made from here. In LLVM all the errors are classified into two types; *programmatic* and *recoverable*. Assertions are used heavily in the llvm-project. An assertion can be made to check if any code invariant or condition is being broken.
For example:
In `clang/lib/Analysis/CallGraph.cpp` we can find the following:

```
    assert(*CI != Root && "No one can call the root node.")
```

We know that in a cfg there can be no calls to the root node, this assert is placed in the code to ensure this condition and also holds a message so that the user can know why the program has stopped execution.

Recoverable Errors

Recoverable errors are mostly the errors which occur because of reasons other than mistakes in the source code. These errors should be reported to the user as well. Reported errors are handled using the error scheme provided by LLVM. The error class can be used for any user defined errors and we can also specify information regarding the error in it. Template functions like `make_error` are provided to construct failure values for the error class created by us which inherit from `ErrorInfo`.

LLVM IR

Assembly Language

Compiler Toolchain

Kaleidoscope