基于Clang的LLVM前端简析

中科院软件所PLCT实验室 史宁宁

2023-07-15

Contents

- Structure of Clang
- FrontendAction
- Preprocessor && Lexer
- Parser
- Sema
- CodeGen
- Clang Tools

High-level structure of Clang

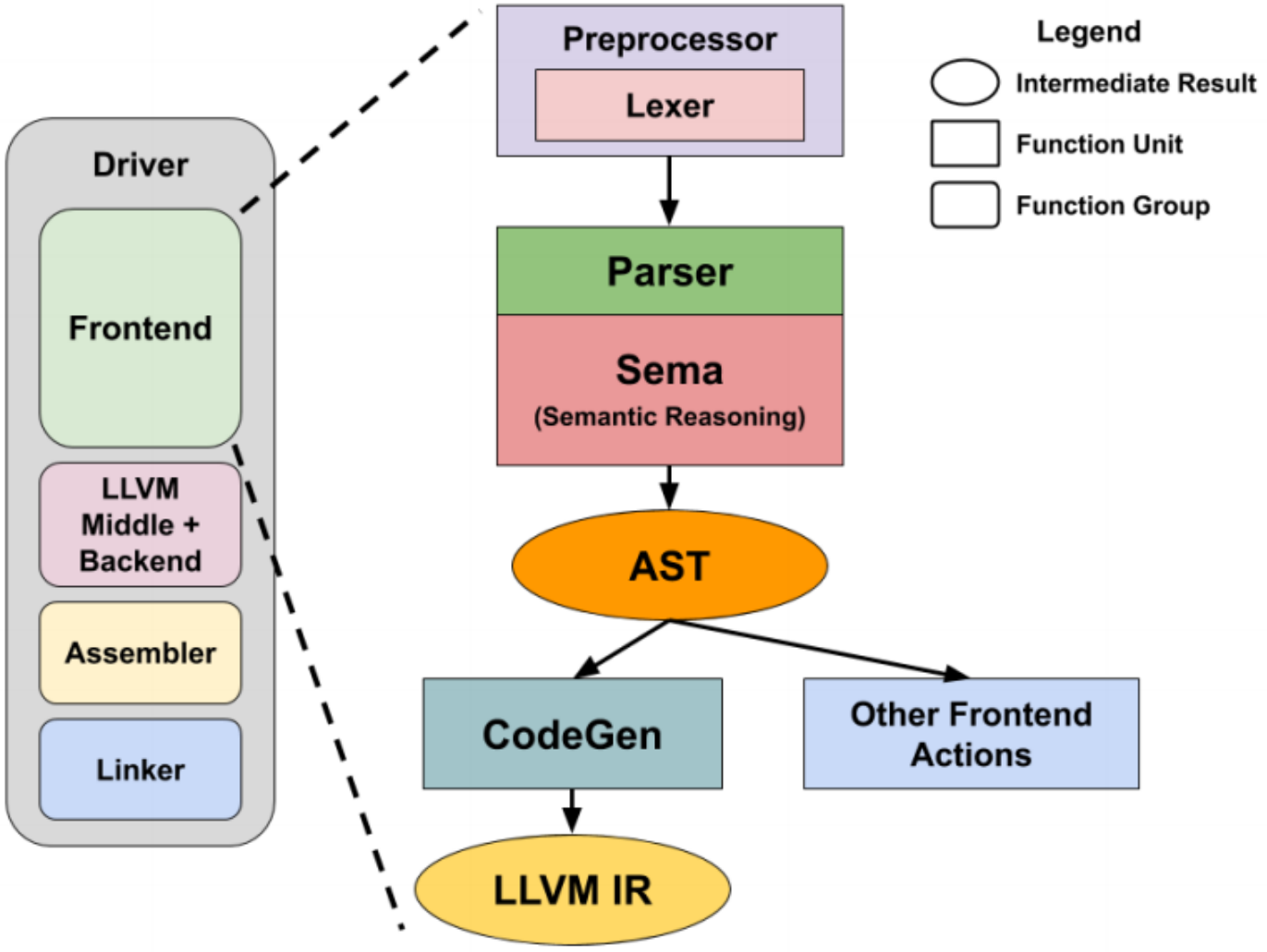


Figure 5.1 – High-level structure of Clang

From: <LLVM Techniques, Tips, and Best Practices – Clang and Middle-End Libraries> P75

Contents

- Structure of Clang
- FrontendAction
- Preprocessor && Lexer
- Parser
- Sema
- CodeGen
- Clang Tools

FrontendAction

- Abstract base class for actions which can be performed by the frontend.[1]
- 源代码:

llvm/clang/include/clang/Frontend/FrontendAction.h

llvm/clang/lib/ Frontend/FrontendAction.cpp

- 子类:

clang::ASTFrontendAction

clang::PreprocessorFrontend

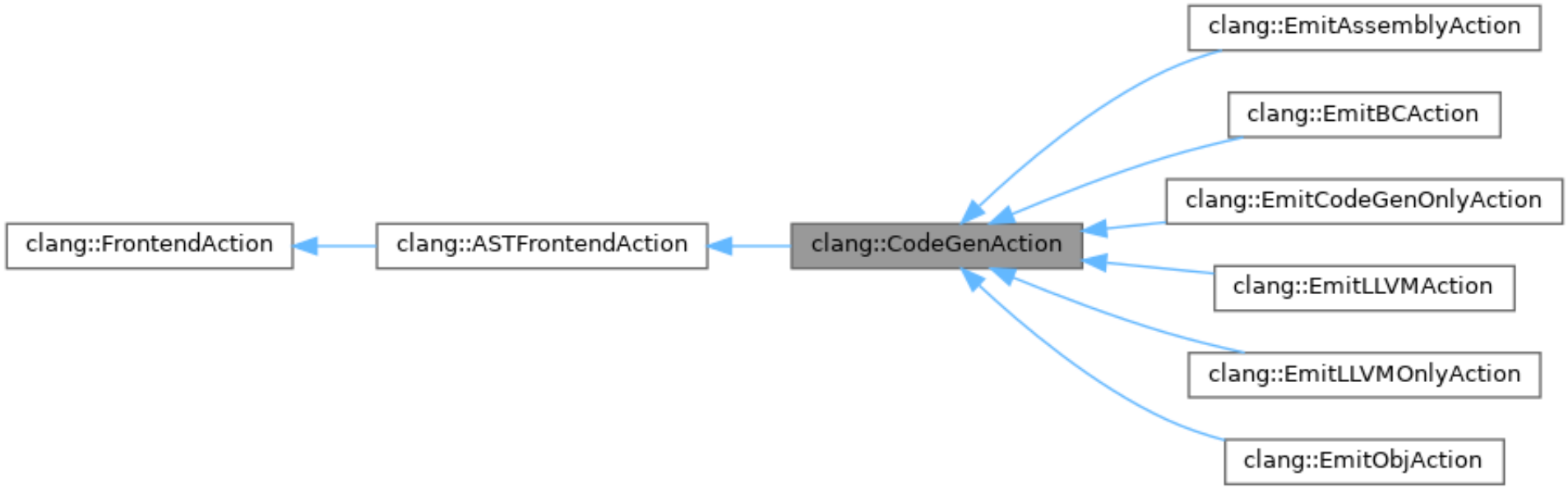
[1]. https://clang.llvm.org/doxygen/classclang_1_1FrontendAction.html

CodeGenAction

- Create a new code generation action.[1]

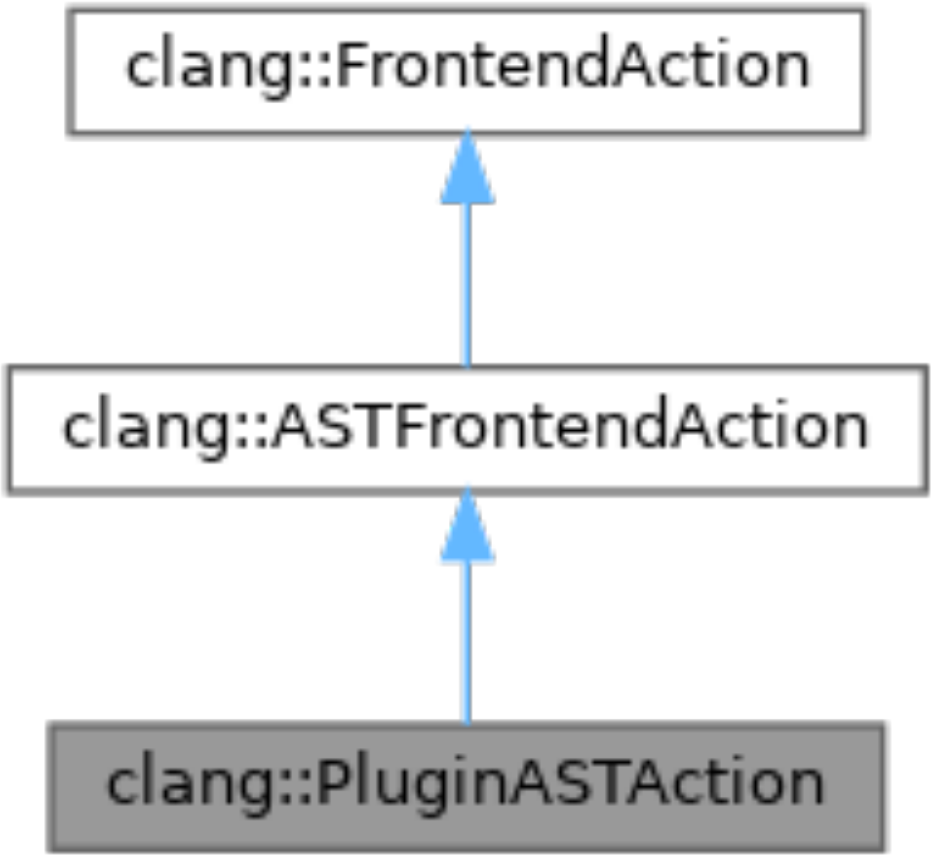
```
#include "clang/CodeGen/CodeGenAction.h"
```

Inheritance diagram for clang::CodeGenAction:



From: https://clang.llvm.org/doxygen/classclang_1_1CodeGenAction.html

PluginASTAction



From: https://clang.llvm.org/doxygen/classclang_1_1PluginASTAction.html

Contents

- Structure of Clang
- FrontendAction
- Preprocessor && Lexer
- Parser
- Sema
- CodeGen
- Clang Tools

Preprocessor and Lexer

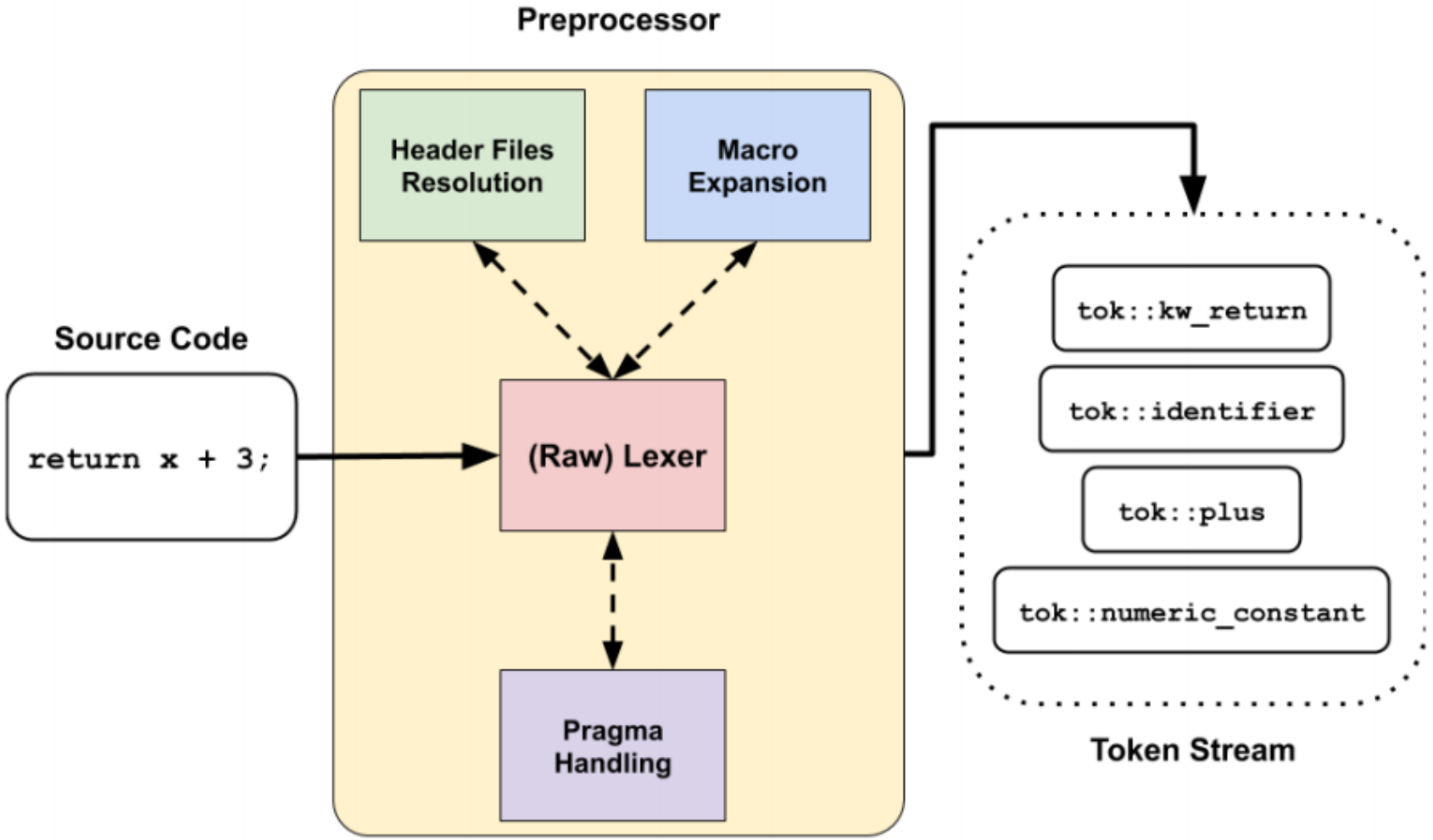


Figure 6.1 – Role of the Clang preprocessor and lexer

From: <LLVM Techniques, Tips, and Best Practices – Clang and Middle-End Libraries> P92

Contents

- Structure of Clang
- FrontendAction
- Preprocessor && Lexer
- Parser
- Sema
- CodeGen
- Clang Tools

Abstract Syntax Tree(AST)

- The result of the parsing process is an AST. The AST is another compact representation of the input program. [1]
- The AST is constructed during parsing. The semantic analysis checks that the tree adheres to the meaning of the language (for example, that used variables are declared) and possibly augments the tree. After that, the tree is used for code generation. [1]

[1]. <Learn LLVM 12> P58

TranslationUnitDecl

- This class represents an input source file, also called a translation unit (most of the time). It contains all the top-level declarations – global variables, classes, and functions, to name a few – as its children, where each of those top-level declarations has its own subtree that recursively defines the rest of the AST.[1]

[1]. <LLVM Techniques, Tips, and Best Practices – Clang and Middle-End Libraries> P113

ASTContext

- As its name suggests, this class keeps track of all the AST nodes and other metadata from the input source files. If there are multiple input source files, each of them gets its own TranslationUnitDecl, but they all share the same ASTContext.[1]

[1]. <LLVM Techniques, Tips, and Best Practices – Clang and Middle-End Libraries> P113

AST Nodes

- The AST nodes – can be further classified into three primary categories: declaration, statement, and expression. The nodes in these categories are represented by subclasses derived from the Decl, Expr, and Stmt classes.[1]

[1]. <LLVM Techniques, Tips, and Best Practices – Clang and Middle-End Libraries> P113

AST Nodes -- Declarations

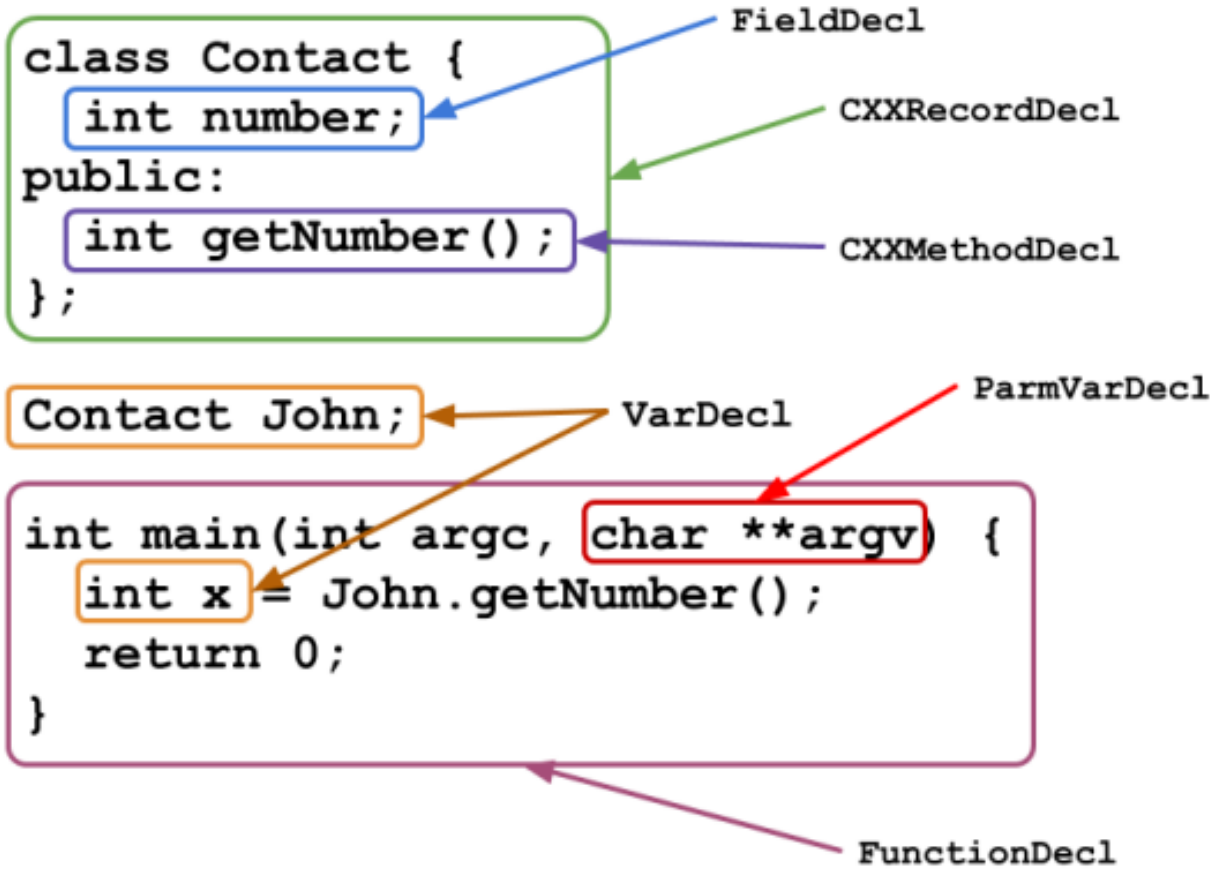


Figure 7.1 – Common declarations in C/C++ and their AST classes

[1]. <LLVM Techniques, Tips, and Best Practices – Clang and Middle-End Libraries> P114

AST Nodes -- Statements

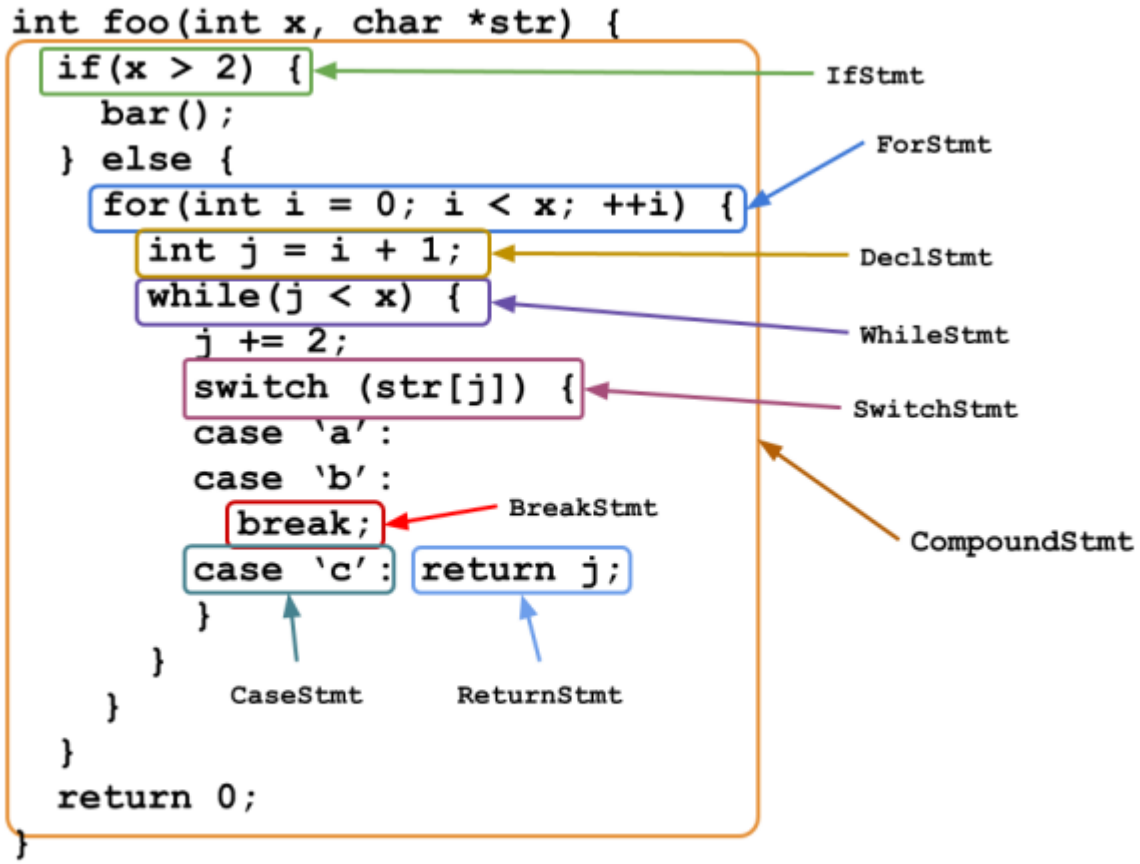


Figure 7.2 – Common statements (excluding expressions) in C/C++ and their AST classes

[1]. <LLVM Techniques, Tips, and Best Practices – Clang and Middle-End Libraries> P115

AST Nodes -- Expressions

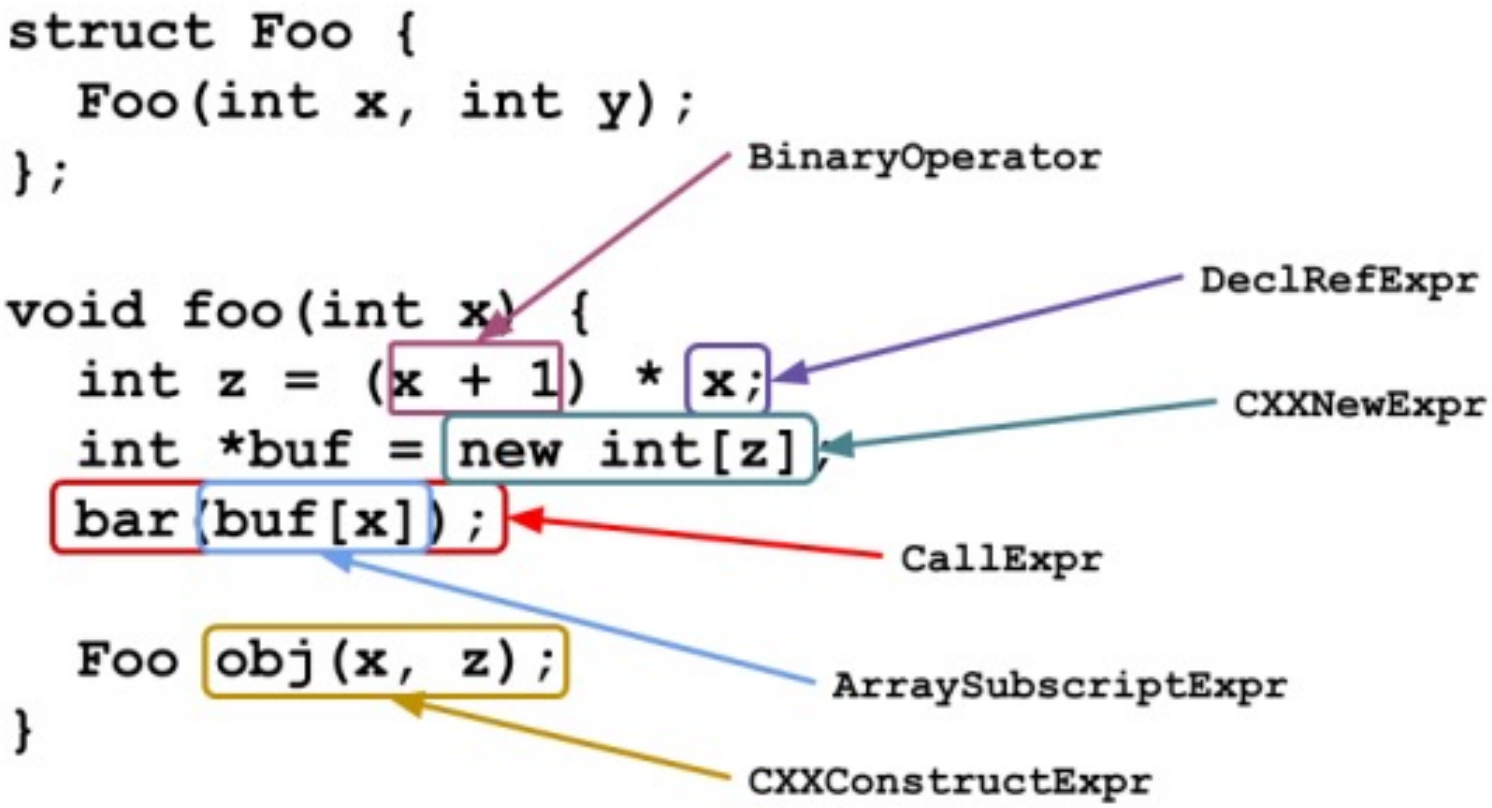


Figure 7.3 – Common expressions in C/C++ and their AST classes

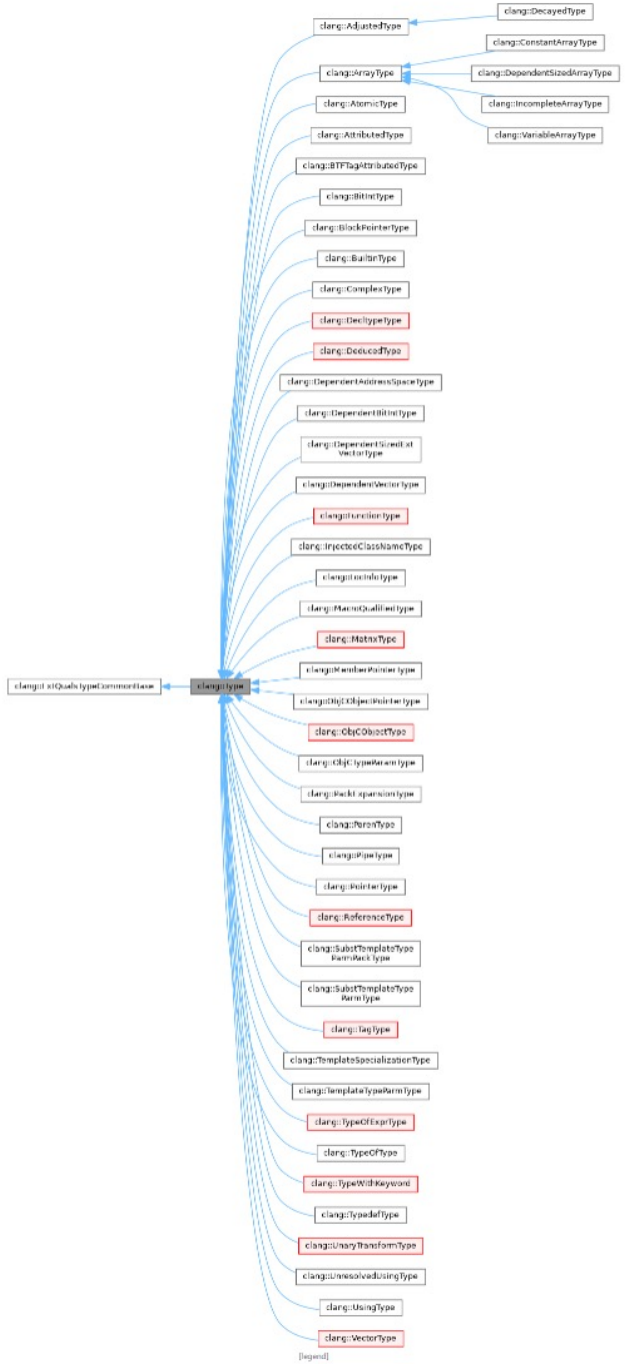
[1]. <LLVM Techniques, Tips, and Best Practices – Clang and Middle-End Libraries> P116

AST’s Type System 1

- The core of Clang AST's type system is the clang::Type class.
- Each source code type is actually represented by a subclass of Type.

From: <LLVM Techniques, Tips, and Best Practices – Clang and Middle-End Libraries> P117

AST’s Type System 2



From:
https://clang.llvm.org/doxygen/classclang_1_1Type.html

AST’s Type System 3

- BuiltinType: For primitive types such as int, char, and float.
- PointerType: For all the pointer types. It has a function called PointerType::getPointee() for retrieving the source code type being pointed to by it.
- ArrayType: For all the array types. Note that it has other subclasses for more specific arrays that have either a constant or variable length.
- RecordType: For struct/class/union types. It has a function called RecordType::getDecl() for retrieving the underlying RecordDecl.
- FunctionType: For representing a function's signature; that is, a function's argument types and return type (and other properties, such as its calling convention).

From: <LLVM Techniques, Tips, and Best Practices – Clang and Middle-End Libraries> P117

ASTMatcher

- ASTMatcher is the utility that helps you write AST pattern matching logic via a clean, concise, and efficient Domain-Specific Language (DSL).[1]
- There are three different basic categories of matchers:
  - Node Matchers: Matchers that match a specific type of AST node.
  - Narrowing Matchers: Matchers that match attributes on AST nodes.
  - Traversal Matchers: Matchers that allow traversal between AST nodes.[2]
- Matching the Clang AST https://clang.llvm.org/docs/LibASTMatchers.html

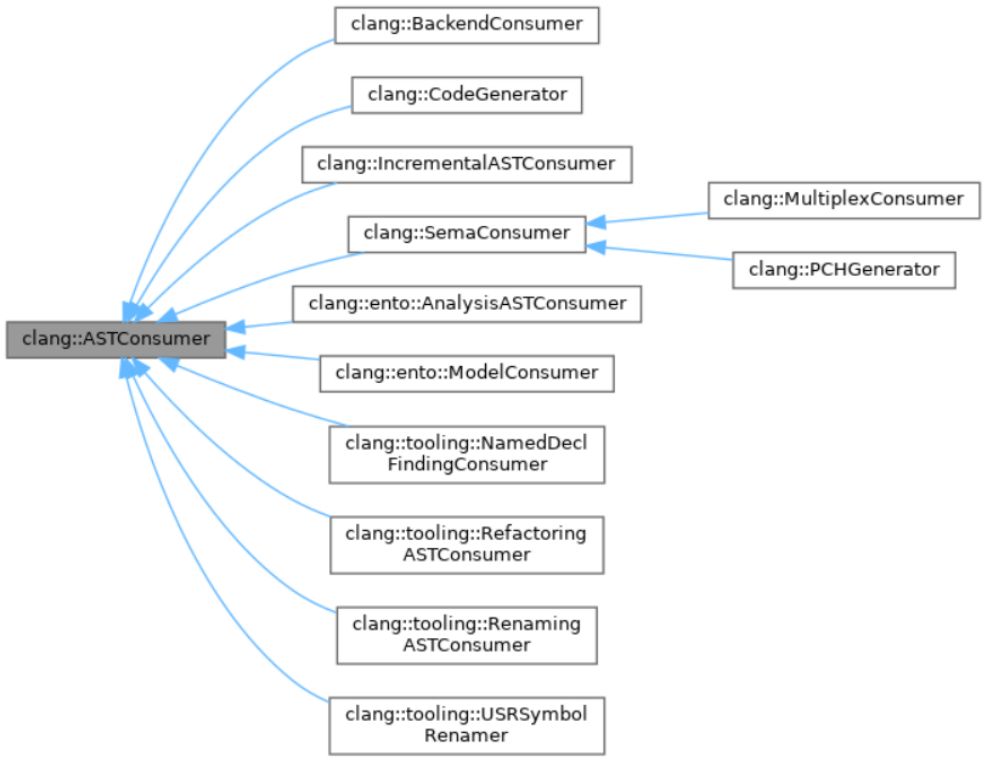
[1]. <LLVM Techniques, Tips, and Best Practices – Clang and Middle-End Libraries> P119
[2]. AST Matcher Reference https://clang.llvm.org/docs/LibASTMatchersReference.html

ASTConsumer

- This is an abstract interface that should be implemented by clients that read ASTs.

#include "clang/AST/ASTConsumer.h"

Inheritance diagram for clang::ASTConsumer:



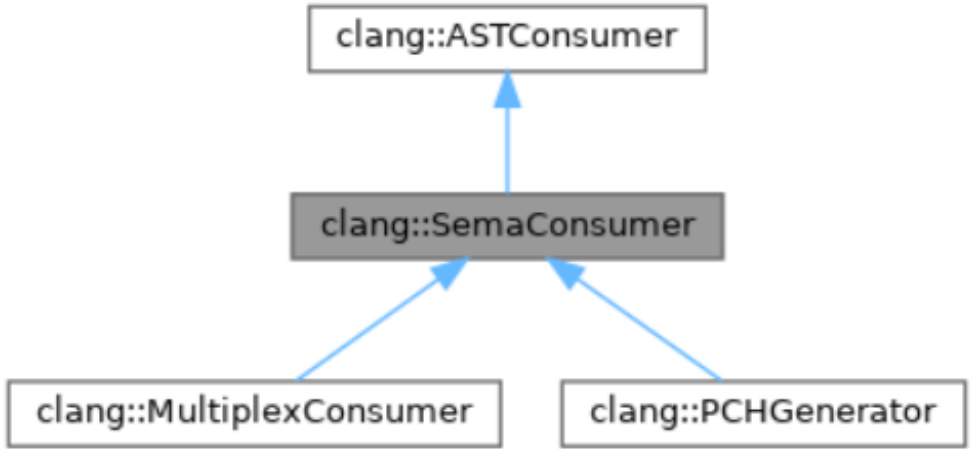
From:
https://clang.llvm.org/doxygen/classclang_1_1ASTConsumer.html

SemaConsumer

- An abstract interface that should be implemented by clients that read ASTs and then require further semantic analysis of the entities in those ASTs.

```
#include "clang/Sema/SemaConsumer.h"
```

Inheritance diagram for clang::SemaConsumer:



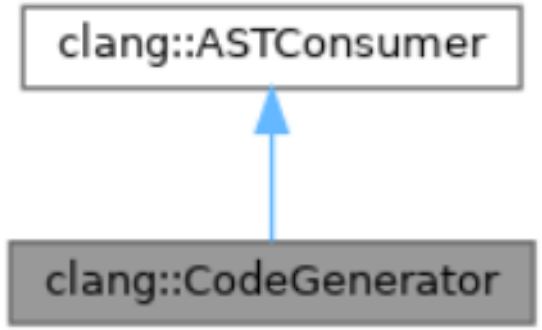
From: https://clang.llvm.org/doxygen/classclang_1_1SemaConsumer.html

CodeGenerator

- The primary public interface to the Clang code generator.

```
#include "clang/CodeGen/ModuleBuilder.h"
```

Inheritance diagram for clang::CodeGenerator:



From: https://clang.llvm.org/doxygen/classclang_1_1CodeGenerator.html

Contents

- Structure of Clang
- FrontendAction
- Preprocessor && Lexer
- Parser
- Sema
- CodeGen
- Clang Tools

Semantic

- The semantic analyzer walks the AST and checks for various semantic rules of the language. [1]
- Clang, on the other hand, does not traverse the AST after parsing. Instead, it performs type checking on the fly, together with AST node generation.[2]

[1]. <Learn LLVM 12> P61
[2]. <Getting Started with LLVM Core Library> P98

Contents

- Structure of Clang
- FrontendAction
- Preprocessor && Lexer
- Parser
- Sema
- CodeGen
- Clang Tools

CodeGen

- If the compiler driver used the CodeGenAction frontend action, this client will be BackendConsumer, which will traverse the AST while generating LLVM IR that implements the exact same behavior that is represented in the tree. The translation to LLVM IR starts at the top-level declaration, TranslationUnitDecl.[1]

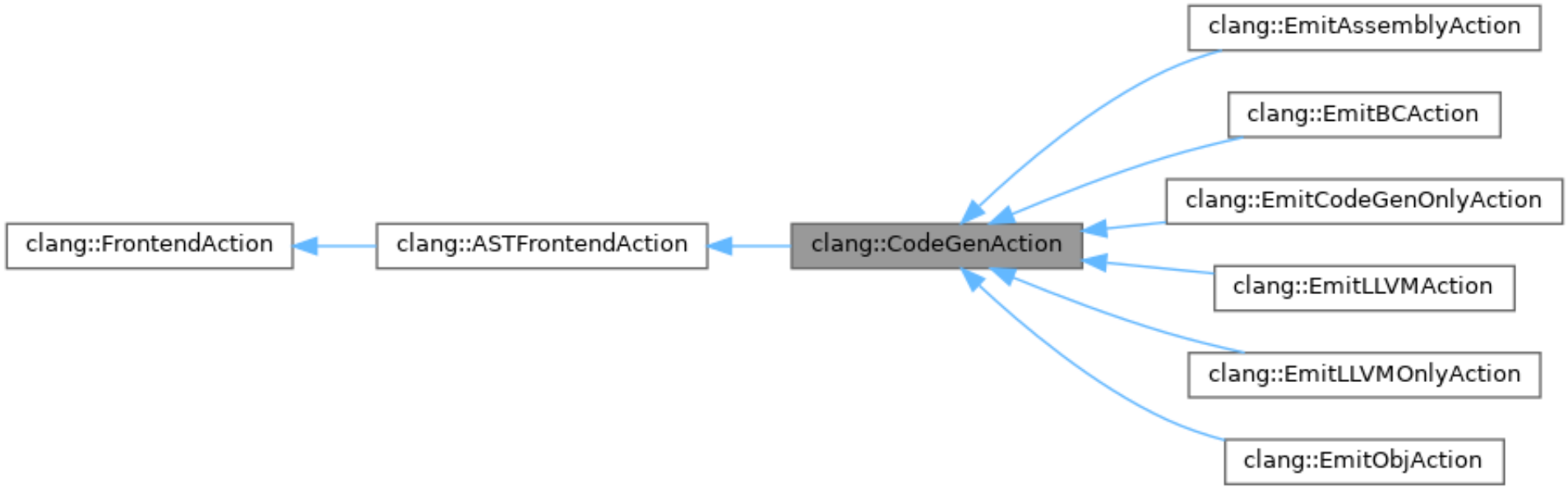
[1]. <Getting Started with LLVM Core Library> P99

CodeGenAction

- Create a new code generation action.[1]

```
#include "clang/CodeGen/CodeGenAction.h"
```

Inheritance diagram for clang::CodeGenAction:



From: https://clang.llvm.org/doxygen/classclang_1_1CodeGenAction.html

BackendConsumer

- llvm/clang/lib/CodeGen/CodeGenAction.cpp

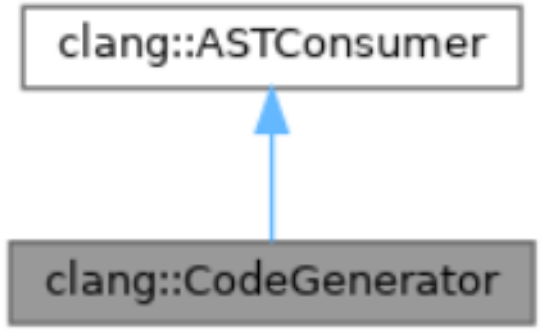
https://clang.llvm.org/doxygen/classclang_1_1BackendConsumer.html

CodeGenerator

- The primary public interface to the Clang code generator.

```
#include "clang/CodeGen/ModuleBuilder.h"
```

Inheritance diagram for clang::CodeGenerator:



From: https://clang.llvm.org/doxygen/classclang_1_1CodeGenerator.html

Contents

- Structure of Clang
- FrontendAction
- Preprocessor && Lexer
- Parser
- Sema
- CodeGen
- Clang Tools

Clang Plugins

- Clang Plugins make it possible to run extra user defined actions during a compilation. [1]
- A Clang plugin allows you to dynamically register a new FrontendAction (more specifically, an ASTFrontendAction) that can process the AST either before or after, or even replace, the main action of clang. [2]

[1]. https://clang.llvm.org/docs/ClangPlugins.html
[2]. <LLVM Techniques, Tips, and Best Practices – Clang and Middle-End Libraries> P82

libTooling

- LibTooling is a library to support writing standalone tools based on Clang.[1]
- Tools built with LibTooling, like Clang Plugins, run FrontendActions over code.[1]
- LibTooling is a library that provides features for building standalone tools on top of Clang‘s techniques. [2]

[1]. https://clang.llvm.org/docs/LibTooling.html
[2]. <LLVM Techniques, Tips, and Best Practices – Clang and Middle-End Libraries> P83

libclang

- The C Interface to Clang provides a relatively small API that exposes facilities for parsing source code into an abstract syntax tree (AST), loading already-parsed ASTs, traversing the AST, associating physical source locations with elements within the AST, and other facilities that support Clang-based development tools.[1]
- This C interface to Clang will never provide all of the information representation stored in Clang's C++ AST, nor should it: the intent is to maintain an API that is relatively stable from one release to the next, providing only the basic functionality needed to support development tools.[1]
- Source code: llvm/clang/tools/libclang
- libclang example： https://github.com/shining1984/screader

[1]. https://clang.llvm.org/doxygen/group__CINDEX.html#details

Thanks!