2

Clang architecture

In this chapter, we will examine the internal architecture of Clang and its relationship with other LLVM components. We will begin with an overview of the overall compiler architecture, with a specific focus on the clang driver. As the backbone of the compiler, the driver runs all compilation phases and controls their execution. Finally, we will concentrate on the frontend portion of the Clang compiler, which includes lexical and semantic analysis, and produces an Abstract Syntax Tree (AST) as its primary output. The AST forms the foundation for most clang tools, and we will examine it more closely in the next chapters.

The following topics will be covered in this chapter:

- Compiler overview
- Clang driver overview, including an explanation of the compilation phases and their execution
- Clang frontend overview, which covers frontend actions, preprocessor, parser, and sema

2.1 Compilers overview

Despite the fact that compilers are used to translate programs from one form to another, they can also be considered as large software systems that use various algorithms and data structures. The knowledge obtained from studying compilers is not specific to compilers

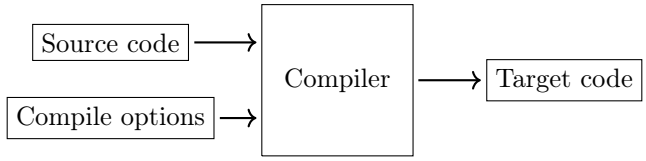

Figure 2.1: Compiler takes source code and compile options and transform them into a code on the target platform

and can be applied to other software as well. On the other hand, compilers are also a subject of active scientific research, and there are many unexplored areas and topics to investigate.

You can find some basic information about the internal structure of a compiler here. We will keep it as basic as possible so the information is applicable to any compiler, not just Clang. We will briefly cover all phases of compilation, which will help to understand Clang's position in the overall compiler architecture.

2.1.1 Compiler workflow

The primary function of a compiler is to convert a program written in a specific programming language (such as C/C++ or FORTRAN) into a format that can be executed on a target platform. This process involves the use of a compiler, which takes the source file and any compilation flags, and produces a build artifact, such as an executable or object file, as shown in fig. 2.1. The term "target platform" can have a broad meaning. It can refer to machine code that is executed on the same host, as is typically the case. But it can also refer to cross-compilation, where the compiler generates code for a different computer architecture than the host. For example, code for a mobile application or embedded application running on ARM can be generated using an Intel machine as the host. Additionally, the target platform is not limited to machine code only. For example, some early C++ compilers (such as "cc") would produce pure C code as output. This was done because, at the time, C was the most widely used and well-established programming language, and the C compiler was the most reliable way to generate machine code. This approach allowed early C++ programs to be run on a wide range of platforms since most systems already had a C compiler available. The produced C code could then be compiled into binary by another compiler.

We are going to focus on compilers that produce binary code, and a typical compiler workflow for such a compiler is shown in fig. 2.2. The stages of compilation can be described as follows:

- Frontend: The Frontend does lexical analysis and parsing, which includes both syntax analysis and semantic analysis. The syntax analysis assumes that your program is

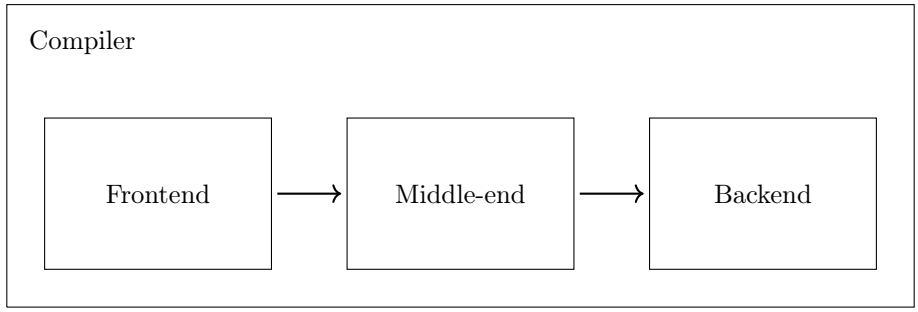

Figure 2.2: Typical compiler workflow: source program is passed via different stages: frontend, middle-end and backend

well-organized according to the language grammar rules. The semantic analysis performs checks on the program's meaning and rejects invalid programs, such as those that use wrong types.

- Middle-end: The Middle-end performs various optimizations on the intermediate representation (IR) code (LLVM-IR for clang).

- Backend: The Backend of a compiler takes the optimized or transformed IR and generates machine code or assembly code that can be executed by the target platform.

The source program is transformed into different forms as it passes through the various stages. For example, the Frontend produces IR code, which is then optimized by the Middle-end and finally converted into native code by the Backend (see fig. 2.3).

2.1.2 Frontend

Our primary focus will be on the frontend, so we will examine its components. The frontend also transforms the source code into various forms before it produces the IR. The first component of the frontend is the Lexer (see fig. 2.4). It converts the source code into a set of tokens, which are used to create a special data structure called the abstract syntax tree (AST). The final component, code generator (Codegen), traverses the AST and generates the IR from it.

We will use a simple C/C++ program that calculates the maximum of two numbers to demonstrate the workings of the Frontend. The code for the program is as follows:

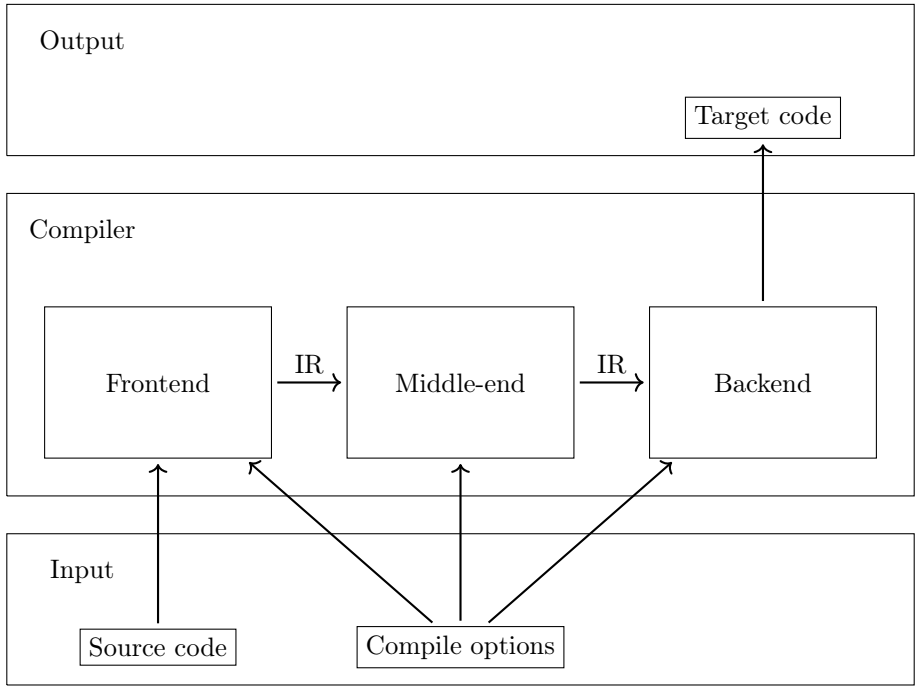

Figure 2.3: Source code transformation by compiler: Input data consists of Source code and Compile options. The source code is transformed by Frontend into IR (Intermediate representation). Middle-end does different optimizations on IR and passes the final (optimazed) result to Backend. Backend generates the Target code. Frontend, Middle-end and Backend use Compile options as setting for the code transformations

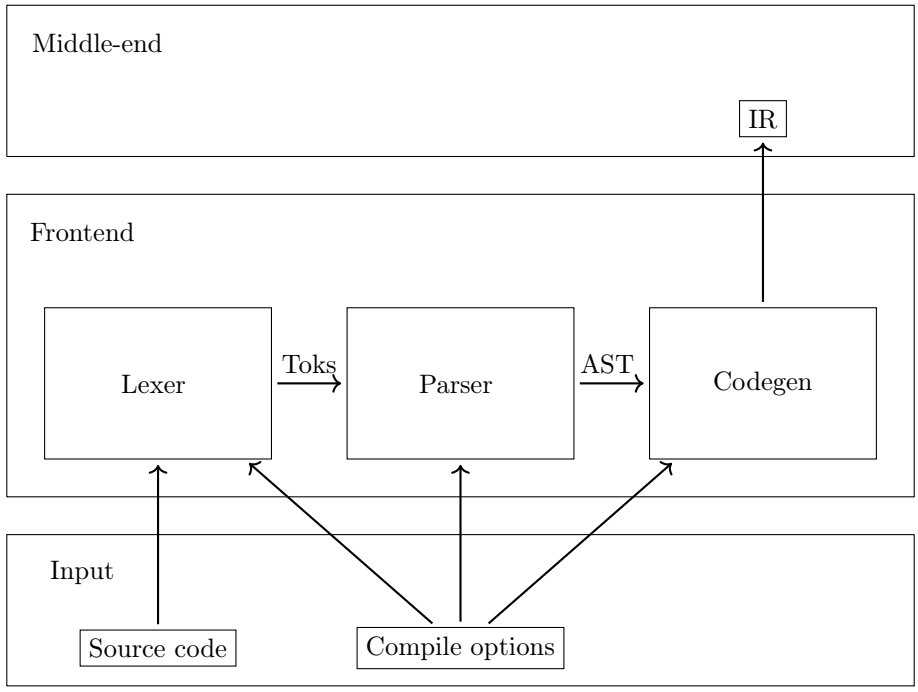

Figure 2.4: Compiler frontend: source code is transformed into a set of tokens (Toks) by Lexer . Parser takes the tokens and creates Abstract syntax tree (AST). Codegen generates IR from AST

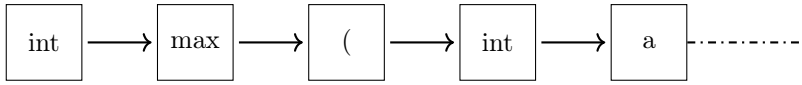

Figure 2.6: Lexer : the program source is converted into a stream of tokens

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

Figure 2.5: Test program for compiler frontend investigations

2.1.2.1 Lexer

The Frontend process starts with the Lexer , which converts the input source into a stream of tokens. In our example program (see fig. 2.5), the first token is the keyword `int` , which represents the integer type. This is followed by the identifier `max` for the function name. The next token is the left parenthesis `(` , and so on (see fig. 2.6).

2.1.2.2 Parser

The Parser is the next component following the Lexer . The primary output produced by the Parser is called as abstract syntax tree (AST). This tree represents the abstract syntactic structure of the source code written in a programming language. The Parser generates the AST by taking the stream of tokens produced by the Lexer as input and organizing them into a tree-like structure. Each node in the tree represents a construct in the source code, such as a statement or expression, and the edges between nodes represent the relationships between these constructs.

The AST for our example program is shown in fig. 2.7. As you can see, our function (`max`) has two parameters (`a` and `b`) and a body. The body is marked as a compound statement in fig. 2.7. The compound statement consists of other statements, such as `return` and `if` . The variables `a` and `b` are used in the bodies of these statements. You may also be interested in the real AST generated by Clang for the compound statement, the result of which is shown in fig. 2.8.

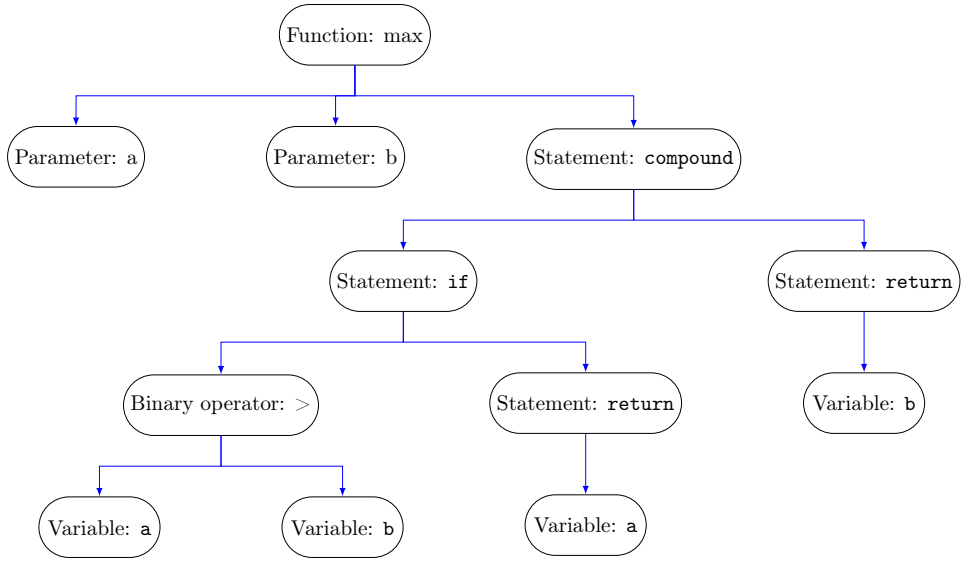

Figure 2.7: The AST for our example program that calculates maximum of 2 numbers

The Parser performs two activities:

1. Syntax analysis: the Parser constructs the AST by analyzing the syntax of the program.
2. Semantic analysis: the Parser analyzes the program semantically.

One of the primary goals of analysis is error detection, and the Parser produces an error message if it fails in syntax or semantic analysis. We can get a sense of this by considering what types of errors are detected by syntax analysis and which ones are detected by semantic analysis.

Syntax analysis assumes that the program should be correct in terms of the grammar specified for the language. For example, the following program is invalid in terms of syntax because a semicolon is missing at the last return statement:

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

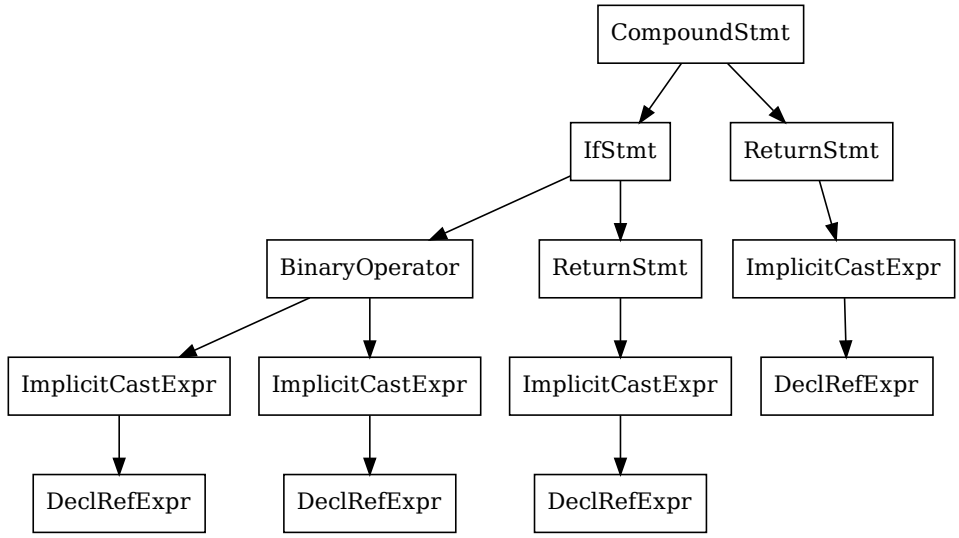

Figure 2.8: The AST for compound statement generated by Clang . The tree generated by *clang -cc1 -ast-view <...>* command

Clang produces the following output for the program:

```
max_invalid_syntax.cpp:4:11: error: expected ';' after return statement
  return b // missing ;
          ^
          ;
```

On the other hand, a program can be syntactically correct but have no sense. The Parser should detect a semantic error in such cases. For instance, the following program has a semantic error related to the wrongly used type for the return value:

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

Clang produces the following output for the program:

```
max_invalid_sema.cpp:4:10: error: cannot initialize return object of type \
'int' with an rvalue of type 'int *'
  return &b; // invalid return type
         ^~
```

AST is mainly constructed as a result of syntax analysis, but for certain languages, such as C++, semantic analysis is also crucial for constructing the AST, particularly for C++ template instantiation.

During syntax analysis, the compiler verifies that the template declaration adheres to the language's grammar and syntax rules, including the proper use of keywords such as "template" and "typename", as well as the formation of the template parameters and body.

Semantic analysis, on the other hand, involves the compiler performing template instantiation, which generates the AST for specific instances of the template. It's worth noting that the semantic analysis of templates can be quite complex, as the compiler must perform tasks such as type checking, name resolution, and more for each template instantiation. Additionally, the instantiation process can be recursive and lead to a significant amount of code duplication, known as code bloat. To combat this, C++ compilers employ techniques such as template instantiation caching to minimize the amount of redundant code generated.

2.1.2.3 Codegen

The Codegen [1] or code generator, which is the final component of the compiler's frontend, has the primary goal of generating the Intermediate Representation (IR). For this purpose, the compiler traverses the AST generated by the parser and converts it into another source code that is called the Intermediate Representation or IR. The IR is a language-independent representation, allowing the same middle-end component to be used for different frontends (FORTRAN vs C++).

The use of Intermediate Representations (IRs) in compilers is a concept that has been around for several decades. The idea of using an intermediate representation to represent the source code of a program during compilation has evolved over time, and the exact date when IR was first introduced in compilers is not clear.

However, it is known that the first compilers in the 1950s and 1960s did not use IRs and instead translated source code directly into machine code. By the 1960s and 1970s, researchers had begun experimenting with using IRs in compilers to improve the efficiency and flexibility of the compilation process.

One of the first widely used IRs was the three-address code, which was used in the mid-1960s in IBM/360's FORTRAN compiler. Other early examples of IRs include the register transfer language (RTL) and the static single assignment (SSA) form, which were introduced in the 1970s and 1980s respectively.

Today, the use of IRs in compilers is a standard practice, and many compilers use multiple IRs throughout the compilation process. This allows for more powerful optimization and code generation techniques to be applied.

2.2 Clang driver overview

When discussing compilers, we typically refer to a command-line utility that initiates and manages the compilation process. For example, to use the GNU Compiler Collection, one must call `gcc` to start the compilation process. Similarly, to compile a C++ program using Clang, one must call `clang` as the compiler. The program that controls the compilation process is known as the driver. The driver coordinates different stages of compilation and connects them together. In the book, we will be focusing on LLVM and using clang as the driver for the compilation process.

It may be confusing for readers that the same word, "clang," is used to refer to both the compiler front-end and the compilation driver. In contrast, with GCC, the driver and C++ compiler are separate executables [2]. However, "clang" is a single executable that functions as both the driver and the compiler front-end. To use Clang as the compiler front-end only,

[1] It's worth mentioning that we also have another Codegen component as a part of Backend that generate the target code.

[2] The C/C++ compiler in GCC is a separate executable called "cc"

the special option `-cc1` must be passed to it.

2.2.1 Example program

We will use the simple "Hello world!" example program for our experiments with clang driver. The main source file is called as `hello.cpp` . The file implements a trivial C++ program that prints "Hello world!" to the standard output:

```
1 #include <iostream>
2
3 int main() {
4   std::cout << "Hello world!" << std::endl;
5   return 0;
6 }
```

You can compile the source with

```
1 $ clang hello.cpp -o /tmp/hello -lstdc++
```

Figure 2.9: Compilation for *`hello.cpp`* : We use *`clang`* executable as the compiler. We also use standard C++ library i.e. we link the executable with *`-lstdc++`* . The result is stored at */tmp/hello*

As you may see, we specified `-lstc++` library option because we used `<iostream>` header from the standard C++ library. We also specified the output for executable (`/tmp/hello`) with `-o` option.

2.2.2 Compilation phases

We used 2 inputs for our example program. The first one is our source code, the second one is a shared library for standard C++ library. The clang driver should combine the inputs together, pass them via different phases of compilation process and finally provide the executable file on the target platform.

Clang uses the same process as shown in fig. 2.2. You can ask Clang to show the phases using `-ccc-print-phases` additional argument

```
$ clang hello.cpp -o /tmp/hello -lstdc++ -ccc-print-phases
```

The output for the command is the following

```
              +- 0: input, "hello.cpp", c++
           +- 1: preprocessor, {0}, c++-cpp-output
        +- 2: compiler, {1}, ir
     +- 3: backend, {2}, assembler
  +- 4: assembler, {3}, object
  |- 5: input, "1%dM", object
  6: linker, {4, 5}, image
```

We can visualize the output as shown in fig. 2.10. As we can see in fig. 2.10, the driver receives an input file `hello.cpp` , which is a C++ file. The file is processed by the preprocessor, and we obtain the preprocessor output (marked as `c++-cpp-output`). The result is compiled into IR form by the compiler, and then the backend converts it into assembly form. This form is later transformed into an object file. The final object file is combined with another object (`libstdc++`) to produce the final binary (`image`).

2.2.3 Tools execution

The phases are combined into several tool executions. The Clang driver invokes different programs to produce the final executable. Specifically, for our example, it calls the `clang` compiler and the `ld` linker. Both programs require additional arguments that are set up by the driver.

For instance, our example program (`hello.cpp`) includes the following header:

```
1 #include <iostream>
2 ...
```

We have not specified any additional arguments (such as search paths, for example `-I`) when we invoked the compilation. However, different architectures and operating systems might have different paths for locating headers.

On Fedora 37, the header is located in the `/usr/include/c++/12/iostream` folder. We can examine a detailed description of the process executed by the driver and the arguments used with the `-###` option:

```
$ clang hello.cpp -o /tmp/hello -lstdc++ -###
```

The output for this command is quite extensive, and certain parts have been omitted here. Please refer to fig. 2.11.

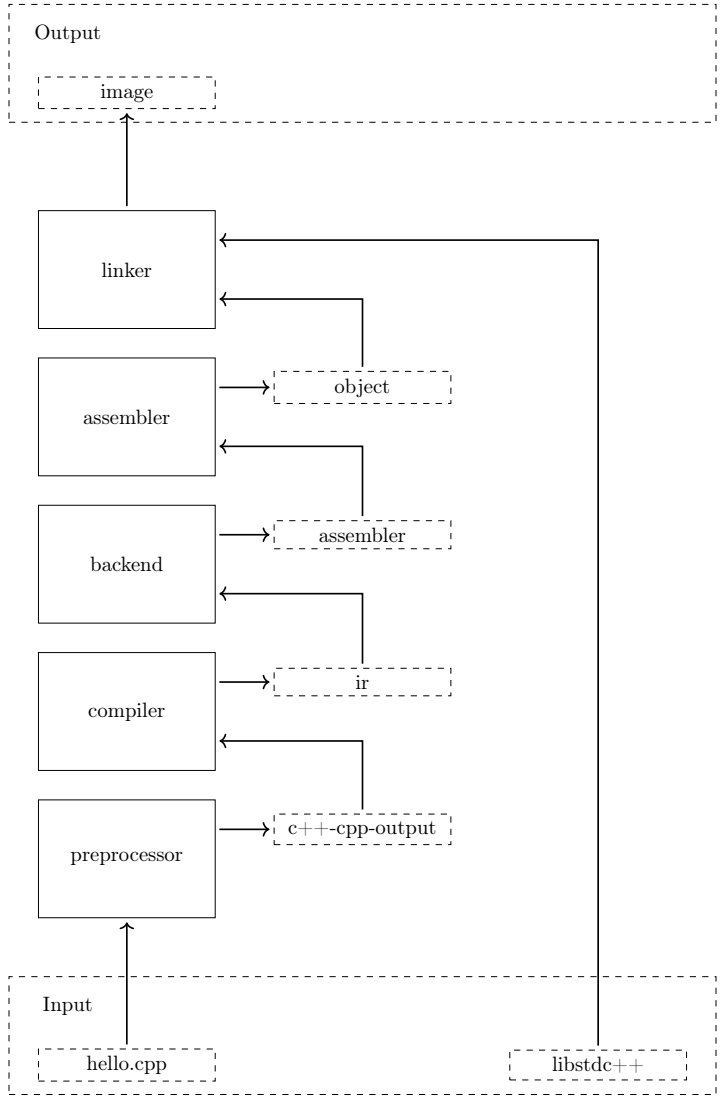

Figure 2.10: Clang driver phases

```
1  clang version 16.0.0 (https://github.com/llvm/llvm-project.git ...)
2   "<...>/llvm-project/build/bin/clang-16"
3      "-cc1" ... \
4      "-internal-isystem" \
5      "/usr/include/c++/12" ... \
6      "-internal-isystem" \
7      "/usr/include/c++/12/x86_64-redhat-linux" ... \
8      "-internal-isystem" ... \
9      "<...>/llvm-project/build/lib/clang/16/include" ... \
10     "-internal-externc-isystem" \
11     "/usr/include" ... \
12     "-o" "/tmp/hello-XXX.o" "-x" "c++" "hello.cpp"
13  ".../bin/ld" ... \
14     "-o" "/tmp/hello" ... \
15     "/tmp/hello-XXX.o" \
16     "-lstdc++" ...
```

Figure 2.11: Clang driver, tools execution, the host system is Fedora 37.

As we can see in fig. 2.11, the driver initiates two processes: `clang-16` with the `-cc1` flag (see lines 2-12) and the linker `ld` (see lines 13-16). The clang compiler implicitly receives several search paths, as seen in lines 5, 7, 9 and 11. These paths are necessary for the inclusion of the iostream header in the test program. The output of the first executable (`/tmp/hello-XXX.o`) serves as input for the second one (see lines 12 and 15). The arguments `-lstdc++` and `-o /tmp/hello` are set for the linker, while the first argument (hello.cpp) is provided for the compiler invocation (first executable).

The process can be visualized as shown in fig. 2.12, where we can see that two executables are executed as part of the compilation process. The first one is `bin/clang-16` with a special flag (`-cc1`). The second one is the linker: `bin/ld` .

2.2.4 Combine all together

We can summarize the knowledge we have acquired so far using fig. 2.13. The figure illustrates two different processes started by the clang driver. The first one is `clang -cc1` (compiler), and the second one is `ld` (linker). The compiler process is the same executable as the clang driver (`clang`), but it is run with a special argument: `-cc1` . The compiler produces an object file that is then processed by the linker (`ld`) to generate the final bi-

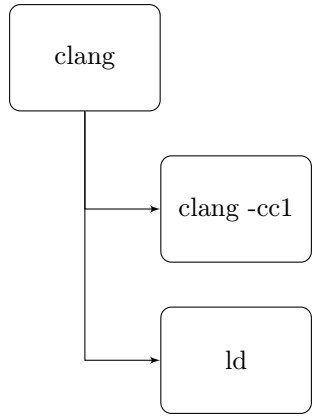

Figure 2.12: Clang driver, tools execution: clang driver runs 2 executables: The clang executable with -cc1 flag and linker - ld executable

nary.

In fig. 2.13, we can observe similar components of the compiler mentioned earlier (see Section 2.1, Compilers overview). However, the main difference is that the preprocessor (part of the lexer) is shown separately, while the frontend and middle-end are combined into the compiler. Additionally, the figure depicts an assembler that is executed by the driver to generate the object code. It is important to note that the assembler can be integrated, as shown in fig. 2.13, or it may require a separate process to be executed.

> Here is an example of specifying an external assembler using the `-c` (compile only) and `-o` (output file) options, along with the appropriate flags for your platform:
>
> ```
> $<...>/llvm-project/build/bin/clang -c hello.cpp -o /tmp/hello.o
> as -o /tmp/hello.o /tmp/hello.s
> ```

2.2.5 Debugging clang

We're going to step through a debugging session for our compilation process, illustrated in fig. 2.9. Our chosen point of interest, or breakpoint, is the `clang::ParseAST` function. In a typical debug session, which resembles the one outlined in fig. 1.4, you would feed command-line arguments following the `--` symbol. The command should look like this:

```
$lldb <...>/llvm-project/build/bin/clang -- hello.cpp -o /tmp/hello -lstdc++
```

In this case, `<...>` represents the directory path used to clone the LLVM project.

Unfortunately, this approach doesn't work with the Clang compiler:

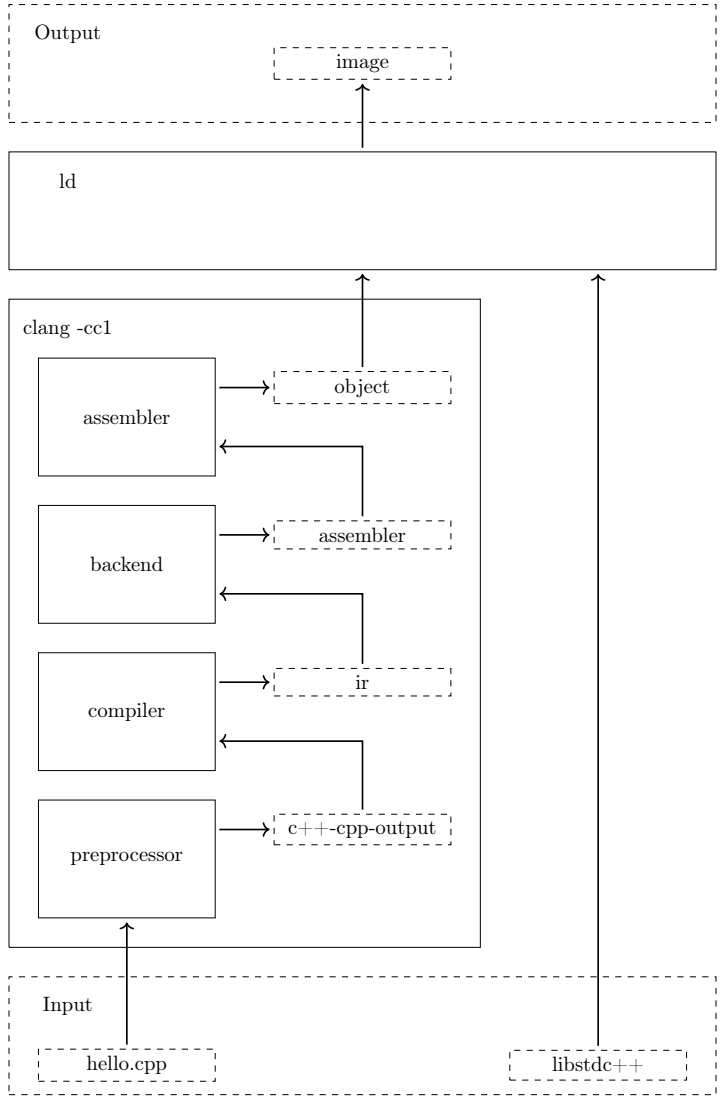

Figure 2.13: Clang driver: The driver got input file ***hello.cpp*** that is a C++ file. It starts 2 processes: ***clang*** and ***ld*** . The first one does real compilation and starts the integrated assembler. The last one is the linker (***ld***) that produces the final binary (***image***) from the result got from compiler and external library (***libstdc++***)

```
1 $ lldb <...>/llvm-project/build/bin/clang -- src/part1/ch2_arch/hello.cpp -o
   ↪ /tmp/hello.o -lstdc++
2 ...
3 (lldb) b clang::ParseAST
4 ...
5 (lldb) r
6 ...
7 2 locations added to breakpoint 1
8 ...
9 Process 247135 stopped and restarted: thread 1 received signal: SIGCHLD
10 Process 247135 stopped and restarted: thread 1 received signal: SIGCHLD
11 Process 247135 exited with status = 0 (0x00000000)
12 (lldb)
```

As we can see from line 7, the breakpoint was set but the process finished successfully (line 11) without any interruptions. In other words, our breakpoint didn't trigger in this instance.

Understanding the internals of clang-driver can help us identify the problem at hand. As mentioned earlier, the clang executable acts as a driver in this context, running two separate processes (refer to fig. 2.12). Therefore, if we wish to debug the compiler, we need to run it using the `-cc1` option.

> It's worth mentioning a certain optimization implemented in clang in 2019 [Ganea", "2019"]. When using the `-c` option, the clang driver doesn't spawn a new process for the compiler:
>
> ```
> $<...>/llvm-project/build/bin/clang -c hello.cpp -o /tmp/hello.o -###
> clang version 16.0.0 ...
> InstalledDir: <...>/llvm-project/build/bin
>  (in-process)
>  "<...>/llvm-project/build/bin/clang-16" "-cc1" ... "hello.cpp"
>  ...
> ```
>
> As shown above, the clang driver does not spawn a new process and instead calls the "cc1" tool within the same process. This feature not only improves the compiler's performance but can also be leveraged for clang debugging.

Upon using `-cc1` option and excluding the `-lstdc++` option (which is specific to the second process, the ld linker), the debugger will generate the following output:

```
1  $ lldb <...>/llvm-project-16/build/bin/clang -- -cc1 hello.cpp -o
     ↪ /tmp/hello.o
2  ...
3  (lldb) b clang::ParseAST
4  ...
5  (lldb) r
6  ...
7  2 locations added to breakpoint 1
8  Process 249890 stopped
9  * thread #1, name = 'clang', stop reason = breakpoint 1.1
10     frame #0: 0x00007fffe803eae0 ... at ParseAST.cpp:116:3
11    113
12    114  void clang::ParseAST(...) {
13    115    // Collect global stats on Decls/Stmts ...
14 -> 116    if (PrintStats) {
15    117      Decl::EnableStatistics();
16    118      Stmt::EnableStatistics();
17    119    }
18 (lldb) c
19 Process 249890 resuming
20 hello.cpp:1:10: fatal error: 'iostream' file not found
21 #include <iostream>
22          ^~~~~~~~~~
23 1 error generated.
24 Process 249890 exited with status = 1 (0x00000001)
25 (lldb)
```

Thus we can see that we were able to successfully set the breakpoint, but the process ended with an error (see lines 20-24). This error arose because we omitted certain search paths, which are typically appended implicitly by the clang driver, necessary to find all the includes required for successful compilation.

We can successfully execute the process if we explicitly include all necessary arguments in the compiler invocation. Here's how to do that:

```
$ lldb <...>/llvm-project/build/bin/clang -- -cc1 -internal-isystem
  ↪ /usr/include/c++/12 -internal-isystem
  ↪ /usr/include/c++/12/x86_64-redhat-linux -internal-isystem
  ↪ <...>/llvm-project/build/lib/clang/16/include -internal-externc-isystem
  ↪ /usr/include hello.cpp -o /tmp/hello.o
...
(lldb) b clang::ParseAST
...
(lldb) r
...
2 locations added to breakpoint 1
Process 251736 stopped
* thread #1, name = 'clang', stop reason = breakpoint 1.1
    frame #0: 0x00007fffe803eae0 ... at ParseAST.cpp:116:3
   113
   114  void clang::ParseAST(...) {
   115    // Collect global stats on Decls/Stmts ...
-> 116    if (PrintStats) {
   117      Decl::EnableStatistics();
   118      Stmt::EnableStatistics();
   119    }
(lldb) c
Process 251736 resuming
Process 251736 exited with status = 0 (0x00000000)
(lldb)
```

In conclusion, we have successfully demonstrated the debugging of a clang compiler invocation. The techniques presented can be effectively employed for exploring the internals of a compiler and addressing compiler-related bugs.

2.3 Clang frontend overview

It's evident that the clang compiler toolchain conforms to the pattern widely described in various compiler books [Cooper and Torczon, 2012]. However, the clang's frontend part diverges significantly from a typical compiler frontend. The primary reason for this distinction is the complexity of the C++ language. Some features, like macros, can modify the

source code itself, while others, like typedef, can influence the kind of token. Clang can also generate output in a variety of formats. For instance, the following command generates an aesthetically pleasing HTML view of the program shown in fig. 2.5:

```
$ clang -cc1 -emit-html max.cpp
```

Take note that we pass the argument to emit the HTML form of the source program to the clang frontend, specified with the `-cc1` option. Alternatively, you can pass an option to the frontend via the `-Xclang` option, which requires an additional argument representing the option itself. For example:

```
$ clang -fsyntax-only -Xclang -emit-html max.cpp
```

You may notice that in the command above, we utilized the `-fsyntax-only` option, instructing Clang to only execute the preprocessor, parser, and semantic analysis stages.

Accordingly, we can instruct the Clang frontend to perform different actions and produce varying types of output based on the provided compilation options. The base class for these actions is termed `FrontendAction` .

2.3.1 Frontend action

The Clang frontend is capable of executing only one frontend action at a time. A frontend action is a specific task or process that the frontend performs based on the provided compiler option. Below is a list of some possible frontend actions (the table only includes a subset of the available frontend actions):

FrontendAction	Compiler option	Description
EmitObjAction	`-emit-obj` (default)	Compile to an object file
EmitBCAction	`-emit-llvm-bc`	Compile to LLVM bytecode
EmitLLVMAction	`-emit-llvm`	Compile to LLVM readable form
ASTPrintAction	`-ast-print`	Build ASTs and then pretty-print them.
HTMLPrintAction	`-emit-html`	Prints the program source in HTML form
DumpTokensAction	`-dump-tokens`	Prints preprocessor tokens

Table 2.1: Frontend actions

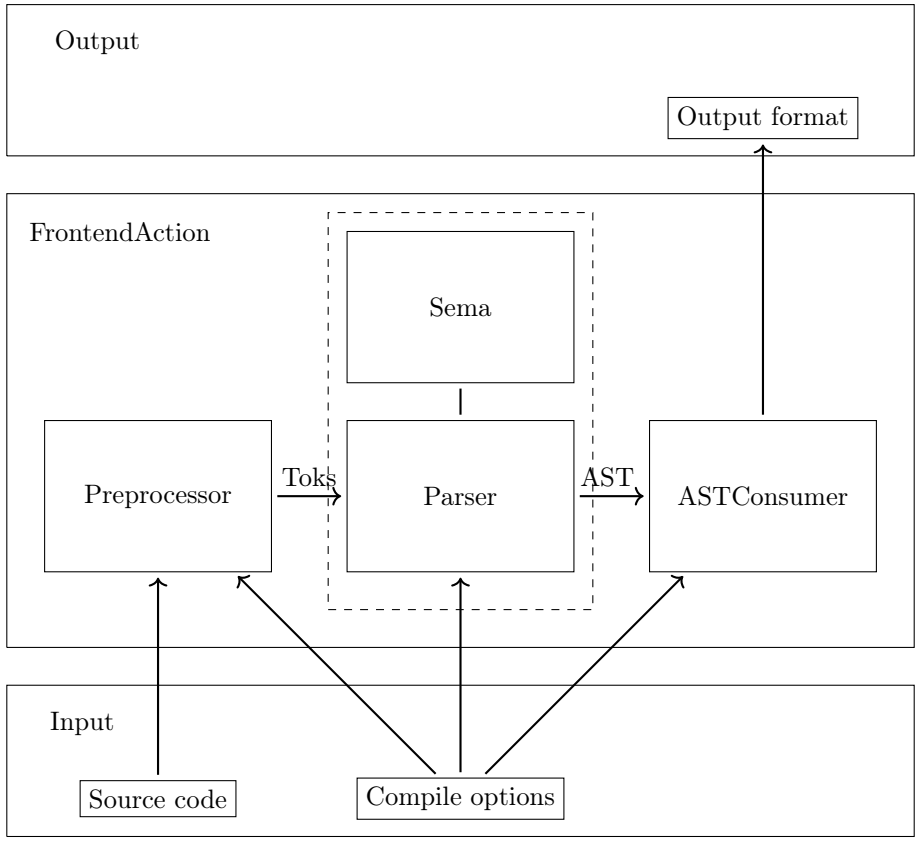

Figure 2.14: Clang frontend components

The diagram shown in fig. 2.14 illustrates the basic frontend architecture, which is similar to the architecture shown in fig. 2.4. However, there are notable differences specific to Clang.

One significant change is the naming of the lexer. In Clang, the lexer is referred to as the Preprocessor. This naming convention reflects the fact that the lexer implementation is encapsulated within the `Preprocessor` class. This alteration was inspired by the unique aspects of the C/C++ language, which includes special types of tokens (macros) that require specialized preprocessing.

Another noteworthy deviation is found in the parser component. While conventional compilers typically perform both syntax and semantic analysis within the parser, Clang distributes these tasks across different components. The `Parser` component focuses solely on syntax analysis, while the `Sema` component handles semantic analysis.

Furthermore, Clang offers the ability to produce output in different forms or formats. For example, the `CodeGenAction` class serves as the base class for various code generation actions, such as `EmitObjAction` or `EmitLLVMAction`.

We will use the code for `max` function from fig. 2.5 for our future exploration of the Clang frontend's internals:

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

By utilizing the `-cc1` option, we can directly invoke the Clang frontend, bypassing the driver. This approach allows us to examine and analyze the inner workings of the Clang frontend in greater detail.

2.3.2 Preprocessor

The first part is the Lexer that is called as Preprocessor in Clang. Its primary goal is to convert the input program into a stream of tokens. You can print the token stream using the `-dump-tokens` options as follows

```
$ clang -cc1 -dump-tokens max.cpp
```

The output of the command is as shown below:

```
int 'int'        [StartOfLine]  Loc=<max.cpp:1:1>
identifier 'max'          [LeadingSpace] Loc=<max.cpp:1:5>
l_paren '('              Loc=<max.cpp:1:8>
int 'int'                Loc=<max.cpp:1:9>
identifier 'a'   [LeadingSpace] Loc=<max.cpp:1:13>
comma ','                Loc=<max.cpp:1:14>
int 'int'        [LeadingSpace] Loc=<max.cpp:1:16>
identifier 'b'   [LeadingSpace] Loc=<max.cpp:1:20>
r_paren ')'              Loc=<max.cpp:1:21>
l_brace '{'      [LeadingSpace] Loc=<max.cpp:1:23>
if 'if'  [StartOfLine] [LeadingSpace]   Loc=<max.cpp:2:3>
l_paren '('      [LeadingSpace] Loc=<max.cpp:2:6>
identifier 'a'           Loc=<max.cpp:2:7>
greater '>'      [LeadingSpace] Loc=<max.cpp:2:9>
identifier 'b'   [LeadingSpace] Loc=<max.cpp:2:11>
```

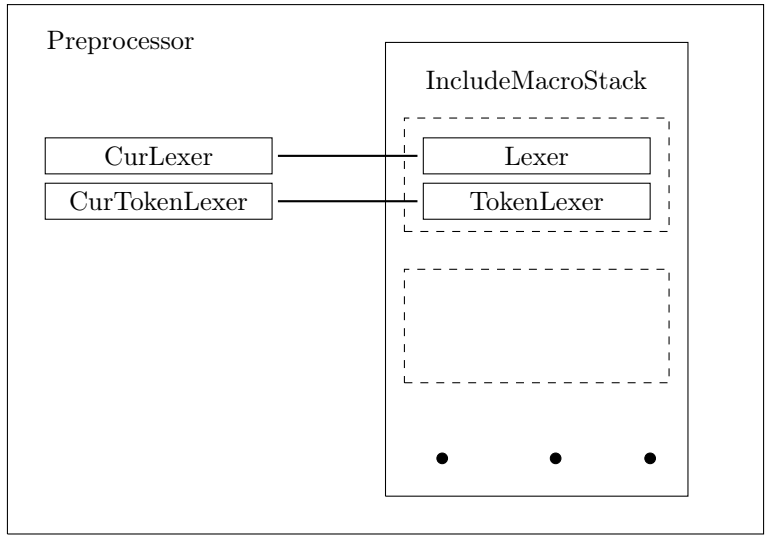

Figure 2.15: Preprocessor (clang lexer) class internals

```
r_paren ')'             Loc=<max.cpp:2:12>
return 'return'  [StartOfLine] [LeadingSpace]   Loc=<max.cpp:3:5>
identifier 'a'   [LeadingSpace] Loc=<max.cpp:3:12>
semi ';'                Loc=<max.cpp:3:13>
return 'return'  [StartOfLine] [LeadingSpace]   Loc=<max.cpp:4:3>
identifier 'b'   [LeadingSpace] Loc=<max.cpp:4:10>
semi ';'                Loc=<max.cpp:4:11>
r_brace '}'      [StartOfLine]  Loc=<max.cpp:5:1>
eof ''          Loc=<max.cpp:5:2>
```

As we can see, there are different types of tokens, such as language keywords (e.g., `int`, `return`), identifiers (e.g., `max`, `a`, `b`, etc.), and special symbols (e.g., semicolon, comma, etc.). The tokens for our small program are called normal tokens, which are returned by the lexer.

In addition to normal tokens, Clang has an additional type of token called annotation tokens. The primary difference is that these tokens also store additional semantic information. For instance, a sequence of normal tokens can be replaced by the parser with a single annotation token that contains information about the type or C++ scope. The primary reason for using such tokens is performance, as it allows for the prevention of reparsing when the parser needs to backtrack.

C/C++ language has some specifics that influence the internal implementation of the `Preprocessor` class. The first one is about macros. The `Preprocessor` class has two different helper classes to retrieve tokens:

- The `Lexer` class is used to convert a text buffer into a stream of tokens.
- The `TokenLexer` class is used to retrieve tokens from macro expansions.

It should be noted that only one of these helpers can be active at a time.

Another specific aspect of C/C++ is the `#include` directive[3]. In this case, we need to maintain a stack of includes, where each include can have its own `TokenLexer` or `Lexer`, depending on whether there is a macro expansion within it. As a result, the `Preprocessor` class keeps a stack of lexers (`IncludeMacroStack` class) for each `#include` directive, as shown in fig. 2.15.

2.3.3 Parser and Sema

The Parser and Sema are crucial components of the Clang compiler frontend. They handle the syntax and semantic analysis of the source code, producing an AST as output. This tree can be visualized for our test program using the command:

```
$ clang -cc1 -ast-dump max.cpp
```

The output of this command is shown below

```
TranslationUnitDecl 0xc4b578 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0xc4bde0 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
| `-BuiltinType 0xc4bb40 '__int128'
...
`-FunctionDecl 0xc91580 <max.cpp:1:1, line:5:1> line:1:5 max 'int (int, int)'
  |-ParmVarDecl 0xc91428 <col:9, col:13> col:13 used a 'int'
  |-ParmVarDecl 0xc914a8 <col:16, col:20> col:20 used b 'int'
  `-CompoundStmt 0xc917b8 <col:23, line:5:1>
    |-IfStmt 0xc91750 <line:2:3, line:3:12>
    | |-BinaryOperator 0xc916e8 <line:2:7, col:11> 'bool' '>'
    | | |-ImplicitCastExpr 0xc916b8 <col:7> 'int' <LValueToRValue>
    | | | `-DeclRefExpr 0xc91678 <col:7> 'int' lvalue ParmVar 0xc91428 'a' 'int'
    | | `-ImplicitCastExpr 0xc916d0 <col:11> 'int' <LValueToRValue>
    | |   `-DeclRefExpr 0xc91698 <col:11> 'int' lvalue ParmVar 0xc914a8 'b' 'int'
    | `-ReturnStmt 0xc91740 <line:3:5, col:12>
    |   `-ImplicitCastExpr 0xc91728 <col:12> 'int' <LValueToRValue>
    |     `-DeclRefExpr 0xc91708 <col:12> 'int' lvalue ParmVar 0xc91428 'a' 'int'
    `-ReturnStmt 0xc917a8 <line:4:3, col:10>
      `-ImplicitCastExpr 0xc91790 <col:10> 'int' <LValueToRValue>
        `-DeclRefExpr 0xc91770 <col:10> 'int' lvalue ParmVar 0xc914a8 'b' 'int'
```

Clang utilizes a hand-written recursive-descent parser [Community, 2023a]. This parser can be considered simple, and this simplicity was one key reason for its selection. Additionally,

[3] which is also applicable to the import directive

the complex rules specified for the C/C++ languages necessitated an ad-hoc parser with easily adaptable rules.

Let's explore how this works with our example. Parsing begins with a top-level declaration known as a `TranslationUnitDecl`, representing a single translation unit. The C++ standard defines a translation unit as follows [for Standardization, 2020, lex.separate]:

> A source file together with all the headers (16.5.1.2) and source files included (15.3) via the preprocessing directive #include, less any source lines skipped by any of the conditional inclusion (15.2) preprocessing directives, is called a translation unit.

The parser first recognizes that the initial tokens from the source code correspond to a function definition, as defined in the C++ standard [for Standardization, 2020, dcl.fct.def.general]:

```
function-definition :
        ... declarator ... function-body
        ...
```

The corresponding code is below

```
1 int max(...) {
2   ...
3 }
```

The function definition necessitates a declarator and function body. We'll start with the declarator, defined in the C++ standard as [for Standardization, 2020, dcl.decl.general]:

```
declarator:
        ...
        ... parameters-and-qualifiers ...
...
parameters-and-qualifiers:
        ( parameter-declaration-clause ) ...
...
parameter-declaration-clause:
        parameter-declaration-list ...
parameter-declaration-list:
        parameter-declaration
        parameter-declaration-list , parameter-declaration
```

In other words, the declarator specifies a list of parameter declarations within brackets. The corresponding piece of code from the source is as follows:

```
1 ... (int a, int b)
2   ...
```

The function definition, as stated above, also requires a function body. The C++ standard specifies the function body as follows: [for Standardization, 2020, dcl.fct.def.general]

```
function-body:
        ... compound-statement
        ...
```

Thus the function body consists of a compound statement, which is defined as follows in the C++ standard [for Standardization, 2020, stmt.block]

```
compound-statement:
        { statement-seq ... }
statement-seq:
        statement
        statement-seq statement
```

Therefore, it describes a sequence of statements enclosed within `{...}` brackets.

Our program has two types of statements: the conditional (if) statement and the return statement. These are represented in the C++ grammar definition as follows [for Standardization, 2020, stmt.pre]:

```
statement:
        ...
        selection-statement
        ...
        jump-statement
        ...
```

In this context, the selection statement corresponds to the `if` condition in our program, while the jump statement corresponds to the `return` operator.

Let's examine the jump statement in more detail [for Standardization, 2020, stmt.jump.general]:

```
jump-statement:
        ...
        return expr-or-braced-init-list;
        ...
```

where expr-or-braced-init-list is defined as [for Standardization, 2020, dcl.init.general]:

```
expr-or-braced-init-list:
        expression
        ...
```

In this context, the `return` keyword is followed by an expression and a semicolon. In our

case, there's an implicit cast expression that automatically converts the variable into the required type (int).

It can be enlightening to examine the parser's operation through the LLDB debugger:

```
1  $ lldb <...>/llvm-project/build/bin/clang -- -cc1 max.cpp
2  ...
3  (lldb) b clang::Parser::ParseReturnStatement
4  (lldb) r
5  ...
6  (lldb) c
7  ...
8  * thread #1, name = 'clang', stop reason = breakpoint 1.1
9      frame #0: ... clang::Parser::ParseReturnStatement(...) at
   ↪  ParseStmt.cpp:2358:3
10    2355 ///         'co_return' expression[opt] ';'
11    2356 ///         'co_return' braced-init-list ';'
12    2357 StmtResult Parser::ParseReturnStatement() {
13 -> 2358   assert((Tok.is(tok::kw_return) || Tok.is(tok::kw_co_return)) &&
14    2359          "Not a return stmt!");
15    2360   bool IsCoreturn = Tok.is(tok::kw_co_return);
16    2361   SourceLocation ReturnLoc = ConsumeToken();  // eat the 'return'.
17 (lldb) bt
18   * frame #0: ... clang::Parser::ParseReturnStatement( ...
19     ...
20     frame #2: ... clang::Parser::ParseStatementOrDeclaration( ...
21     frame #3: ... clang::Parser::ParseCompoundStatementBody( ...
22     frame #4: ... clang::Parser::ParseFunctionStatementBody( ...
23     frame #5: ... clang::Parser::ParseFunctionDefinition( ...
24 ...
```

Figure 2.16: Second return statement parsing at max.cpp example program

As you can see in fig. 2.16, line 3, we've set a breakpoint for the parsing of return statements[4]. Our program has two return statements. We bypass the first call (line 6) and halt at the second method invocation (line 13). The backtrace (from the 'bt' command at line

[4]Specifically, at the `clang::Parser::ParseReturnStatement` method

17) displays the call stack for the parsing process. This stack mirrors the parsing blocks we described earlier, adhering to the C++ grammar detailed in [for Standardization, 2020, lex.separate].

The parsing results in the generation of AST. We can also inspect the process of AST creation using the debugger. To do this, we need to set a corresponding breakpoint at the `clang::ReturnStmt::Create` method:

```
$ lldb <...>/llvm-project/build/bin/clang -- -cc1 max.cpp
...
(lldb) b clang::ReturnStmt::Create
(lldb) r
...
(lldb) c
...
* thread #1, name = 'clang', stop reason = breakpoint 1.1
    frame #0: ... clang::ReturnStmt::Create(...) at Stmt.cpp:1204:8
   1201
   1202 ReturnStmt *ReturnStmt::Create( ... ) {
-> 1204   bool HasNRVOCandidate = NRVOCandidate != nullptr;
   1205   ...
   1206   ...
   1207   return new (Mem) ReturnStmt(RL, E, NRVOCandidate);
(lldb) bt
* thread #1, name = 'clang', stop reason = breakpoint 1.1
  * frame #0: ... clang::ReturnStmt::Create( ...
    frame #1: ... clang::Sema::BuildReturnStmt( ...
    frame #2: ... clang::Sema::ActOnReturnStmt( ...
    frame #3: ... clang::Parser::ParseReturnStatement( ...
    frame #4: ... clang::Parser::ParseStatementOrDeclarationAfterAttributes(
  ↪ ...
    ...
```

As can be seen, the AST node for the return statement is created by the Sema component.

The beginning of the return statement parser can be located in frame 4:

```
(lldb) f 4
```

```
frame #4: ... clang::Parser::ParseStatementOrDeclarationAfterAttributes( ...
   325       break;
   326     case tok::kw_return:              // C99 6.8.6.4: return-statement
-> 327       Res = ParseReturnStatement();
   328       SemiError = "return";
   329       break;
   330     case tok::kw_co_return:            // C++ Coroutines: ...
(lldb)
```

As we can observe, there is a reference to the C99 standard [International Organization for Standardization (ISO), 1999] for the corresponding statement. The standard [International Organization for Standardization (ISO), 1999] provides a detailed description of the statement and the process for handling it.

The code assumes that the current token is of type `tok::kw_return`, and in this case, the parser invokes the relevant `clang::Parser::ParseReturnStatement` method.

While the process of AST node creation can vary across different C++ constructs, it generally follows the pattern displayed in fig. 2.17.

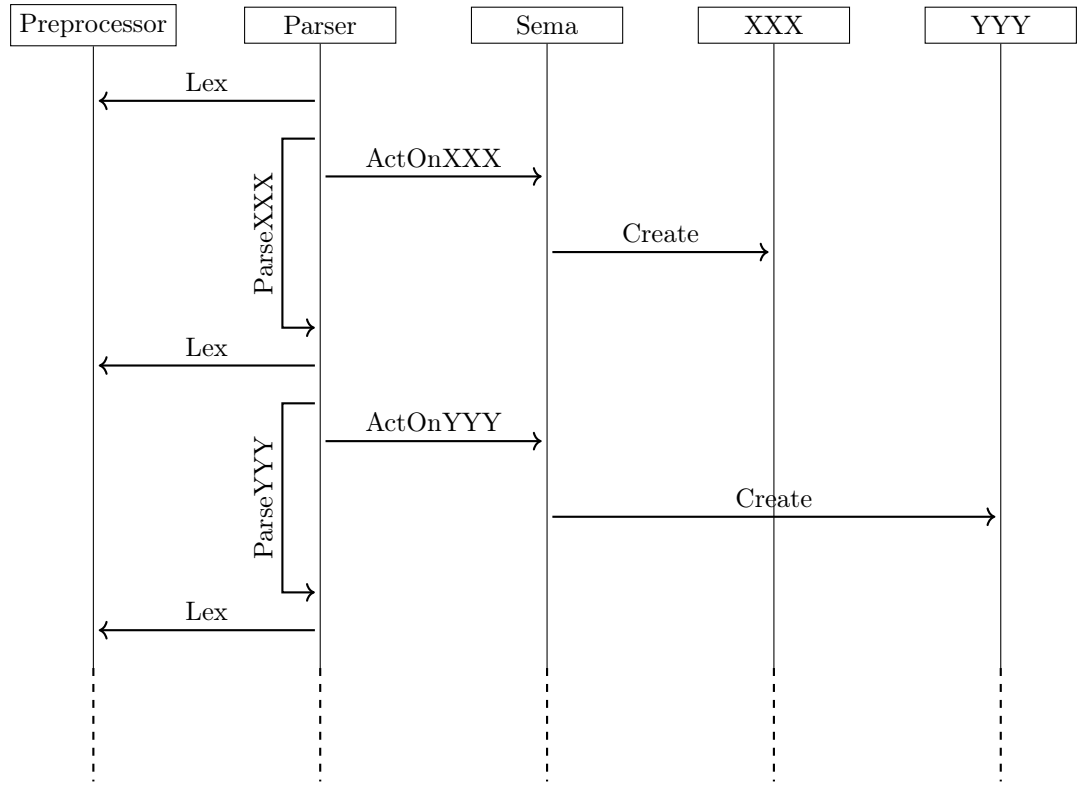

Figure 2.17: C++ parsing in Clang frontend

As can be seen, the `Parser` invokes the `Preprocessor::Lex` method to retrieve a token from the lexer. It then calls a method corresponding to the token, for example, `Parser:ParseXXX` for the token `XXX`. This method then calls `Sema::ActOnXXX`, which creates the corresponding object using `XXX::Create`. The process is then repeated with a new token.

With this, we have now fully explored how the typical compiler frontend flow is implemented in Clang. We can see how the lexer component (the preprocessor) works in tandem with the parser (which comprises the parser and sema components) to produce the primary data structure for future code generation: the Abstract Syntax Tree (AST). The AST is not only essential for code generation but also for code analysis and modification. Clang provides easy access to the AST, thereby enabling the development of a diverse range of compiler tools.

2.4 Summary

In this chapter, we have acquired a basic understanding of compiler architecture and delved into the various stages of the compilation process, with a focus on the Clang driver. We have explored the internals of the Clang frontend, studying the preprocessor that transforms a program into a set of tokens, and the parser, which interacts with a component called 'Sema'. Together, these elements perform syntax and semantic analysis.

The upcoming chapter will center on the Clang Abstract Syntax Tree (AST)—the primary data structure employed in various Clang tools. We will discuss its construction and the methods for traversing it.

2.5 Further reading

- Working Draft, Standard for Programming Language C++: https://eel.is/c++draft/
- "Clang" CFE Internals Manual: https://clang.llvm.org/docs/InternalsManual.html
- Keith Cooper and Linda Torczon: Engineering A Compiler, 2012 [Cooper and Torczon, 2012]