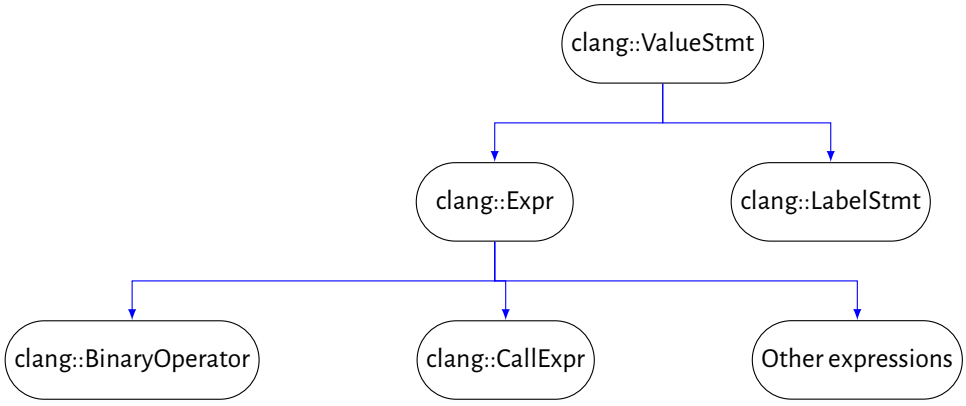

Figure 3.2: Clang AST: Statements with a value

3.1.2 Declarations

Declarations can also be combined into 2 primary groups: Declarations with context and without. Declarations with context can be considered as placeholders for other declarations. For example C++ namespace as well as translation unit or function declaration might contain other declarations. Declaration of a friend entity (`clang::DeclFriend`) can be considered as an example of a declaration without context.

It has to be noted that classes that are inherited from `DeclContext` have also `clang::Decl` as their top parent.

Some declarations can be redeclarated as in the following example

```
1 extern int a;
2 int a = 1;
```

Such declarations have an additional parent that is implemented via `clang::Redeclarable<...>` template.

3.1.3 Types

C++ is a statically typed language, which means that the types of variables must be declared at compile-time. The types allow compiler to make a reasonable conclusion about the program meaning i.e. types are important part of semantic analysis. `clang::Type` is the basic class for types in Clang.

Types in C/C++ might have qualifiers that are called as CV-qualifiers at standard [for Standardization, 2020, basic.type.qualifier]. CV here is for 2 keywords `const` and `volatile` that can be used as the qualifier for a type.

> *C99 standard has an additional type qualifier `restrict` that is also supported by clang [International Organization for Standardization (ISO), 1999, 6.7.3]*

Clang has a special class to support a type with qualifier: `clang::QualType` that is a pair of a pointer to `clang::Type` and a bit mask with information about the type qualifier. The class has a method to retrieve a pointer to the `clang::Type` and check different qualifiers. The code below shows how we can check a type for a const qualifier (the code is from LLVM release 16.0, `clang/lib/AST/ExprConstant.cpp`, line 3855)

```
bool checkConst(QualType QT) {
  // Assigning to a const object has undefined behavior.
  if (QT.isConstQualified()) {
    Info.FFDiag(E, diag::note_constexpr_modify_const_type) << QT;
    return false;
  }
  return true;
}
```

It's worth mentioning that `clang::QualType` has `operator->()` and `operator*()` implemented, i.e. it can be considered as a smart pointer for underlying `clang::Type` class.

In addition to the qualifiers type can have additional information that represents different memory address models, for instance there can be a pointer to an object or reference. `clang::Type` has the following helper methods to check different address models:

- `clang::Type::isPointerType()` for pointer type check
- `clang::Type::isReferenceType()` for reference type check

Types in C/C++ can also use aliases that are introduced by using `typedef` or `using` keywords. The following code defines `foo` and `bar` as aliases for `int` type

```
using foo = int;
typedef int bar;
```

Original types, `int` at our case, are called as canonical. You can test if the type is canonical or not using `clang::QualType::isCanonical()` method. `clang::QualType` also provides a method to retrieve the canonical type from an alias: `clang::QualType::getCanonicalType()`.

3.2 AST traversal

The compiler requires traversal of the AST to generate IR code. Thus, having a well-structured data structure for tree traversal is paramount for AST design. To put it another way, the design of the AST should prioritize facilitating easy tree traversal. A standard approach in many systems is to have a common base class for all AST nodes. This class typically provides a method to retrieve the node's children, allowing for tree traversal using popular algorithms like Breadth-First Search (BFS) Cormen et al. [2009]. Clang, however, takes a different approach: its AST nodes don't share a common ancestor. This poses the question: how is tree traversal organized in Clang?

Clang employs three unique techniques:

- The Curiously Recurring Template Pattern (CRTP) for visitor class definition,
- Ad-hoc methods tailored specifically for different nodes,
- Macros, which can be perceived as the connecting layer between the ad-hoc methods and CRTP.

We will explore these techniques through a simple program designed to identify function definitions and display the function names together with their parameters.

3.2.1 DeclVisitor test tool

Our test tool will build upon the `clang::DeclVisitor` class, which is defined as a straightforward visitor class aiding in the creation of visitors for C/C++ declarations.

We will use the same CMake file as it was created for our first clang tool (see fig. 1.6). The sole addition for the new tool is the `clangAST` library. The resultant `CMakeLists.txt` is shown in fig. 3.3:

```
project("declvisitor")

if ( NOT DEFINED ENV{LLVM_HOME})
  message(FATAL_ERROR "$LLVM_HOME is not defined")
else()
  message(STATUS "$LLVM_HOME found: $ENV{LLVM_HOME}")
  set(LLVM_HOME $ENV{LLVM_HOME} CACHE PATH "Root of LLVM installation")
  set(LLVM_LIB ${LLVM_HOME}/lib)
  set(LLVM_DIR ${LLVM_LIB}/cmake/llvm)
  find_package(LLVM REQUIRED CONFIG)
  include_directories(${LLVM_INCLUDE_DIRS})
  link_directories(${LLVM_LIBRARY_DIRS})
  set(SOURCE_FILE DeclVisitor.cpp)
  add_executable(declvisitor ${SOURCE_FILE})
  set_target_properties(declvisitor PROPERTIES COMPILE_FLAGS "-fno-rtti")
  target_link_libraries(declvisitor
    LLVMSupport
    clangAST
    clangBasic
    clangFrontend
    clangSerialization
    clangTooling
  )
```

Figure 3.3: CMakeLists.txt file for DeclVisitor test tool

The `main` function of our tool is presented below:

```
1  #include "clang/Tooling/CommonOptionsParser.h"
2  #include "clang/Tooling/Tooling.h"
3  #include "llvm/Support/CommandLine.h" // llvm::cl::extrahelp
4
5  #include "FrontendAction.hpp"
6
7  namespace {
8  llvm::cl::OptionCategory TestCategory("Test project");
9  llvm::cl::extrahelp
10     CommonHelp(clang::tooling::CommonOptionsParser::HelpMessage);
11 } // namespace
12
13 int main(int argc, const char **argv) {
14   llvm::Expected<clang::tooling::CommonOptionsParser> OptionsParser =
15       clang::tooling::CommonOptionsParser::create(argc, argv, TestCategory);
16   if (!OptionsParser) {
17     llvm::errs() << OptionsParser.takeError();
18     return 1;
19   }
20   clang::tooling::ClangTool Tool(OptionsParser->getCompilations(),
21                                  OptionsParser->getSourcePathList());
22   return Tool.run(clang::tooling::newFrontendActionFactory<
23                       clangbook::declvisitor::FrontendAction>()
24                       .get());
25 }
```

Figure 3.4: The main function for DeclVisitor test tool

From lines 5 and 23, it's evident that we employ a custom frontend action specific to our project: `clangbook::declvisito`
The code for this class is provided below

```
1  #include "Consumer.hpp"
2  #include "clang/Frontend/FrontendActions.h"
3
4  namespace clangbook {
5  namespace declvisitor {
6  class FrontendAction : public clang::ASTFrontendAction {
7  public:
8    virtual std::unique_ptr<clang::ASTConsumer>
9    CreateASTConsumer(clang::CompilerInstance &CI,
10                     clang::StringRef File) override {
11     return std::make_unique<Consumer>();
12   }
13 };
14 } // namespace declvisitor
15 } // namespace clangbook
```

Figure 3.5: Custom FrontendAction class for DeclVisitor test tool

You'll notice that we have overridden the `clang::ASTFrontendAction::CreateASTConsumer` function to instantiate our custom `clangbook::declvisitor::Consumer` (as highlighted in lines 9-12). The implementation for our tailored AST consumer is as follows:

```
1  #include "Visitor.hpp"
2  #include "clang/Frontend/ASTConsumers.h"
3
4  namespace clangbook {
5  namespace declvisitor {
6  class Consumer : public clang::ASTConsumer {
7  public:
8    Consumer() : V(std::make_unique<Visitor>()) {}
9
10   virtual void HandleTranslationUnit(clang::ASTContext &Context) override {
11     V->Visit(Context.getTranslationUnitDecl());
12   }
13
14 private:
15   std::unique_ptr<Visitor> V;
16 };
17 } // namespace declvisitor
18 } // namespace clangbook
```

Figure 3.6: Consumer class for DeclVisitor test tool

Here, we can see that we've created a sample visitor and invoked it using the `clang::ASTConsumer::HandleTranslationUnit` (see fig. 3.6, line 11). However, the most intriguing portion is the code for the visitor:

```
1  #include "clang/AST/DeclVisitor.h"
2
3  namespace clangbook {
4  namespace declvisitor {
5  class Visitor : public clang::DeclVisitor<Visitor> {
6  public:
7    void VisitFunctionDecl(const clang::FunctionDecl *FD) {
8      llvm::outs() << "Function: '" << FD->getName() << "'\n";
9      for (auto Param : FD->parameters()) {
10       Visit(Param);
11     }
12   }
13   void VisitParmVarDecl(const clang::ParmVarDecl *PVD) {
14     llvm::outs() << "\tParameter: '" << PVD->getName() << "'\n";
15   }
16   void VisitTranslationUnitDecl(const clang::TranslationUnitDecl *TU) {
17     for (auto Decl : TU->decls()) {
18       Visit(Decl);
19     }
20   }
21 };
22 } // namespace declvisitor
23 } // namespace clangbook
```

Figure 3.7: Visitor class implementation

We will explore the code in more depth later. For now, we observe that it prints the function name at line 8 and the parameter name at line 14.

We can compile our program using the same sequence of commands as we did for our test project, as detailed in Section 1.4, Test project: syntax check with clang tool.

```
export LLVM_HOME=<...>/llvm-project/install
mkdir build
cd build
cmake -G Ninja -DCMAKE_BUILD_TYPE=Debug ..
ninja
```

Figure 3.8: Configure and build commands for DeclVisitor test tool

As you may notice we used `-DCMAKE_BUILD_TYPE=Debug` option for CMake. That is because we might want to investigate the result program under debugger.

We will use the same program we referenced in our previous investigations (see fig. 2.5) to also study AST traversal:

```
1 int max(int a, int b) {
2   if (a > b)
3     return a;
4   return b;
5 }
```

Figure 3.9: Test program max.cpp

This program consists of a single function, `max`, which takes two parameters: `a` and `b`, and returns the maximum of the two.

We can run our program as follows:

```
$ ./declvisitor max.cpp
...
Function: 'max'
        Parameter: 'a'
        Parameter: 'b'
```

Let's investigate the Visitor class implementation in detail.

3.2.2 Visitor implementation

Let's delve into the Visitor code (see fig. 3.7). Firstly, you'll notice an unusual construct where our class is derived from a base class parameterized by our own class:

```
5 class Visitor : public clang::DeclVisitor<Visitor> {
```

This construct is known as the Curiously Recurring Template Pattern, or CRTP.

The Visitor class has several callbacks that are triggered when a corresponding AST node is encountered. The first callback targets the AST node representing a function declaration

```
7  void VisitFunctionDecl(const clang::FunctionDecl *FD) {
8    llvm::outs() << "Function: '" << FD->getName() << "'\n";
9    for (auto Param : FD->parameters()) {
10     Visit(Param);
11   }
12 }
```

Figure 3.10: FunctionDecl callback

As shown in fig. 3.10, the function name is printed on line 8. Our subsequent step involves printing the names of the parameters. To retrieve the function parameters, we can utilize the `parameters()` method from the `clang::FunctionDecl` class. This method was previously mentioned as an ad-hoc approach for AST traversal. Each AST node provides its own methods to access child nodes. Since we have an AST node of a specific type (i.e., `clang::FunctionDecl*`) as an argument, we can employ these methods.

The function parameter is passed to the `Visit(...)` method of the base class `clang::DeclVisitor<>`. This call is subsequently transformed into another callback, specifically for the `clang::ParmVarDecl` AST node.

```
13 void VisitParmVarDecl(const clang::ParmVarDecl *PVD) {
14   llvm::outs() << "\tParameter: '" << PVD->getName() << "'\n";
15 }
```

Figure 3.11: ParmVarDecl callback

You might be wondering how this conversion is achieved. The answer lies in a combination of the CRTP and C/C++ macros. To understand this, we need to dive into the `Visit()` method implementation of the `clang::DeclVisitor<>` class. This implementation heavily relies on C/C++ macros, so to get a glimpse of the actual code, we must expand these macros. This can be done using the `-E` compiler option. Let's make some modifications to our `CMakeLists.txt` and introduce a new custom target.

```
25  add_custom_command(
26    OUTPUT ${SOURCE_FILE}.preprocessed
27    COMMAND ${CMAKE_CXX_COMPILER} -E -I ${LLVM_HOME}/include
      ↪  ${CMAKE_CURRENT_SOURCE_DIR}/${SOURCE_FILE} >
      ↪  ${SOURCE_FILE}.preprocessed
28    DEPENDS ${SOURCE_FILE}
29    COMMENT "Preprocessing ${SOURCE_FILE}"
30  )
31  add_custom_target(preprocess ALL DEPENDS ${SOURCE_FILE}.preprocessed)
```

Figure 3.12: Custom target to expand macros

We can run the target as follows

`ninja preprocess`

The resulting file can be located in the build folder specified earlier, named

`DeclVisitor.cpp.preprocessed` . This is the folder we pointed to when executing the cmake command (see fig. 3.8). Within this file, the generated code for the `Visit()` method appears as follows:

```
1 RetTy Visit(typename Ptr<Decl>::type D) {
2   switch (D->getKind()) {
3     ...
4     case Decl::ParmVar: return
      ↪  static_cast<ImplClass*>(this)->VisitParmVarDecl(static_cast<typename
      ↪  Ptr<ParmVarDecl>::type>(D));
5     ...
6   }
7 }
```

This code showcases the use of the CRTP in Clang. In this context, CRTP is employed to redirect back to our `Visitor` class, which is referenced as `ImplClass`. CRTP allows the base class to call a method from an inherited class. This pattern can serve as an alternative to virtual functions and offers several advantages, the most notable being performance-related. Specifically, the method call is resolved at compile-time, eliminating the need for a vtable lookup associated with virtual method calls.

The code was generated using C/C++ macros, as demonstrated below. This particular code was sourced from the `clang/include/clang/AST/DeclVisitor.h` header.

```
1  #define DISPATCH(NAME, CLASS) \
2  return
   ↪  static_cast<ImplClass*>(this)->Visit##NAME(static_cast<PTR(CLASS)>(D))
```

The NAME is replaced with the node name; in our case, it's ParmVarDecl.

The DeclVisitor is used to traverse C++ declarations. Clang also has StmtVisitor and TypeVisitor to traverse statements and types, respectively. These are built on the same principles as we considered in our example with the declaration visitor. However, these visitors come with several issues. They can only be used with specific groups of AST nodes. For instance, the DeclVisitor can only be used with descendants of the Decl class. Another limitation is that we are required to implement recursion. For example, we set up recursion to traverse the function declaration in lines 9-11 (fig. 3.7). The same recursion was employed to traverse declarations within the translation unit (see fig. 3.7, lines 17-19). This brings up another concern: it's possible to miss some parts of the recursion. For instance, our code will not function correctly if the max function declaration is specified inside a namespace. To address such scenarios, we would need to implement an additional visit method specifically for namespace declarations.

These challenges are addressed by the recursive visitor, which we will discuss shortly.

3.3 Recursive AST Visitor

Recursive AST visitors address the limitations observed with specialized visitors. We will create the same program, which searches for and prints function declarations along with their parameters, but we'll use a recursive visitor this time.

The CMakeLists.txt for recursive visitor test tool will be similar to used previously. The only project name and source file name was changed

```
cmake_minimum_required(VERSION 3.16)
project("recursivevisitor")

if ( NOT DEFINED ENV{LLVM_HOME})
  message(FATAL_ERROR "$LLVM_HOME is not defined")
else()
  message(STATUS "$LLVM_HOME found: $ENV{LLVM_HOME}")
  set(LLVM_HOME $ENV{LLVM_HOME} CACHE PATH "Root of LLVM installation")
  set(LLVM_LIB ${LLVM_HOME}/lib)
  set(LLVM_DIR ${LLVM_LIB}/cmake/llvm)
  find_package(LLVM REQUIRED CONFIG)
  include_directories(${LLVM_INCLUDE_DIRS})
  link_directories(${LLVM_LIBRARY_DIRS})
  set(SOURCE_FILE RecursiveVisitor.cpp)
  add_executable(recursivevisitor ${SOURCE_FILE})
  set_target_properties(recursivevisitor PROPERTIES COMPILE_FLAGS
   ↪  "-fno-rtti")
  target_link_libraries(recursivevisitor
    LLVMSupport
    clangAST
    clangBasic
    clangFrontend
    clangSerialization
    clangTooling
  )
endif()
```

Figure 3.13: CMakeLists.txt file for RecursiveVisitor test tool

The `main` function for our tool is similar to the 'DeclVisitor' one defined in fig. 3.4.

```
1  #include "clang/Tooling/CommonOptionsParser.h"
2  #include "clang/Tooling/Tooling.h"
3  #include "llvm/Support/CommandLine.h" // llvm::cl::extrahelp
4
5  #include "FrontendAction.hpp"
6
7  namespace {
8  llvm::cl::OptionCategory TestCategory("Test project");
9  llvm::cl::extrahelp
10     CommonHelp(clang::tooling::CommonOptionsParser::HelpMessage);
11 } // namespace
12
13 int main(int argc, const char **argv) {
14   llvm::Expected<clang::tooling::CommonOptionsParser> OptionsParser =
15       clang::tooling::CommonOptionsParser::create(argc, argv, TestCategory);
16   if (!OptionsParser) {
17     llvm::errs() << OptionsParser.takeError();
18     return 1;
19   }
20   clang::tooling::ClangTool Tool(OptionsParser->getCompilations(),
21                                  OptionsParser->getSourcePathList());
22   return Tool.run(clang::tooling::newFrontendActionFactory<
23                       clangbook::recursivevisitor::FrontendAction>()
24                       .get());
25 }
```

Figure 3.14: The main function for the RecursiveVisitor test tool.

As you can see, we changed only the namespace name for our custom frontend action at line 23.

The code for the frontend action and consumer is the same as in fig. 3.5 and fig. 3.6, with the only difference being the namespace change from `declvisitor` to `recursivevisitor`. The most interesting part of the program is the Visitor class implementation.

```
#include "clang/AST/RecursiveASTVisitor.h"

namespace clangbook {
namespace recursivevisitor {
class Visitor : public clang::RecursiveASTVisitor<Visitor> {
public:
  bool VisitFunctionDecl(const clang::FunctionDecl *FD) {
    llvm::outs() << "Function: '" << FD->getName() << "'\n";
    return true;
  }
  bool VisitParmVarDecl(const clang::ParmVarDecl *PVD) {
    llvm::outs() << "\tParameter: '" << PVD->getName() << "'\n";
    return true;
  }
};
} // namespace recursivevisitor
} // namespace clangbook
```

Figure 3.15: Visitor class implementation

There are several changes compared to the code for 'DeclVisitor' (see fig. 3.7). The first is that recursion isn't implemented. We've only implemented the callbacks for nodes of interest to us. A reasonable question arises: how is the recursion controlled? The answer lies in another change: our callbacks now return a boolean result. The value `false` indicates that the recursion should stop, while `true` signals the visitor to continue the traversal.

The program can be compiled using the same sequence of commands as we used previously, see fig. 3.8.

We can run our program as follows:

```
$ ./recursivevisitor max.cpp
...
Function: 'max'
        Parameter: 'a'
        Parameter: 'b'
```

As we can see, it produces the same result as we obtained with the DeclVisitor implementation. The AST traversal techniques considered so far are not the only ways for AST traversal. Most of the tools that we will consider later will use a different approach based on AST matchers.

3.4 AST matchers

AST matchers provide another approach for locating specific AST nodes. They can be particularly useful in linters when searching for improper pattern usage or in refactoring tools when identifying AST nodes for modification.

We will create a simple program to test AST matchers. The program will identify a function definition with the name `max`.

We will use a slightly modified `CMakeLists.txt` file from the previous examples.

```
1  cmake_minimum_required(VERSION 3.16)
2  project("matchvisitor")
3
4  if ( NOT DEFINED ENV{LLVM_HOME})
5    message(FATAL_ERROR "$LLVM_HOME is not defined")
6  else()
7    message(STATUS "$LLVM_HOME found: $ENV{LLVM_HOME}")
8    set(LLVM_HOME $ENV{LLVM_HOME} CACHE PATH "Root of LLVM installation")
9    set(LLVM_LIB ${LLVM_HOME}/lib)
10   set(LLVM_DIR ${LLVM_LIB}/cmake/llvm)
11   find_package(LLVM REQUIRED CONFIG)
12   include_directories(${LLVM_INCLUDE_DIRS})
13   link_directories(${LLVM_LIBRARY_DIRS})
14   set(SOURCE_FILE MatchVisitor.cpp)
15   add_executable(matchvisitor ${SOURCE_FILE})
16   set_target_properties(matchvisitor PROPERTIES COMPILE_FLAGS "-fno-rtti")
17   target_link_libraries(matchvisitor
18     LLVMFrontendOpenMP
19     LLVMSupport
20     clangAST
21     clangASTMatchers
22     clangBasic
23     clangFrontend
24     clangSerialization
25     clangTooling
26   )
27 endif()
```

Figure 3.16: CMakeLists.txt for AST matchers test tool

There are two additional libraries were added: `LLVMFrontendOpenMP` and `clangASTMatchers` (see lines 18 and 21 in fig. 3.16).

The `main` function for our tool looks like

```
1  #include "clang/Tooling/CommonOptionsParser.h"
2  #include "clang/Tooling/Tooling.h"
3  #include "llvm/Support/CommandLine.h" // llvm::cl::extrahelp
4
5  #include "MatchCallback.hpp"
6
7  namespace {
8  llvm::cl::OptionCategory TestCategory("Test project");
9  llvm::cl::extrahelp
10     CommonHelp(clang::tooling::CommonOptionsParser::HelpMessage);
11 } // namespace
12
13 int main(int argc, const char **argv) {
14   llvm::Expected<clang::tooling::CommonOptionsParser> OptionsParser =
15       clang::tooling::CommonOptionsParser::create(argc, argv, TestCategory);
16   if (!OptionsParser) {
17     llvm::errs() << OptionsParser.takeError();
18     return 1;
19   }
20   clang::tooling::ClangTool Tool(OptionsParser->getCompilations(),
21                                  OptionsParser->getSourcePathList());
22   clangbook::matchvisitor::MatchCallback MC;
23   clang::ast_matchers::MatchFinder Finder;
24   Finder.addMatcher(clangbook::matchvisitor::M, &MC);
25
26   return Tool.run(clang::tooling::newFrontendActionFactory(&Finder).get());
27 }
```

Figure 3.17: The main function for AST matchers test tool

As you can observe (lines 22-24), we employ the `MatchFinder` class and define a custom callback (included via the header in line 5) that outlines the specific AST node we intend to match.

The callback is implemented as follows:

```
1  #include "clang/ASTMatchers/ASTMatchFinder.h"
2  #include "clang/ASTMatchers/ASTMatchers.h"
3
4  namespace clangbook {
5  namespace matchvisitor {
6  using namespace clang::ast_matchers;
7  static const char *MatchID = "match-id";
8  clang::ast_matchers::DeclarationMatcher M =
9      functionDecl(decl().bind(MatchID), matchesName("max"));
10
11 class MatchCallback : public clang::ast_matchers::MatchFinder::MatchCallback
   ↪  {
12 public:
13   virtual void
14   run(const clang::ast_matchers::MatchFinder::MatchResult &Result) final {
15     if (const auto *FD =
       ↪  Result.Nodes.getNodeAs<clang::FunctionDecl>(MatchID)) {
16       const auto &SM = *Result.SourceManager;
17       const auto &Loc = FD->getLocation();
18       llvm::outs() << "Found 'max' function at " << SM.getFilename(Loc) <<
         ↪  ":"
19                    << SM.getSpellingLineNumber(Loc) << ":"
20                    << SM.getSpellingColumnNumber(Loc) << "\n";
21     }
22   }
23 };
24
25 } // namespace matchvisitor
26 } // namespace clangbook
```

Figure 3.18: The match callback for AST matchers test tool

The most crucial section of the code is located at lines 7-9. Each matcher is identified by an ID, which, in our case, is 'match-id'. The matcher itself is defined in lines 8-9:

```
clang::ast_matchers::DeclarationMatcher M =
    functionDecl(decl().bind(MatchID), matchesName("max"));
```

This matcher seeks a function declaration using `functionDecl()` that has a specific name, as seen in `matchesName()`. We utilized a specialized Domain-Specific Language (DSL) to specify the matcher. The DSL is implemented using C++ macros. It's worth noting that the recursive AST visitor serves as the backbone for AST traversal inside the matchers implementation.

The program can be compiled using the same sequence of commands as we used previously, see fig. 3.8.

We will utilize a slightly modified version of the example shown in fig. 2.5, with an additional function added.

```
int max(int a, int b) {
  if (a > b) return a;
  return b;
}

int min(int a, int b) {
  if (a > b) return b;
  return a;
}
```

Figure 3.19: Test program minmax.cpp for AST matchers

When we run our test tool on the example, we will obtain the following output:

```
./matchvisitor minmax.cpp
...
Found 'max' function at minmax.cpp:1:5
```

As we can see, it has located only one function declaration with the name specified for the matcher.

The AST matchers reference page (https://clang.llvm.org/docs/LibASTMatchersReference.html) provides extensive information about various matchers and their usage.

The DSL for matchers is typically employed in custom Clang tools, such as clang-tidy (as discussed in Chapter 5, **clang-tidy linter framework**), but it can also be used as a standalone tool. A specialized program called `clang-query` enables the execution of different match queries.

3.5 Explore clang AST with clang-query

AST matchers are incredibly useful, and there's a utility that facilitates checking various matchers and analyzing the AST of your source code. This utility is known as `clang-query` . You can build and install this utility using the following command.

```
ninja install-clang-query
```

You can run the tool as follows

```
llvm-project/install/bin/clang-query minmax.cpp
```

We can use `match` command as follows

```
clang-query> match functionDecl(decl().bind("match-id"), matchesName("max"))

Match #1:

minmax.cpp:1:1: note: "match-id" binds here
int max(int a, int b) {
^~~~~~~~~~~~~~~~~~~~~~
minmax.cpp:1:1: note: "root" binds here
int max(int a, int b) {
^~~~~~~~~~~~~~~~~~~~~~
1 match.
clang-query>
```

The default output, referred to as `diag` , is available. Among several potential outputs, the most relevant one for us is `dump` . When the output is set to `dump` , clang-query will display the located AST node. For example, the following demonstrates how to match a function parameter named `a` :

```
clang-query> set output dump
clang-query> match parmVarDecl(hasName("a"))

Match #1:

Binding for "root":
ParmVarDecl 0x6775e48 <minmax.cpp:1:9, col:13> col:13 used a 'int'


Match #2:

Binding for "root":
ParmVarDecl 0x6776218 <minmax.cpp:6:9, col:13> col:13 used a 'int'

2 matches.
clang-query>
```

This tool proves useful when you wish to test a particular matcher or investigate a portion of the AST tree. We will utilize this tool to explore how Clang handles compilation errors.

3.6 Processing AST in the case of errors

One of the most interesting aspects of Clang pertains to error processing. Error processing encompasses error detection, the display of corresponding error messages, and potential error recovery. The latter is particularly intriguing in terms of the Clang AST. Error recovery occurs when Clang doesn't halt upon encountering a compilation error but continues to compile in order to detect additional issues.

Such behavior is beneficial for various reasons. The most evident one is user convenience. When programmers compile a program, they typically prefer to be informed about as many errors as possible in a single compilation run. If the compiler were to stop at the first error, the programmer would have to correct that error, recompile, then address the subsequent error, and recompile again, and so forth. This iterative process can be tedious and frustrating, especially with larger codebases or intricate errors.

Another compelling reason centers on IDE integration, which will be discussed in more detail in Chapter 8, **IDE support and code navigation**. IDEs offer navigation support coupled with an integrated compiler. We will explore `clangd` as one of such tools. Editing code in IDEs often leads to compilation errors. Most errors are confined to specific sections of the code, and it might be suboptimal to cease navigation in such cases.

Clang employs various techniques for error recovery. For the syntax stage of parsing, it utilizes heuristics; for instance, if a user forgets to insert a semicolon, Clang may attempt to add it as part of the recovery process. The semantic stage of parsing is more intricate, and Clang implements unique techniques to manage recovery during this phase.

Consider a program (maxerr.cpp) that is syntactically correct but has a semantic error. For example, it might use an undeclared variable. In this program, refer to line 3 where the undeclared variable `ab` is used

```
1 int max(int a, int b) {
2   if (a > b) {
3     return ab;
4   }
5   return b;
6 }
```

Figure 3.20: The maxerr.cpp test program with a semantic error, undeclared variable

We are interested in the AST result produced by Clang, and we will use `clang-query` to examine it, which can be run as follows:

```
llvm-project/install/bin/clang-query maxerr.cpp
...
maxerr.cpp:3:12: error: use of undeclared identifier 'ab'
    return ab;
           ^
```

From the output, we can see that clang-query displayed a compilation error detected by the compiler. It's worth noting that despite this, an AST was produced for the program, and we can examine it. We are particularly interested in the return statements and can use the corresponding matcher to highlight the relevant parts of the AST.

We will also set up the output to produce the AST and search for return statements that are of interest to us

```
clang-query> set output dump
clang-query> match returnStmt()
```

The resulting output identifies two return statements in our program: the first match on line 5 and the second match on line 3

```
Match #1:

Binding for "root":
ReturnStmt 0x6b63230 <maxerr.cpp:5:3, col:10>
`-ImplicitCastExpr 0x6b63218 <col:10> 'int' <LValueToRValue>
  `-DeclRefExpr 0x6b631f8 <col:10> 'int' lvalue ParmVar 0x6b62ec8 'b' 'int'


Match #2:

Binding for "root":
ReturnStmt 0x6b631b0 <maxerr.cpp:3:5, col:12>
`-RecoveryExpr 0x6b63190 <col:12> '<dependent type>' contains-errors lvalue

2 matches.
```

Figure 3.21: ReturnStmt node matches at maxerr.cpp test program

As we can see, the first match corresponds to semantically correct code on line 5 and contains a reference to the parameter `a`. The second match is for line 3, which has a compilation error. Notably, Clang has inserted a special type of AST node: `RecoveryExpr`. It's worth noting that in certain situations, Clang might produce an incomplete AST. This can cause issues with Clang tools, such as lint checks. In instances of compilation errors, lint checks might yield unexpected results because Clang couldn't recover accurately from the compilation errors. We will revisit the problem when exploring the clang-tidy lint check framework in Chapter 5, **clang-tidy linter framework**.

3.7 Summary

We explored the Clang AST, a major instrument for creating various Clang tools. We learned about the architectural design principles chosen for the implementation of Clang AST and investigated different methods for AST traversal. We delved into specialized traversal techniques, such as those for C/C++ declarations, and also looked into more universal techniques that employ recursive visitors and Clang AST matchers. Our exploration concluded with the `clang-query` tool and how it can be used for Clang AST exploration. Specifically, we used it to understand how Clang processes compilation errors.

The next chapter will discuss the basic libraries used in Clang and LLVM development. We will explore the LLVM code style and foundational Clang/LLVM classes, such as `SourceManager` and `SourceLocation`. We will also cover the TableGen library, which is used for code generation, and the LIT (LLVM Integration Test) framework.

3.8 Further reading

- How to write RecursiveASTVisitor: https://clang.llvm.org/docs/RAVFrontendAction.html
- AST Matcher Reference: https://clang.llvm.org/docs/LibASTMatchersReference.html

4

Basic libraries and tools

LLVM is written in the C++ language and, as of July 2022, it uses the C++17 version of the C++ standard [Community, 2022b]. On the other hand, LLVM contains numerous internal implementations for fundamental containers, primarily aimed at optimizing performance. Hence, familiarity with these extensions is essential for anyone wishing to work with LLVM and clang. Additionally, LLVM has introduced other development tools such as TableGen, a domain-specific language (DSL) designed for structural data processing, and LIT, the LLVM test framework. More details about these tools are discussed later in this chapter.

We plan to use a simple example project to demonstrate these tools. This project will be a Clang plugin that estimates the complexity of the source code being compiled. It will also print a warning if the number of functions or methods exceeds a limit specified as a parameter.

4.1 LLVM coding style

LLVM adheres to specific code-style rules [Community, 2023b]. The primary objective of these rules is to promote proficient C++ practices with a special focus on performance. As previously mentioned, LLVM employs C++17 and prefers using data structures and algorithms from the STL (Standard Template Library). On the other hand, LLVM offers many optimized versions of data structures that mirror those in the STL. For example, `llvm::SmallVector<>` can be regarded as an optimized version of `std::vector<>`, especially for small vector sizes, a common trait for data structures used in compilers.

Given a choice between an STL object/algorithm and its corresponding LLVM version, the LLVM coding

standard advises favoring the LLVM version.

Other rules related to performance concern restrictions. For instance, both RTTI (Run-Time Type Information) and C++ exceptions are disallowed. However, there are situations where RTTI could prove beneficial; thus, LLVM offers alternatives like `llvm::isa<>` and other similar template helper functions. More information on this can be found in Section 4.2.1, RTTI replacement and cast operators. Instead of C++ exceptions, LLVM frequently employs C-style `asserts`.

Sometimes, asserts are not sufficiently informative. LLVM recommends adding textual messages to them to simplify debugging. Here's a typical example from Clang's code:

```
static bool unionHasUniqueObjectRepresentations(const ASTContext &Context,
                                                const RecordDecl *RD) {
  assert(RD->isUnion() && "Must be union type");
  ...
```

In the code, we check if the parameter is a union and raise an assert with a corresponding message if it's not.

Besides performance considerations, LLVM also introduces some additional requirements. One of these requirements concerns comments. Code comments are very important. Furthermore, both LLVM and Clang have comprehensive documentation generated from the code. They use Doxygen (https://www.doxygen.nl/) for this purpose. This tool is the de facto standard for commenting in C/C++ programs, and you have most likely encountered it before.

Clang and LLVM are not monolithic pieces of code but are implemented as a set of libraries. These libraries can be considered good examples of LLVM code style enforcement. Let's look at the libraries in detail.

4.2 LLVM basic libraries

We are going to start with RTTI replacement in LLVM code and discuss how it's implemented. We will then continue with basic containers and smart pointers. We will conclude with some important classes used to represent token locations and how diagnostics are realized in Clang. Later (see Section 4.5, Clang plugin project), we will use some of these classes in our test project.

4.2.1 RTTI replacement and cast operators

As mentioned earlier, LLVM avoids using RTTI due to performance concerns. LLVM has introduced several helper functions that replace RTTI counterparts, allowing for the casting of an object from one type to another. The fundamental ones are as follows:

- `llvm::isa<>` is akin to Java's `instanceof` operator. It returns `true` or `false` depending on whether the reference to the tested object belongs to the tested class or not.

- `llvm::cast<>`: Use this cast operator when you're certain that the object is of the specified derived type. If the cast fails (i.e., the object isn't of the expected type), `llvm::cast` will abort the program. Use it only when you're confident the cast won't fail.

- `llvm::dyn_cast<>`: This is perhaps the most frequently used casting operator in LLVM. `llvm::dyn_cast` is employed for safe downcasting when you anticipate the cast will usually succeed, but there's some uncertainty. If the object isn't of the specified derived type,`llvm::dyn_cast<>` returns `nullptr`.

The cast operators do not accept `nullptr` as input. However, there are two special cast operators that can handle null pointers:

- `llvm::cast_if_present<>`: A variant of `llvm::cast<>` that accepts `nullptr` values.

- `llvm::dyn_cast_if_present<>`: A variant of `llvm::dyn_cast<>` that accepts `nullptr` values.

Both operators can handle `nullptr` values. If the input is `nullptr` or if the cast fails, they simply return `nullptr`.

> *It's worth noting that `llvm::cast_if_present<>` and `llvm::dyn_cast_if_present<>` were introduced recently, specifically in 2022. They serve as replacements for `llvm::cast_or_null<>` and `llvm::dyn_cast_or_null<>`, which had been in recent use. The older versions are still supported and now redirect calls to the newer cast operators. For more information, see the discussion about this change: https://discourse.llvm.org/t/psa-swapping-out-or-null-with-if-present/65018*

The following question might arise: how can the dynamic cast operation be performed without RTTI? This can be achieved with certain specific decorations, as illustrated in a simple example inspired by [Community, 2023f]. We'll begin with a base class `clangbook::Animal` that has two descendants: `clangbook::Horse` and `clangbook::Sheep`. Each horse can be categorized by its speed (in mph), and each sheep by its wool mass. Here's how it can be used:

```
46 void testAnimal() {
47   auto AnimalPtr = std::make_unique<clangbook::Horse>(10);
48   if (llvm::isa<clangbook::Horse>(AnimalPtr)) {
49     llvm::outs()
50         << "Animal is a Horse and the horse speed is: "
51         << llvm::dyn_cast<clangbook::Horse>(AnimalPtr.get())->getSpeed()
52         << "mph \n";
53   } else {
54     llvm::outs() << "Animal is not a Horse\n";
55   }
56 }
```

Figure 4.1: LLVM `isa<>` and `dyn_cast<>` usage example

The code should produce the following output

```
Animal is a Horse and the horse speed is: 10mph
```

Line 46 in fig. 4.1 demonstrates the use of `llvm::isa<>`, while line 56 showcases `llvm::dyn_cast<>`. In the latter, we cast the base class to `clangbook::Horse` and call a method specific to that class.

Let's look into the class implementations, which will provide insights into how the RTTI replacement works. We will start with the base class `clangbook::Animal`:

```
9  class Animal {
10 public:
11   enum AnimalKind { AK_Horse, AK_Sheep };
12
13 public:
14   Animal(AnimalKind K) : Kind(K){};
15   AnimalKind getKind() const { return Kind; }
16
17 private:
18   const AnimalKind Kind;
19 };
```

Figure 4.2: `clangbook::Animal` class