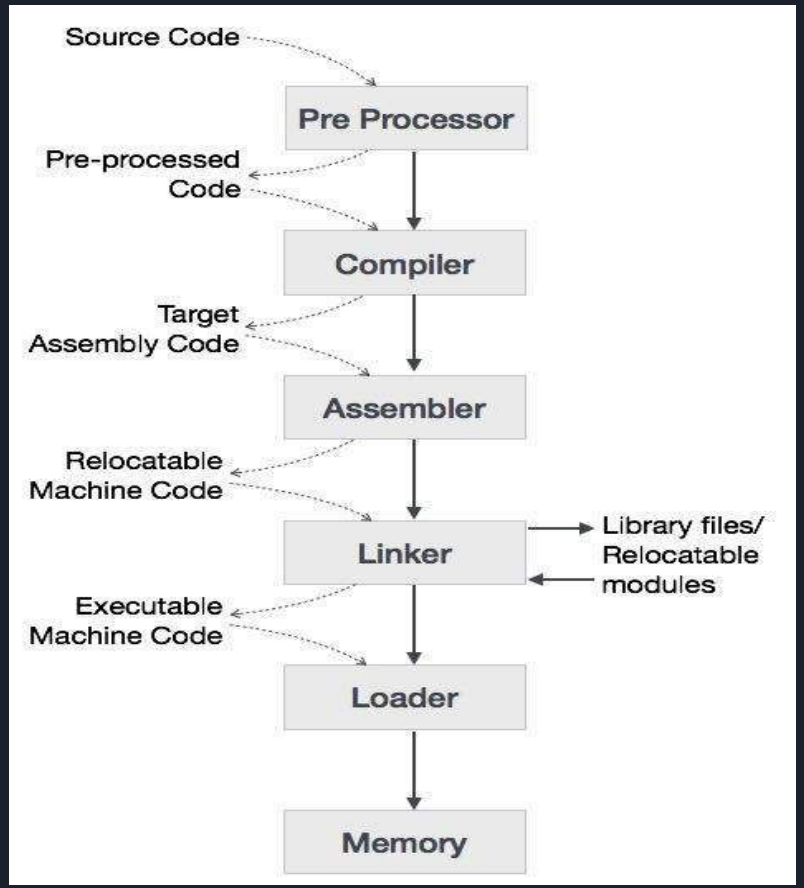
Under the hood of a compilation and

Intro to the LLVM Compiler Infrastructure which powers a lot of different programming languages Rust,C/C++,Swift,go,etc.

What is Compilation?

What is Compilation?

Compilation process in C is also known as **the process of converting Human Understandable Code (C Program) into a Machine Understandable Code (Binary Code))** Compilation process in C involves four steps: pre-processing, compiling, assembling, and linking.


Source Code
Pre Processor
Pre-processed Code
Compiler
Target Assembly Code
Assembler
Relocatable Machine Code
Linker
Library files/ Relocatable modules
Executable Machine Code
Loader
Memory

How it get converted to machine code? Magic?

How it get converted to machine code? Magic?

No, it’s compiler only. :)

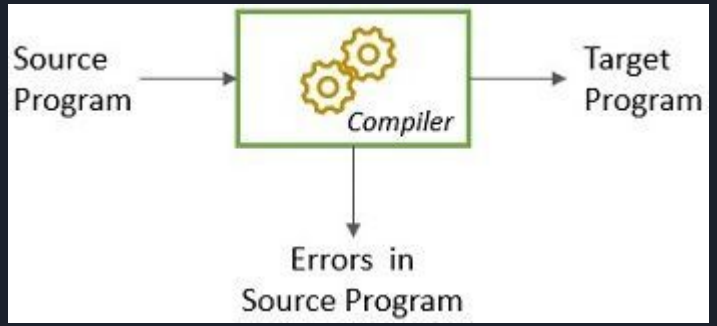
What are Compilers?

What are Compilers?

- Translating computer code written in one language (source) to another (target).

- The target language may be low-level such as assembly, object code etc.

- Compilers also provide optimizations (see: Don't Help the Compiler) and error handling.



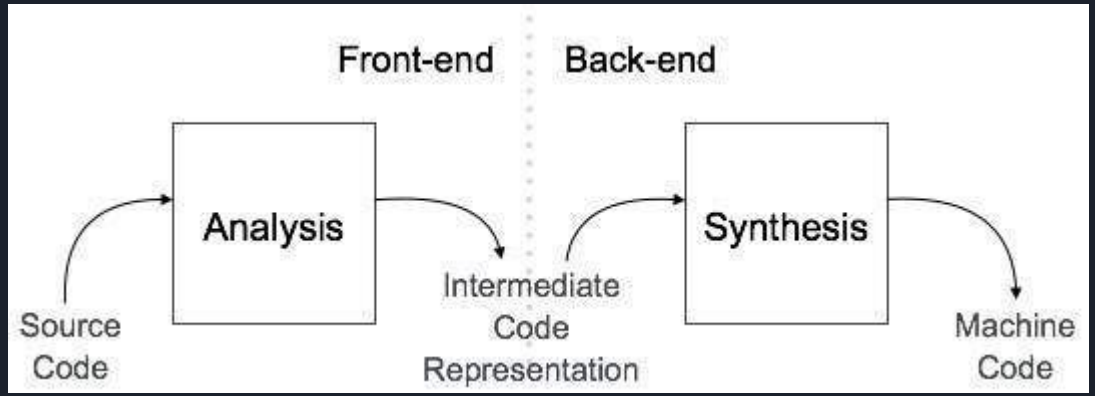
Why do we need Compilers?

- Could you write code in Binary?(0,1)
- Hardware/Processor only understands instructions in the form of electronic charge and which is counterpart of binary language in software programming.
- It would be a difficult and cumbersome task for computer programmers to write binary codes .
- which is why we have compilers to write such codes.

Compiler Architecture

A compiler can broadly be divided into two phases based on the way they compile.

Analysis or Frontend phase and Synthesis or Backend phase.



Frontend of Compiler

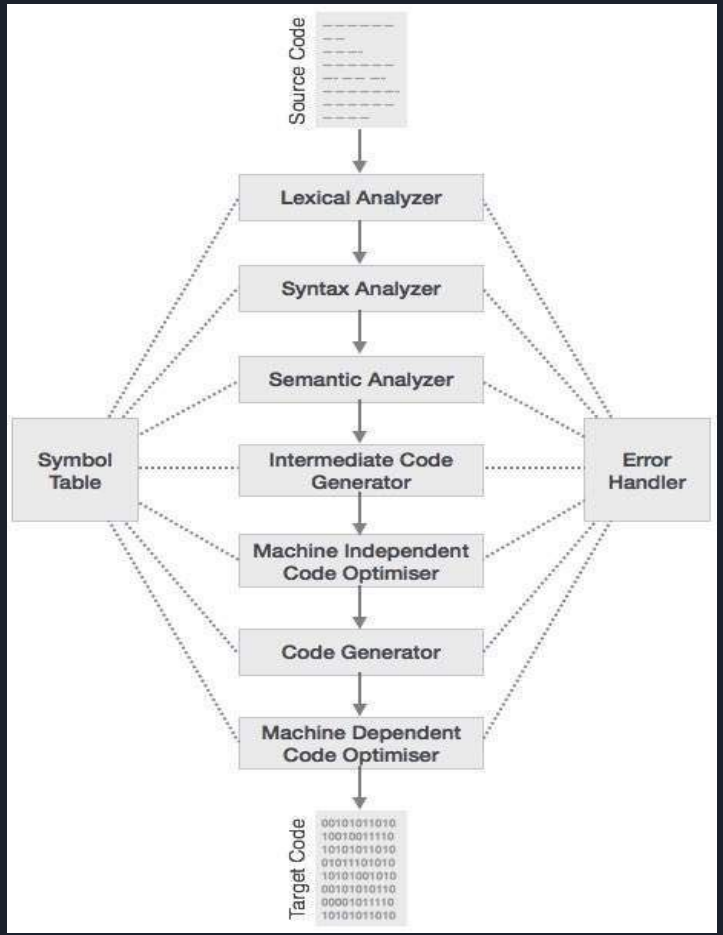
the **analysis** phase of the compiler reads the source program, divides it into core parts and then checks for lexical, grammar and syntax errors.The analysis phase generates an intermediate representation of the source program and symbol table, which should be fed to the Synthesis phase as input.

Backend of Compiler

the **synthesis** phase generates the target program with the help of intermediate source code representation and symbol table.

Phases of Compiler

The compilation process is a sequence of various phases. Each phase takes input from its previous stage, has its own representation of source program, and feeds its output to the next phase of the compiler. Let us understand the phases of a compiler.


Source Code
Lexical Analyzer
Syntax Analyzer
Semantic Analyzer
Symbol Table
Intermediate Code Generator
Error Handler
Machine Independent Code Optimiser
Code Generator
Machine Dependent Code Optimiser
Target Code

Dive into each phase

Lexical Analysis : The first phase of scanner works as a text scanner. This phase scans the source code as a stream of characters and converts it into meaningful lexemes. Lexical analyzer represents these lexemes in the form of tokens as:

<token-name, attribute-value>

Syntax Analysis: The next phase is called the syntax analysis or **parsing**. It takes the token produced by lexical analysis as input and generates a parse tree (or syntax tree). In this phase, token arrangements are checked against the source code grammar, i.e. the parser checks if the expression made by the tokens is syntactically correct.

Semantic Analysis: Semantic analysis checks whether the parse tree constructed follows the rules of language. For example, assignment of values is between compatible data types, and adding string to an integer. Also, the semantic analyzer keeps track of identifiers, their types and expressions; whether identifiers are declared before use or not etc. The semantic analyzer produces an annotated syntax tree as an output.

Intermediate Code Generation: After semantic analysis the compiler generates an intermediate code of the source code for the target machine. It represents a program for some abstract machine. It is in between the high-level language and the machine language. This intermediate code should be generated in such a way that it makes it easier to be translated into the target machine code.

Code Optimization: The next phase does code optimization of the intermediate code. Optimization can be assumed as something that removes unnecessary code lines, and arranges the sequence of statements in order to speed up the program execution without wasting resources (CPU, memory).

Code generation: In this phase, the code generator takes the optimized representation of the intermediate code and maps it to the target machine language. The code generator translates the intermediate code into a sequence of (generally) re-locatable machine code. Sequence of instructions of machine code performs the task as the intermediate code would do.

Symbol Table: It is a data-structure maintained throughout all the phases of a compiler. All the identifier's names along with their types are stored here. The symbol table makes it easier for the compiler to quickly search the identifier record and retrieve it. The symbol table is also used for scope management.

Take a look back in Past

- *1940s:* computers programmed in assembly
- *1951-2:* Grace Hopper developed A-0 for the UNIVAC
- *1957:* FORTRAN compiler developed by team led by John Backus
- 1960s: development of the first bootstrapping compiler for LISP

How the fortran compiler were coded?

It was created in assembly instructions , which can directly maps to machine opcodes, which are (multi-)byte values of machine code that can be directly interpreted by the processor.

So it is quite possible to write code in opcodes by looking the opcodes of the particular ISA, with the matching assembly instructions, and hand-determining memory addresses/offsets for things like jumps.

The first programs were done in exactly this fashion - hand-written opcodes.

However, most of the time it's simpler to use an assembler to "compile" assembly code, which automatically does these opcode lookups, as well as being helpful in computing addresses/offsets for named jump labels.

Usage and Scope

- Domain specific and device specific processing. Eg. Halide is a DSL for image processing and computational photography.

- Tools such as linters (clang-tidy), code formatters (ClangFormat), code completion (Intellisense), static analysis (Clang Static Analyzer), Debuggers etc.

- High Performance Computing (HPC) makes use of optimizing compilers for applications related to weather modelling, healthcare, physics simulations etc.

- and more

Compilers today

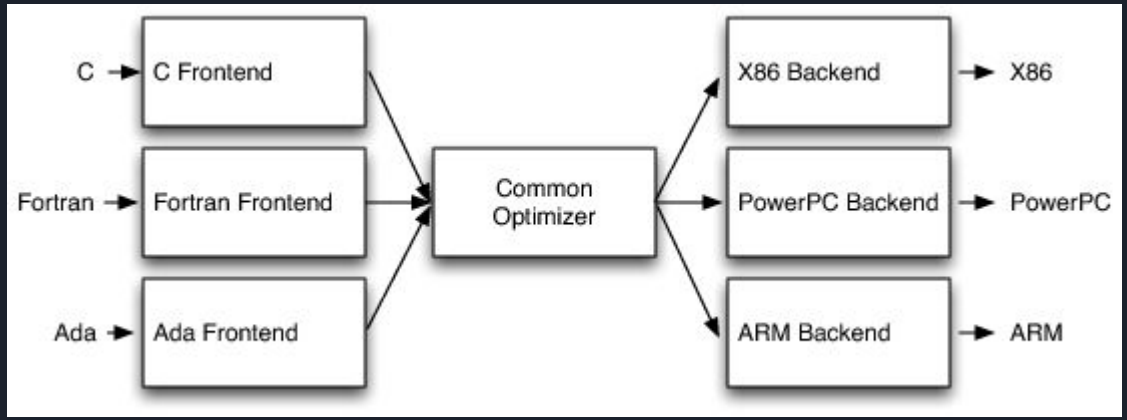
- Many compilers exist for C/C++/Fortran as these are the main HPC languages .
- Extensive use of open source compilers like GCC, Clang/LLVM .
- There are bunch of proprietary compilers exist like ICC, AOCC, PGI, ARM etc.
- Machine Learning techniques are exploring for more optimizations .
- There are different compilers project which are really active such as Rustc, Julia compiler etc. Community is pretty active.

Stakeholders In the Compiler Industry

What is LLVM and why is it so important?

- The LLVM project started in 2000 at the UIUC, created by Vikram Adve and Chris Lattner.
- It is a set of Compiler and toolchain technologies that can be used to create frontend for any programming language and a backend for any ISA.
- LLVM is designed around Language independent IR that serves as a portable, high level assembly like language that can be optimized with number of transformations.
- It currently has support for C, C++, Fortran, Rust, Objective C/C++, Julia, Haskell, etc.
- It has permissive open source license that enables collaboration and commercial use .

LLVM Architecture



The most important aspect of its design is the LLVM Intermediate Representation (IR), which is the form it uses to represent code in the compiler.

LLVM IR is designed to host mid-level analyses and transformations that you find in the optimizer section of a compiler. It was designed with many specific goals in mind, including supporting lightweight runtime optimizations, cross-function/interprocedural optimizations, whole program analysis, and aggressive restructuring transformations, etc.

LLVM IR example

Consider the addition of two numbers in C :

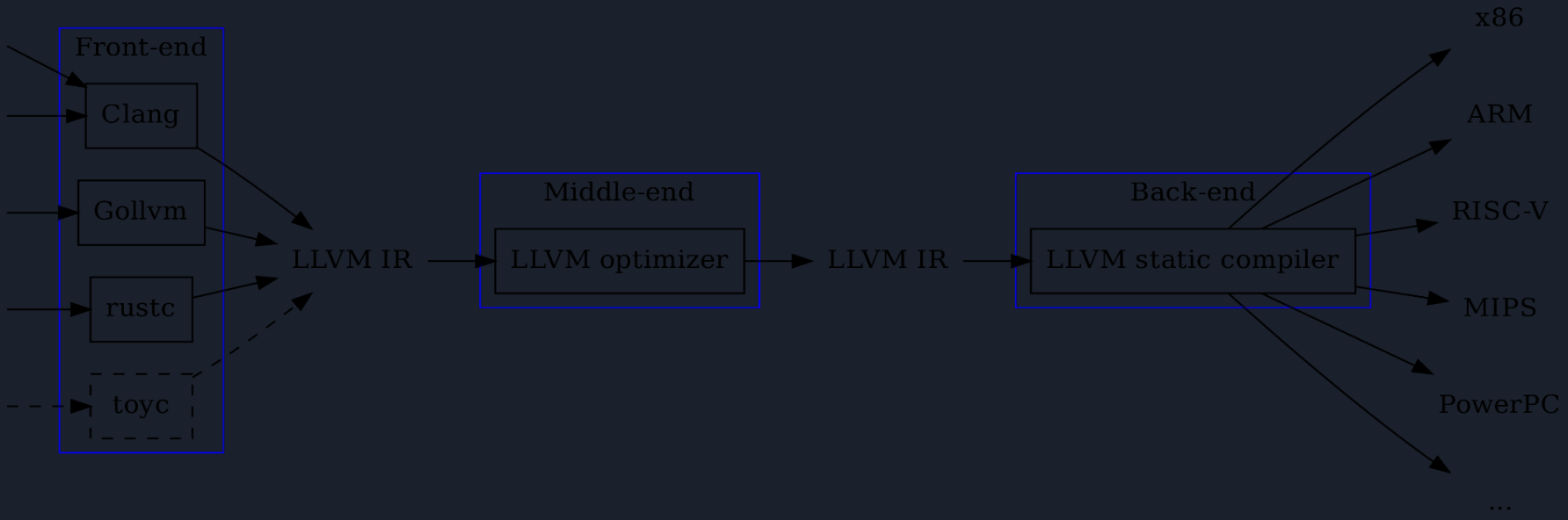
```
unsigned add1(unsigned a, unsigned b) {
 return a+b;
}
```

Equivalent IR :

```
define i32 @add1(i32 %a, i32 %b) {
entry:
 %tmp1 = add i32 %a, %b
 ret i32 %tmp1
}
```

As you can see from this example, LLVM IR is a low-level RISC-like virtual instruction set. Like a real RISC instruction set, it supports linear sequences of simple instructions like add, subtract, compare, and branch. These instructions are in three address form, which means that they take some number of inputs and produce a result in a different register.LLVM IR supports labels and generally looks like a weird form of assembly language.

Image credits
https://blog.gopheracademy.com/advent-2018/llvm-ir-and-go/



More on LLVM

- In an LLVM-based compiler, a front end is responsible for parsing, validating and diagnosing errors in the input code, then translating the parsed code into LLVM IR.
- This IR is optionally fed through a series of analysis and optimization passes which improve the code, then is sent into a code generator to produce native machine code.
- As we have codegen for different targets .

More on LLVM Architecture :

https://www.aosabook.org/en/llvm.html

https://llvm.org/

https://youtu.be/IR_L1xf4PrU

Want to dive into the LLVM Source Code ?

Subprojects

- >15 subprojects in the monorepo llvm/llvm-project
- 2 incubated projects

To get involved, two main questions -

- What to work on?
- How to work on it?

https://clang.llvm.org/cxx_status.html Implement missing language constructs.

https://github.com/llvm/llvm-project/issues Fix bugs in existing language constructs.

https://clang-analyzer.llvm.org/ Work on implementing new static analysis tool for CSA .

https://github.com/google/sanitizers Improve different sanitizers under LLVM like thread sanitizer, Address sanitizer, Memory sanitizer

Get Involve

Discussion forum - https://discourse.llvm.org/

Discord - https://discord.gg/xS7Z362

Thanks for bearing with me :) .

Contact me if you wants to ask something at <discord, phyBrackets#4476> .