Clang AST

Nikhil Hegde

Compiler Optimizations course @ QUALCOMM India Pvt. Ltd.

Nikhil D. Hegde, Assistant Professor, Computer Science and Engineering

Qualification:
Ph.D. Purdue University, USA. (2019).
M.Tech. Indian Institute of Technology Madras (2005).
B.E. B.M.S. College of Engineering, Bangalore (2002).

Industry Experience: **9 years** Developing hardware, middleware, and software for mobile platforms @ {Intel, ST Microelectronics, Nokia, AdsFLO}-India Pvt. Ltd., & Infosys Ltd.

Teaching: Software Construction, Compilers, Parallel and High-Performance Computing, Scientific Computing, and Advanced C Programming.

Research Area: Parallel and Distributed Computing, Scientific Computing, and Programming Languages

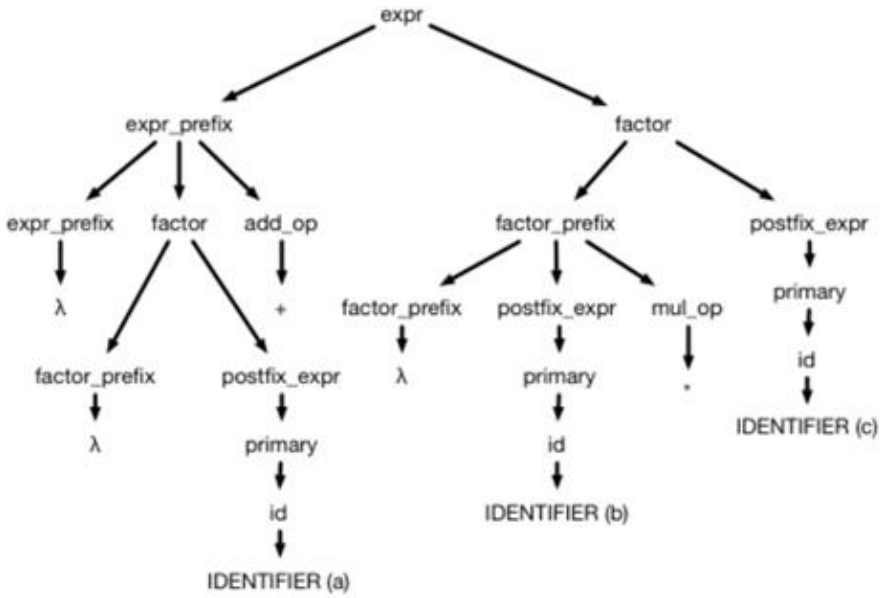
Research Summary: Optimizing `irregular` applications, Automatic code generation for massively parallel systems

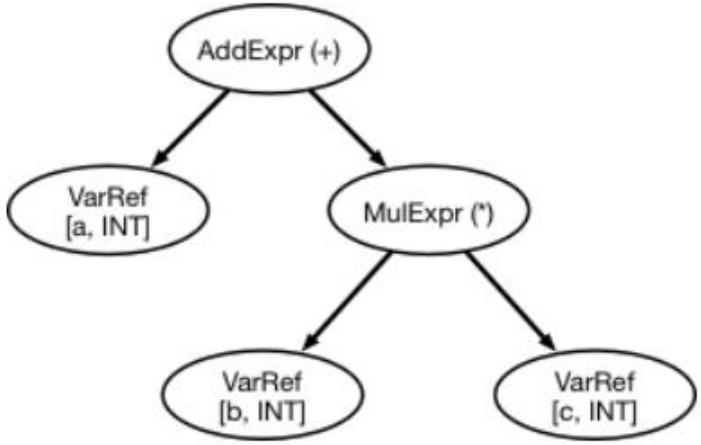
E-mail: nikhilh@iitdh.ac.in

Clang

- C language family *frontend* for LLVM
- One of the responsibilities of the frontend is:
  - Reading a program text and producing an AST (abstract syntax tree), which is a representation of the program.

Parse Tree vs. AST (a*b+c)





- Clang AST – enables you to develop tools
  - for analysing program text

Clang Usage

- As Clang Plugin
  - E.g. `clang -fpass-plugin=mypass.so test.c`

  (a silly use case: replacing + operator in `test.c` with *)
- LibTooling
  - A c++ program (with main function) written to do some analysis of some other program (C, C++)
  - E.g. `bin/myanalyzer test.c`
- LibClang
  - E.g., if We want to write a python program that analyses C++ source code

Clang AST

- Is rich (has source code location info at every node)
- Has types resolved
  - A design choice of compiler construction is that type checking can be done separately from AST construction
- Is huge
  - ~10K .cpp files
  - > 100k lines of code
- Has several classes, objects of which form the AST nodes.
  - The AST nodes are optimized for size.

Clang AST Classes

- `ASTContext` – one of the first classes that we encounter

  AST nodes point to e.g.:

  Identifier Table - storing identifiers

  Source Manager – managing source code locations

  - organizes information around the AST

  - provides entry point to the AST with the help of `TranslationUnitDecl* getTranslationUnitDecl()`

Clang AST Core Classes

- `Decl`
  - Base class for many other classes e.g. `VarDecl`

  `int x=100;` //int x corresponds to a VarDecl node

  exercise: what type of node represents 100? `IntegerLiteral`

  Other examples: `CXXRecordDecl`
- `Stmt`
  - `CompoundStmt`

    No common base class

    `ReturnStmt -> e.g., return x;`
  - `BinaryOperator ->` e.g., `i > 0, x + 2`
    - Is an expression (`Expr`) and expressions are `Stmts`
- `Type`
  - `PointerType`
  - `ParenType`

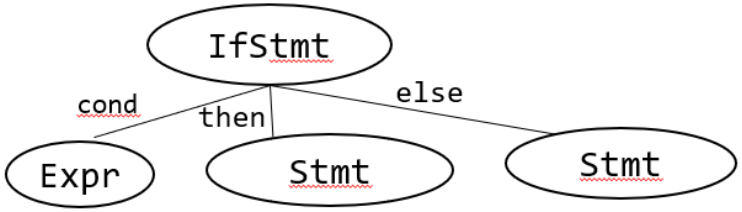
Clang AST Glue Classes

- `DeclContext`
  - Inherited by `Decl`s that contain other `Decl`s
- `TemplateArgument`
  - To represent template arguments
- `NestedNameSpecifier`
- `QualType`
  - For representing type qualifiers e.g. `const, unsigned` etc.

Clang AST Glue Methods

Help in traversing AST

- `IfStmt : getThen(), getElse(), getCond()`



Traverse across parts of AST

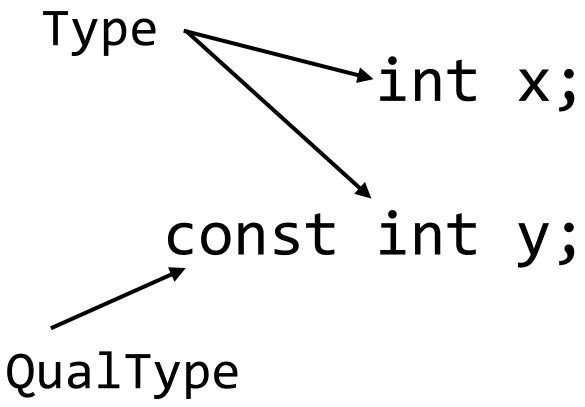
- `CXXRecordDecl: getDescribedClassTemplate() method`

- `Type: getAsCXXRecordDecl() method`

For getting the declaration of a struct/union/class

Types

- Represent types in AST

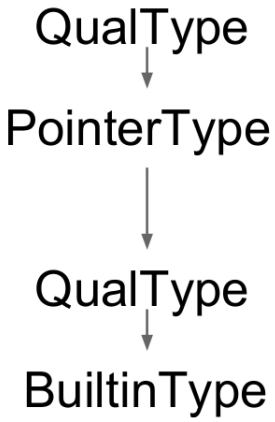



```
int * const * y;
```

Exercise: what is the type of y?

Types can be nested. Qualifier is sandwiched.

```
Constant pointer to a int *
```




```
class PointerType{
   QualType getPointeeType() const;
}
```

Navigating Source

- `SourceLocation` – tells the location of a token

AST Traversal

- Multiple abstractions available:
  - `RecursiveASTVisitor`
  - `ASTMatcher`

  - `Next: RecursiveASTVisitor`

ASTFrontEndAction

```
class FindNamedClassAction : public clang::ASTFrontendAction {
public:
  virtual std::unique_ptr<clang::ASTConsumer> CreateASTConsumer(
    clang::CompilerInstance &Compiler, llvm::StringRef InFile) {
    return
std::make_unique<FindNamedClassConsumer>(&Compiler.getASTContext());
  }
};
```

- Inherit from `ASTFrontendAction` and override `CreateASTConsumer`
  - `ASTFrontendAction` is an interface that allows user-specific actions to happen during compilation.
- `CreateASTConsumer`
  - Consumes the AST produced by the Clang parser

ASTConsumer

```
class FindNamedClassConsumer : public clang::ASTConsumer {
public:
  explicit FindNamedClassConsumer(ASTContext *Context)
    : visitor(Context) {}

  virtual void HandleTranslationUnit(clang::ASTContext &Context) {
    visitor.TraverseDecl(Context.getTranslationUnitDecl());
  }
private:
  FindNamedClassVisitor visitor;
};
```

- Override (as many) methods to take user-specific action on visiting AST nodes.
- `HandleTranslationUnit` called after entire source code is parsed (not while it is being parsed)
  - `ASTContext` class represents AST for the source file.
  - `Visitor.TraverseDecl(Context.getTranslationUnitDecl())` begins visiting nodes of the tree

RecursiveASTVisitor

```
class FindNamedClassVisitor
  : public RecursiveASTVisitor<FindNamedClassVisitor> {
public:
  bool VisitCXXRecordDecl(CXXRecordDecl *Declaration) {
    //for illustration only. Dump shows which nodes already visited.
    Declaration->dump();

    // The return value indicates whether we want the traversal to proceed.
    // Return false to stop the traversal of the AST.
    return true;
  }
};
```

- Don’t call Visit functions directly
- Implement `VisitStmt, VisitDecl, VisitPantry etc.` as per your needs

Pattern Matching and Source Location

```
if (Declaration->getQualifiedNameAsString() == "n::m::C") {
      FullSourceLoc FullLocation = Context->getFullLoc(Declaration-
>getBeginLoc());
      if (FullLocation.isValid())
        llvm::outs() << "Found declaration at "
                     << FullLocation.getSpellingLineNumber() << ":"
                     << FullLocation.getSpellingColumnNumber() << "\n";
}
```

- Get the location manager from ASTContext

Demo

- Environment setup
  - In-tree build
  - Linking with installed libraries built from source

```
1. clone llvm-project from github and cd llvm-project
2. mkdir build && cd build
3. cmake -G Ninja ../llvm -DLLVM_ENABLE_PROJECTS="clang;clang-tools-extra" \
4. -DCMAKE_BUILD_TYPE=Release -DLLVM_BUILD_TESTS=ON


5. mkdir ../clang-tools-extra/find-class-decls
6. echo 'add_subdirectory(find-class-decls)' >> ../clang-tools-
   extra/CMakeLists.txt
7. vim clang-tools-extra/find-class-decls/CMakeLists.txt
8. add cpp files to ../clang-tools-extra/find-class-decls

9. type ninja (from the build directory) to build your program analyzer
```

Try it yourself

- Discover all implementations of variadic function definitions in the source code (single file) and print the names of those functions

SSA

- It would be convenient if variable names were unique on the LHS of assignment.
- However, mutation is fact of life so when programmers write programs we can't expect them to use uniq names on LHS and variables get reassigned.
- So how do we convert programs to SSA form? SSA - static single assignment. Every variable, statically, in the program text has exactly one assignment.
- Phi nodes – special instructions that help deal with control flow.
- LLVM has an IR that is in SSA form. It has APIs that allow you to construct such a IR.

Acknowledgements

- https://clang.llvm.org/docs/ (clang documentation)
- https://clang.llvm.org/docs/IntroductionToTheClangAST.html (Introduction to the Clang AST) and slides from Manuel Klimek

Appendix - Session 1, Clang AST, 23/4/2025

How to use the code provided?

`find-class-decls` contains the source code. After completing steps 1 to 4 mentioned in slide (Demo), copy this directory inside the `llvm/clang-tools-extra` directory. Now you can skip steps 5, 7, and 8 mentioned in the slide (Demo). Execute step 6 followed by step 9.

Once step 9 is successful, execute:

```
./bin/find-class-decls “namespace n { namespace m {class C {}; }}”
```

To refer to the class C, one would use `n::m::C` and this is the pattern that we are trying to match. In the input argument to `find-class-decls` we provide a code snippet with matching pattern and hence, we see print statement on the terminal.

To view the AST corresponding to any C program, say in `test.c`:

```
clang -Xclang -ast-dump -fsyntax-only test.c
```

In the above command, what follows `-Xclang` are arguments to Clang parser. `-ast-dump` is telling to dump the AST. `-fsyntax-only` is telling Clang to stop processing at semantic processing stage.

Note that the C program in `test.c` need not have a main function. If you were to omit `-fsyntax-only` you would see a linker error.

Reference:

https://clang.llvm.org/docs/LibASTMatchersTutorial.html