LLVM

Architecture of *clang*

Analyze an open source compiler based on LLVM

Christopher Guntli

June 3, 2011

HSR - University of Applied Science in Rapperswil

Abstract

Clang is an open source compiler for C/C++/Objective-C. It is written in modern C++ using STL[AS95]. Clang is designed as a drop-in replacement for GCC[GNU87]. Therefore it has some similarities in the architecture. One of clangs main design goal was to make clang usable in an IDE for code completion and refactoring. Clang has also a plug-in API to extend it with additional functionalities like checking naming conventions.

1 Introduction

A few years ago in 2002 a group at the University of Illinois started to working on a low level virtual machine (LLVM). *clang* is an open source compiler for C/C++ and Objective-C using the LLVM infrastructure. Like LLVM, *clang* is also written in C++. *clang* is designed to be a drop in replacement for GCC. Therefore, they use the same command line arguments to specify the compiler options.

1.1 Motivation for Creating a new Compiler

There are multiple reasons for creating another open source compiler. It's getting more complicated to extend GCC in each new release and GCC gets also slower with every new version. It takes a lot of time to get familiar with GCC to extend it with new features[cla10a].

Since GCC[GNU87] have changed the license to GPLv3[Gnu07a] for GCC, some open source projects are stuck with older releases due to license issues. The source code for *clang* is under the more liberal NCSA license, which is based on the MIT license and the BSD license.

clang focuses on a modern and easy to understand design and on IDE integration. The authors of *clang* believe it doesn't make sense that each IDE has its own C++ parser to help the developer with auto completion and refactoring. By using the same libraries in the IDE as for the compiler the IDE should be able to check if the code could be accepted by the compiler.

1.2 Features of *clang*

clang is a C/C++/Objective-C frontend for LLVM[llv10]. Therefore, it can use all of the LLVM optimization algorithms to produce efficient code. But the main design goal was to provide user friendly error messages. *clang* prints the source code with a marker to the location where the parser ran into a problem. It also has a high level interface to access the AST, cache it and cross reference it over multiple files with an indexer. *clang* also has a plug-in interface for extending it with additional actions to perform an AST.

1.2.1 Expressive Diagnostics

The *clang* team aims to provide as clear and expressive error messages as possible. They try to make it as user friendly as they can for a command line compiler. To do this *clang* should pinpoint exactly what is wrong in the code[cla10b]. This is done through a diagnostics engine

which processes the error information into a user friendly message.

1.2.2 GCC Compatibility

GCC has lots of extensions to the standard language definition. However, since most people only care about their code getting compiled *clang* has to support all these extensions as well. Otherwise most open source projects wouldn't care about another compiler[cla10b].

It would be nice to focus on the language specification but since GCC is the defacto-standard for open source compiler. GCC is used to compile an enormous amount of code today. Therefore, *clang* has no chance to be an alternative if it doesn't understand the extensions the open source projects are using from GCC because these projects have to be compiled, preferably with an open source compiler [cla10b].

However, in *clang* extensions are explicitly recognized and as such extension diagnostics can write error and warning messages or just ignore them, if they are used[cla10b].

2 Architecture

clang is designed as a cross compiler. Therefore, it is no problem to adapt it to new architectures. However, *clang* is only a compiler and depends on the platforms tool chain to create libraries and executables, *clang* only creates object files.

2.1 Library Based Architecture

A library-based architecture makes the reuse and integration of functionality provided by *clang* more flexible and easier to integrate into other projects. In addition the library based architecture encourages clean APIs and separation. Therefore, making it easier for developers to understand, since they only have to understand small pieces of the big picture).

The following list shows *clang*s base libraries. Besides the **main** function and some functions to handle command line arguments every line of code is in a library and could be linked into another project.

libsupport Basic support library, from LLVM.

libsystem System abstraction library, from LLVM.

libbasic Diagnostics, SourceLocations, SourceBuffer abstraction, file system caching for input source files.

libast Provides classes to represent the C AST, the C type system, built-in functions, and various helpers for analyzing and manipulating the AST (visitors, pretty printers, etc.).

liblex Lexing and preprocessing, identifier hash table, pragma handling, tokens, and macro expansion.

libparse Parsing. This library invokes coarse-grained 'Actions' provided by the client (e.g. libsema builds ASTs) but knows nothing about ASTs or other client-specific data structures.

libsema Semantic Analysis. This provides a set of parser actions to build a standardized AST for programs.

libcodegen "Lower" the AST to LLVM IR for optimization & code generation.

librewrite Editing of text buffers (important for code rewriting transformation, like refactoring).

libanalysis Static analysis support.

libindex Cross-translation-unit infrastructure and indexing support.

clang A driver program, client of the libraries at various levels.

[cla10b]

2.2 Driver, Console Line Interface

The entry point for the compiler, the *main* function is together with some functions for handling the command line arguments in a file called driver. The driver has two operation modes. The user usually starts *clang* in the first mode, where it creates a list with all possible parameters using the arguments passed by the user and default values for every parameter not defined. *clang* then starts a new process with this list of parameters for each specified source file and uses the *cc1* parameter. This way each process enters the *cc1* mode and compiles the source file, see figure 3

The second mode of *clang* is used to actual compile source files. However, this is again split into two methods. One called *cc1as_main* for assembler files (.s, .asm) and *cc1_main* for all other files to compile them.

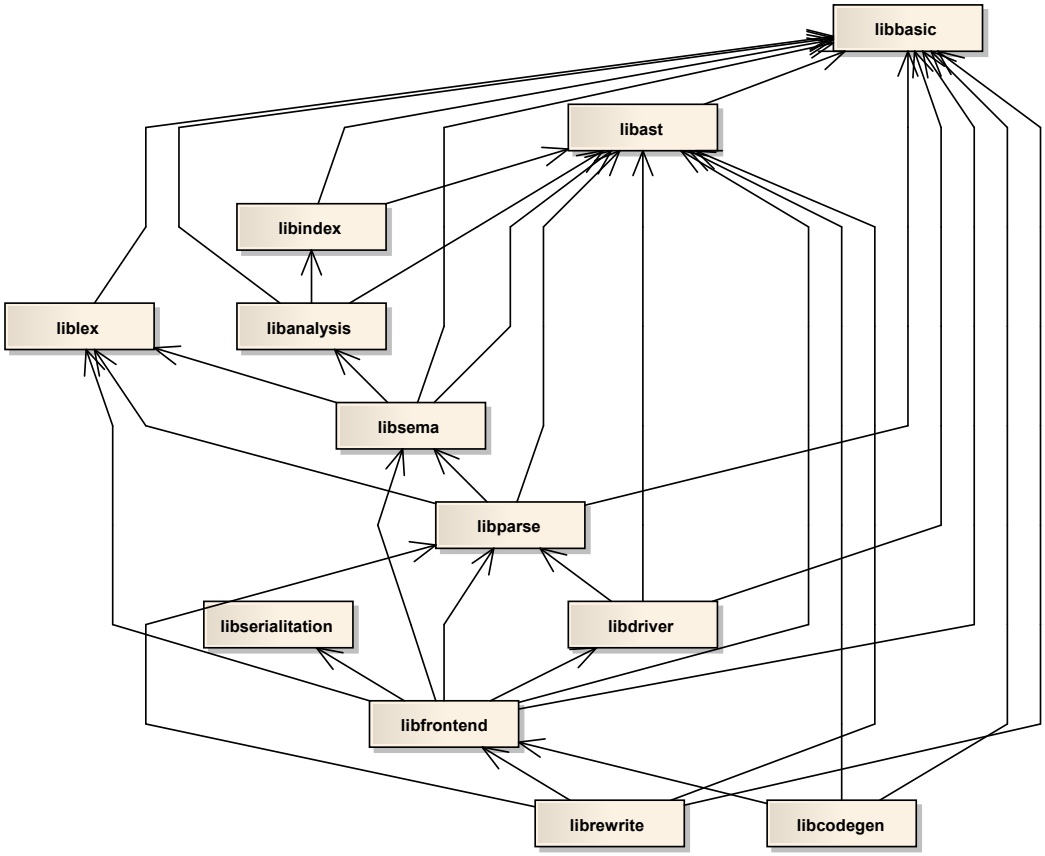

Figure 1: *clangs* libraries and their dependencies

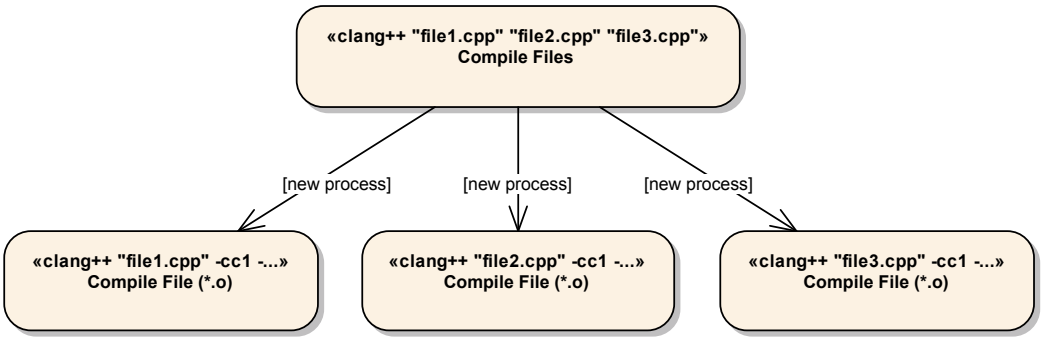

Figure 2: Driver starts a process with the -cc1 argument for all files

2.3 cc1_main

In *cc1_main clang* creates a *CompilerInstance* which is the class that represents the actual compiler. The *ComilerInstance* initializes its preprocessor parser and AST.

2.4 FrontEndActions

A *FrontendAction* is a task that should be performed on an AST. They implement the AST visitor interface called *ASTConsumer*. Most actions write their output into a file or to the standard output. A *FrontendAction* is for example code generation or preprocess only. Additional *FrontendActions* can be loaded through a plug-in interface.

2.5 Lexer, Parser and Preprocessor

To parse source code the lexer has to convert it into a token stream for the parser. Unfortunately the token stream has to be processed by a preprocessor that expands preprocessor directives while creating the tokens from the source code. The lexer uses a preprocessor to process the preprocessor directives. While processing the source code the lexer calls the preprocessor as soon it encounters a preprocessor directive. The lexer passes the identifier of the preprocessing directive and itself to the preprocessor. The preprocessor reads all tokens from the lexer until it gets a newline token. The preprocessor saves these tokens and adds them to a preprocessor symbol table with the identifier passed from the lexer in the first place. The lexer then continues on the next token after the newline. If the lexer reads an identifier token, it has to check with the preprocessor if the name has been defined in a preprocessor directive. If so the preprocessor returns the tokens to replace the identifier in the code as shown in Figure 4.

2.6 AST, Abstract Syntax Tree

clang has only one type of an AST for C++ C and Objective-C. However, there are some special nodes which apply only for one of the languages. *clangs* AST has two different base classes for nodes in the AST, *Decl* for declarations and *Stmt* for statements. To represent the complete AST a lot of derived classes from these two base classes exist.

The parser stores each declaration in a a *DeclContext*, which forms the actual tree for the code. A function called *ParseAST* uses an instance of the *Parser* class to get the root declarations from the source code and passes them to an *ASTConsumer*. Figure 5 shows how *clang* parses source code.

Each parse function, returns a node for the AST. The calling function then builds the parent node by using all nodes returned.

2.6.1 DeclContext

The named declarations are all stored in an instance of *DeclContex*. However some declarations are also derived from *DeclContext*. For example a namespace node has two base classes one is *NamedDecl* for the namespace declaration in the current *DeclContext* and the other is the *DeclContext* for the members.

2.6.2 Example AST

clang has an xml writer for the AST. This can be used to print the ast for a translation unit into a file or to the standard output. Listing 1 shows a hello-world in C++ and Listing 2 shows the AST created for if. However, to make the example not to complicated the AST nodes for the includes were removed. The symbol table, which is included in the AST as reference section was also removed.

2.7 Symbol Table

clang has a symbol map storing the memory layout for each user defined type. But the actual symbol lookup is done with the AST. The AST uses an *ASTContext* for long living symbols like classes. These symbols are used by the parser to parse C++ correctly, since some grammar rules depend on the information if an identifier is a type. Variables and functions are stored in the *DeclContext*, which is part of the AST.

A symbol is always looked up in the *DeclContext*. Each *DeclContext* has a reference to its parent. If a symbol is not found in the current *DeclContext* it is been looked up in the parent context. The *DeclContext* tree implements the scoping.

2.7.1 IdentifierTable

clang has a map for all identifiers and where they were parsed (source file) and stores its location. However, the information in the IdentifierTable is not meant for a symbol lookup. It is for checking if an identifier exists. The

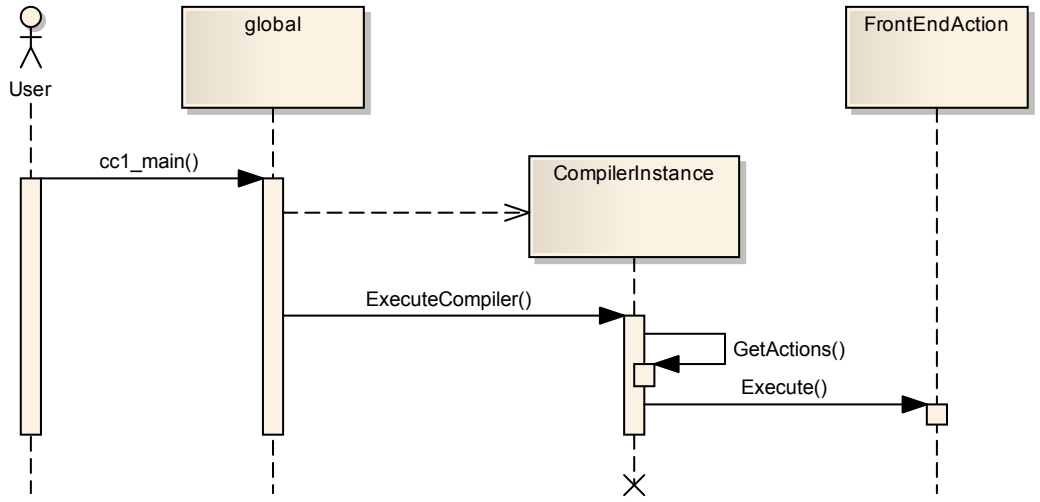

Figure 3: Concept of the compiler using Actions

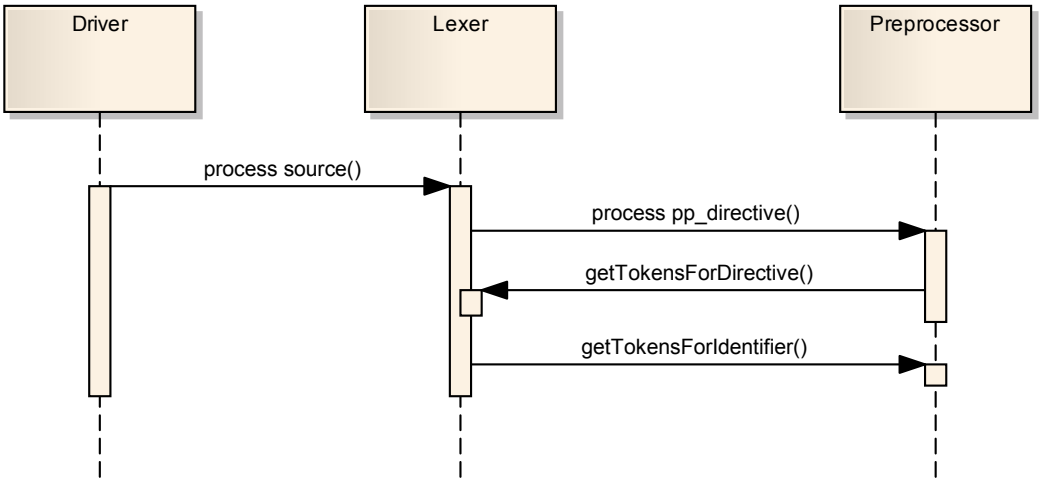

Figure 4: How *clang* compiles a single source file

Listing 1: hello world in C++

```
#include <iostream>
using namespace std;

int main(int argc, char** argv) {
        cout << "hello world" << endl;
}
```

Listing 2: XML AST from generated for hello world

```
<TranslationUnit>
  <!—... includes ... —>
  <UsingDirective file="f48" line="2" col="17" context="_2" name="&lt;using-directive&gt;"
    ref="_1D"/>
  <Function id="_1F6A" file="f48" line="4" col="5" context="_2" name="main" type="_46"
      function_type="_1F6B" num_args="2">
    <ParmVar id="_1F6C" file="f48" line="4" col="14" context="_1F6A" name="argc" type="_46"/>
    <ParmVar id="_1F6D" file="f48" line="4" col="27" context="_1F6A" name="argv" type="_EFB"/>
    <Body>
      <CompoundStmt file="f48" line="4" col="33" endline="6" endcol="1" num_stmts="1">
        <CXXOperatorCallExpr file="f48" line="5" col="2" endcol="27" type="_1B62"
            num_args="2">
          <ImplicitCastExpr file="f48" line="5" col="24" type="_1F6F">

            <DeclRefExpr file="f48" line="5" col="24" type="_1B65" ref="_1B5E"
              name="operator&lt;&lt;"/>
          </ImplicitCastExpr>
          <CXXOperatorCallExpr file="f48" line="5" col="2" endcol="10" type="_1F72"
              num_args="2">
            <ImplicitCastExpr file="f48" line="5" col="7" type="_1F74">
              <DeclRefExpr file="f48" line="5" col="7" type="_1F76" ref="_1F79"
                name="operator&lt;&lt;"/>
            </ImplicitCastExpr>
            <DeclRefExpr file="f48" line="5" col="2" type="_1F5A" ref="_1F58" name="cout"/>
            <ImplicitCastExpr file="f48" line="5" col="10" type="_1B6">
              <StringLiteral file="f48" line="5" col="10" type="_1F7B" value="hello␣world"/>
            </ImplicitCastExpr>
          </CXXOperatorCallExpr>
          <ImplicitCastExpr file="f48" line="5" col="27" type="_1F7D">
            <DeclRefExpr file="f48" line="5" col="27" type="_1F7F" ref="_1F84" name="endl"/>
          </ImplicitCastExpr>
        </CXXOperatorCallExpr>
      </CompoundStmt>
    </Body>
  </Function>
</TranslationUnit>
```

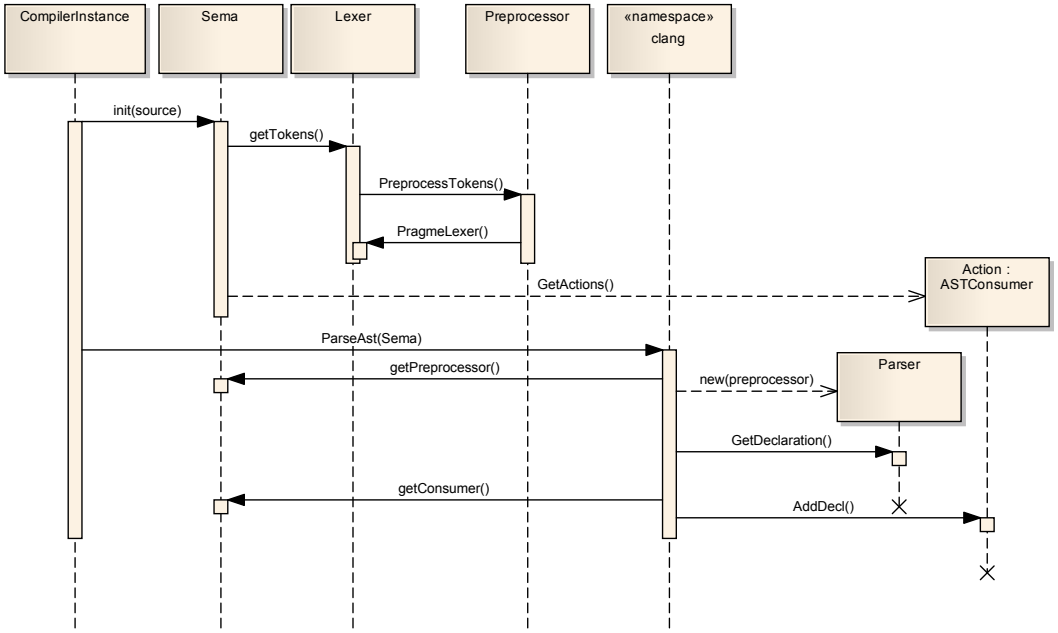

Figure 5: How *clang* compiles a single source file

IdentifierTable is used to look if a name exists in the precompiled header and loads the declaration from the precompiled header.

2.7.2 Symbol lookup

Symbol lookup is only required for identifier expressions. For all other nodes it is enough to check the child nodes. If *clang* encounters an identifier it looks in the DeclContext of the node the Identifier is going to be added.

2.7.3 Function Overloading

A function can have a list of declarations for overloading. If a name already exist for a function *clang* appends an overload declaration to the function declaration found.

Whenever *clang* calls *parseIdentifierExpression* the returned node is not a normal expression referencing the declaration for the name, because the parser doesn't know which overloaded symbol is used. Therefore, an UnresolvedExpression is returned containing the identifier.

After continuing parsing and *clang* finds a parameter list it gets all possible overloads. With the information from the parameter list *clang* is able to create an identifier expression using the correct declaration and returns a correct sub tree without unresolved symbols, see Figure 6.

2.7.4 Argument Dependent Lookup

Argument dependent lookup applies for all functions with an unqualified name and for overloadable operators on classes. Before *clang* can check for a function overloading or determine the type of an expression it has to check in the namespaces and classes for each parameter value. Therefore, *clang* gets the declaration for each parameter expression and looks in the DeclContext of the declaration and if the declaration is also a DeclContext in it as well.

ADL is also performed after creating all sub trees for an expression and checks if ADL applies to the expression. ADL uses function overloading to find the right declaration during the lookup.

2.7.5 Precompiled Headers

Precompiled headers is a feature to spead up compilation by storing headers, which are used in a lot of translation units in a single file. This file has "pch" as its extension.

clang has a binary serializer for its AST. It uses this serializer to save the complete AST for all declarations in the precompiled header. During compilation of a translation unit *clang* uses lazy loading to get the information from the precompiled header as needed.

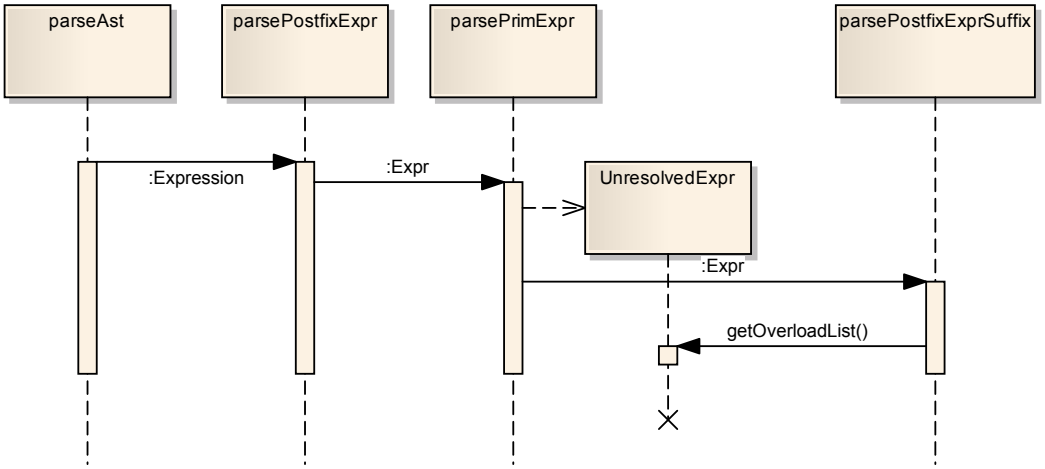

Figure 6: How *clang* is building the AST and does function overloading

3 Integrate *clang* into IDEs

Every modern IDE has its own parser to assist the developer while writing the code. However for C++ the parser from the IDE sometimes parses the code a bit differently than the compiler or doesn't understand the extensions provided by the compiler. *clang* with its library based architecture aims to fill this gap by using the same libraries in the compiler as in the IDE for parsing the code. Therefore, the IDE knows what the compiler would do with the given code.

3.1 Information provided

Normally, one of the first steps in a compiler is to throw away all comments in the code. However, this is a major drawback for IDE tools, since they should keep them during if they modify the code during a refactoring. So compilers and IDE tools have obviously different needs for parsing the source code. *clang* offers libraries and interfaces to make it possible to collect all information needed through interfaces.

3.2 Diagnostics Interface

clang has an interface *DiagnosticClient* to collect warnings and errors report them to the user. The default client used by*clang* produces nice formatted and colored messages for the shell. However, in an IDE this is difficult to handle. A different client could produce messages in a more convenient format for parsing in the IDE to display them nicely in the UI of the IDE. It would also be possible to use the parser in the IDE as a library and use a *DiagnosticClient* which directly displays the messages in the IDE without the need for parsing the output of another process.

3.3 Indexer and *ASTConsumer*

Instead of having a different parser in the IDE it could implement the *ASTConsumer* interface and use the *clang* libraries to parse the source code. The IDE would work on the exactly same AST as the compiler would create.

3.4 Additional Checks and Analysis

A plug-in interface for *FrontendAction*s allows dynamically extending *clang* with additional features. For example an idea for a plug-in could be checking all declaration names to matching naming conventions. The plug-in would use the diagnostic engine provided by*clang* to report any naming violations. No matter if the IDE uses a custom *DiagnosticClient* and parses *clangs* output or uses directly the*clang* libraries the plug-in would work in both cases.

3.5 libclang

clang is under heavy development. Therefore, the API might change in the next couple years. However, to use it in an IDE a stable API would be nice to have. A library with a more stable API for *clang* exists, *libclang*. It doesn't allow access to all information from the parser, but it will be stable over different versions. The goal behind this library

is to provide an interface for IDE developers to access*clang* with a stable API and integrate*clang* into their products. The library offers an indexer for multiple files, a parser to access the AST. It will cache the ASTs for already parsed code files and supports parsing for unsaved documents.

3.6 *clang* vs. CDT vs. C3P0

A lot IDEs already have an infrastructure for parsing and indexing C++ tightly integrated in the IDE. On the other hand*clang* offers a platform for C++ tools like static source analysis. Every tool based on*clang* can be used in the build process and would be automatically integrated into the IDE. However,*clang* only supports C/C++ and Objective-C. IDEs often support more languages. For example Eclipse can be used to develop in almost any language using the appropriate plug-in. However this has some drawbacks in the architecture. The architecture of the C/C++ plug-in called CDT has a more complex AST due to compatibility in with the Eclipse base. I believe that make ASTs compatible for multiple languages and extending them with special nodes is in the end more overhead as maintain multiple independent ASTs for different language families. For example having an AST for JAVA based language one for .net based, one C/C++/ObjC and another one for D. Even though all language shares some similarities it makes it more complicated to implement features for refactoring if there are too many compromises in the AST. The chances to be able using a feature on an AST for multiple languages are almost none. Not even the simplest refactoring like "rename" works on multiple languages.

C3P0 has a more *clang* like goals. C3P0 aims to be focused on only one language and therefore, provide a simpler and easier to use AST. But C3P0 is written in Java. Development of new tools might be faster in Java than in C++ even if there are a lot of libraries covering a lot of functionality. However, *clang* offers a more advanced interface with a lot more functionality than C3P0 currently does. *clang* has already some analysis implemented. *clang* also has the advantage of the LLVM infrastructure. Tools written for analysis on LLVM code can also be used together with *clang*, since it is able to output LLVM code.

3.7 Conclusion

Using a generic AST for language independent analysis is more complex and probably less efficient than using directly the underlying infrastructure like the JVM CLI or LLVM. Therefore, there is no advantage in the CDT-AST. for writing new tools C3P0 or *clang* is the better choice. But which one should be taken depends on the goals of the tool to be implemented. For prototypes to check if an idea could actually make a valuable software C3P0 is just right using ANTLR tree walker or write software in JAVA will get first results sooner. However *clang* offers a more advanced API with more features. It might take a bit longer but the result is probably more valuable because it is easier to integrate in the existing tools if they are based on *clang* and LLVM. The tools written on top of *clang* libraries can also greatly benefit from existing analysis tools. Since it is a wise choice to build new language tools on top of *clang* because it offers a lot of functionality and has a lot of sophisticated base libraries an IDE would greatly benefit integrating *clang* as there API for C++ code assistance or at least an API based on *clang*. The analysis tools can be easily integrated in the IDE and the build process and the IDE could provide a more immediate feedback to the developer about the code he is currently writing. New language features are ready when the compiler supports it. Now the IDE have the problem that new language features have to be implemented in the compiler as well as in the coding assistance. Coupling the IDE tightly to a compiler has the benefit that the IDE exactly knows which features the compiler supports and they are integrated within the IDE as well. *clang* is a good base for building Language tools since it offers a lot of interfaces to a wide range of functionality. The LLVM backend is another big advancement for integrating low level features.

libclang And a lot of interfaces to derive from allow an easy way to get *clang*s information. The diagnostics engine makes it easy to access the compiler warnings and errors. Extending *clang* with plug-ins with tasks to perform on an AST makes it possible to easily integrate new tools in the build process and IDE at once.

3.8 Using libclang index for multiple translation units

libclang offers an index to use a shared context for multiple tranlsation units to be used inside an IDE. Listing 3 shows how to create such an index using the C interface of libclang.

After creating an index for a translation unit it can be used to access its AST and do code completion. Listing 3 uses the *TranslationUnitVisitor* to visit the AST for a precompiled header and a translation unit seperatly.

Listing 3: using clangs interface to build an index for multiple translation units

```
CXChildVisitResult TranslationUnitVisitor(CXCursor cursor,
        CXCursor parent, CXClientData data) {
    return CXChildVisit_Recurse; // visit complete ast recursivly
}

void VisitTranslationUnitWithPCH() {
        // excludeDeclsFromPCH = 1, displayDiagnostics=1
        Idx = clang_createIndex(1, 1);

        // IndexTest.pch was produced with the following command:
        // "clang -x c IndexTest.h -emit-ast -o IndexTest.pch"
        TU = clang_createTranslationUnit(Idx, "IndexTest.pch");

        // This will load all the symbols from 'IndexTest.pch'
        clang_visitChildren(clang_getTranslationUnitCursor(TU),
                            TranslationUnitVisitor, 0);
        clang_disposeTranslationUnit(TU);

        // This will load all the symbols from 'IndexTest.c', excluding symbols
        // from 'IndexTest.pch'.
        char *args[] = { "-Xclang", "-include-pch=IndexTest.pch" };
        TU = clang_createTranslationUnitFromSourceFile(Idx, "IndexTest.c", 2, args,
                                                       0, 0);

        clang_disposeTranslationUnit(TU);
}
```

4 Example

An example to write a *clang* plug-in using its API is added in the source tree for clang. The example for a complete plug-in is called **PrintFunctionNames**.

References

[AS95] Meng Lee Alexander Stepanov. The standard template library. http://www.stepanovpapers.com/STL/DOC.PDF, 1995.

[Ber88] Berkley. Berkley software distribution license. http://www.linfo.org/bsdlicense.html, 1988.

[cla10a] clang. clang: a c language family frontend for llvm. http://clang.llvm.org, 2010.

[cla10b] clang. clang: a list of features clang offers. http://clang.llvm.org/features.html, 2010.

[Com10] C++ Standards Comittee. Working Draft, Standard for Programming Language C++. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2010/n3225.pdf, November 2010.

[Cor10a] Thomas Corbat. Master Thesis, C-Plus-Plus-Parser-for-C++0x. http://sinv-56016.edu.hsr.ch/C3P0Files/MA-C3P0.pdf, December 2010.

[Cor10b] Thomas Corbat. Term Project, C-Plus-Plus-Parser-for-C++0x. http://sinv-56016.edu.hsr.ch/C3P0Files/C3P0.pdf, July 2010.

[EM01] John Gough Erik Meijer. Technical overview of the common language runtime. http://research.microsoft.com/en-us/um/people/emeijer/papers/clr.pdf, 2001.

[GNU87] GNU. Gnu compiler collection. http://gcc.gnu.org/, 1987.

[Gnu07a] Gnu. Gnu general public license. http://www.gnu.org/licenses/gpl.html, 2007.

[Gnu07b] Gnu. Gnu lesser general public license. http://www.gnu.org/licenses/lgpl.html, 2007.

[llv10] llvm. The llvm compiler infrastructure. http://www.llvm.org, 2010.

[MIT88] MIT. Mit license. http://www.opensource.org/licenses/mit-license.php, 1988.

[Stroo] Bjarne Stroustrup. *Die C++-Programiersprache*. Addison-Wesley, 4 edition, 2000.

[TL99] Frank Yellin Tim Lindholm. The java virtual machine specification. http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html, 1999.

[Uoi02] UoI/NCSA. University of illinois/ncsa open source license. http://www.opensource.org/licenses/UoI-NCSA.php, 2002.

Listing 4: Example for a plug-in printing all function names to the output (PrintFunctionNames)

```
namespace {
class PrintFunctionsConsumer : public ASTConsumer {
public:
  virtual void HandleTopLevelDecl(DeclGroupRef DG) {
    for (DeclGroupRef::iterator i = DG.begin(), e = DG.end(); i != e; ++i) {
      const Decl *D = *i;
      if (const NamedDecl *ND = dyn_cast<NamedDecl>(D))
        llvm::errs() << "top-level-decl: \"" << ND->getNameAsString() << "\"\n";
    }
  }
};

class PrintFunctionNamesAction : public PluginASTAction {
protected:
  ASTConsumer *CreateASTConsumer(CompilerInstance &CI, llvm::StringRef) {
    return new PrintFunctionsConsumer();
  }

  bool ParseArgs(const CompilerInstance &CI,
                 const std::vector<std::string>& args) {
    for (unsigned i = 0, e = args.size(); i != e; ++i) {
      llvm::errs() << "PrintFunctionNames arg = " << args[i] << "\n";
      // Example error handling.
      if (args[i] == "-an-error") {
        Diagnostic &D = CI.getDiagnostics();
        unsigned DiagID = D.getCustomDiagID(
          Diagnostic::Error, "invalid argument '" + args[i] + "'");
        D.Report(DiagID);
        return false;
      }
    }
    if (args.size() && args[0] == "help")
      PrintHelp(llvm::errs());

    return true;
  }
  void PrintHelp(llvm::raw_ostream& ros) {
    ros << "Help for PrintFunctionNames plug-in goes here\n";
  }
};
}

static FrontendPluginRegistry::Add<PrintFunctionNamesAction>
X("print-fns", "print function names");
```