1. what is llvm (describe easily): LLVM is a compiler infrastructure

designed as a set of reusable libraries with well-defined interfaces.

LLVM helps build new computer languages and improve existing languages.

It automates many of the difficult and unpleasant tasks involved in

language creation, such as porting the outputted code to multiple

platforms and architectures.

2. when it was invented

3.how does it work:on the font end, the llvm compiler infrastructure uses

different compiler such as Clang,Gollvm,rustc,toyc etc for different

language and turn source code into an interim format. Then the LLVM

code generator on the back end turns the interim format into final machine code.

The compiler has five basic phases:

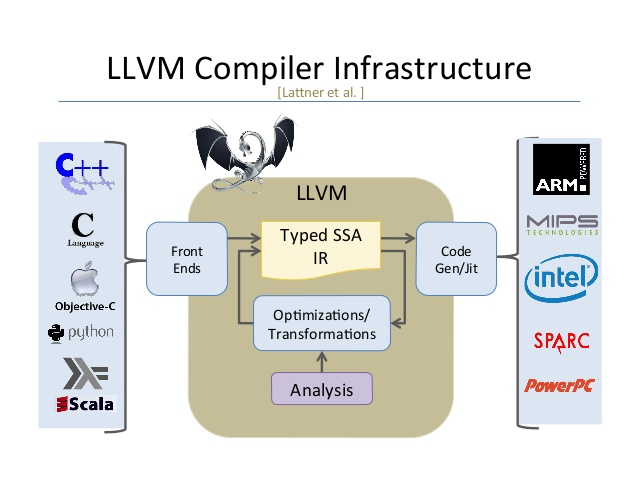
Lexical Analysis — Converts program text into words and tokens (everything apart from words, such as spaces and semicolons).

Parsing — Groups the words and tokens from the lexical analysis into a form that makes sense.

Semantic Analyser — Identifies the types and logics of the programs.

Optimization — Cleans the code for better run-time performance and addresses memory-related issues.

Code Generation — Turns code into a binary file that is executable.



4.how it is diff from other compilers

5.how programming languages use llvm

6.what llvm does not do

7.advantages of llvm

8.disadvantages of llvm

9.requirements

10.how llvm is designed:

11.LLvm IR:

LLVM IR is a low-level intermediate representation used by the LLVM compiler framework. You can think of LLVM IR as a platform-independent assembly language with an infinite number of function local registers.

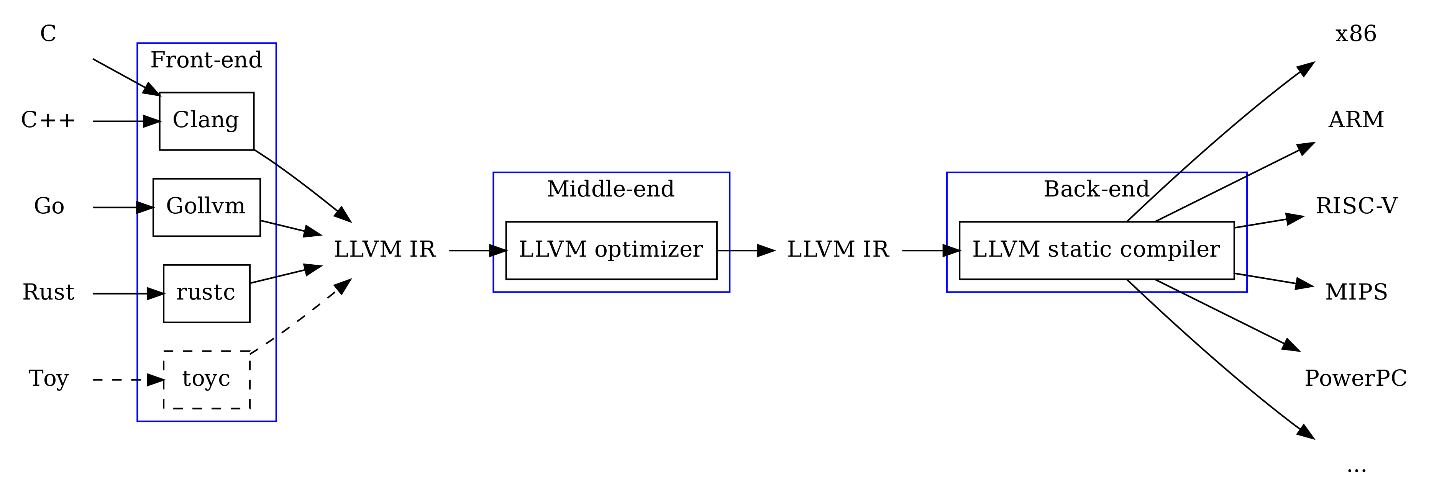
When developing compilers there are huge benefits with compiling your source language to an intermediate representation (IR)1 instead of compiling directly to a target architecture (e.g. x86). As many optimization techniques are general (e.g. dead code elimination, constant propagation), these optimization passes may be performed directly on the IR level and thus shared between all targets2.

Compilers are therefore often split into three components, the front-end, middle-end and back-end; each with a specific task that takes IR as input and/or produces IR as output.

Front-end: compiles source language to IR.

Middle-end: optimizes IR.

Back-end: compiles IR to machine code.



12.automatic code optimization with llvm:

LLVM doesn’t just compile the IR to native machine code. You can also programmatically direct it to optimize the code with a high degree of granularity, all the way through the linking process. The optimizations can be quite aggressive, including things like inlining functions, eliminating dead code (including unused type declarations and function arguments), and unrolling loops.

Again, the power is in not having to implement all this yourself. LLVM can handle them for you, or you can direct it to toggle them off as needed. For example, if you want smaller binaries at the cost of some performance, you could have your compiler front end tell LLVM to disable loop unrolling.