LLVM Tutorial

*The slides are based on material taken from Chris Lattner's and Lin Tan presentations

What is LLVM

- LLVM used to stand for “Low-Level Virtual Machine”
  - Now just a brand for the umbrella project
- Collection of low-level toolchain components (e.g., assemblers, compilers, debuggers, etc.)
  - http://llvm.org/ProjectsWithLLVM/
- Famous for its **Clang compiler**!
  - Widely used in academic research :)

Why not GCC?

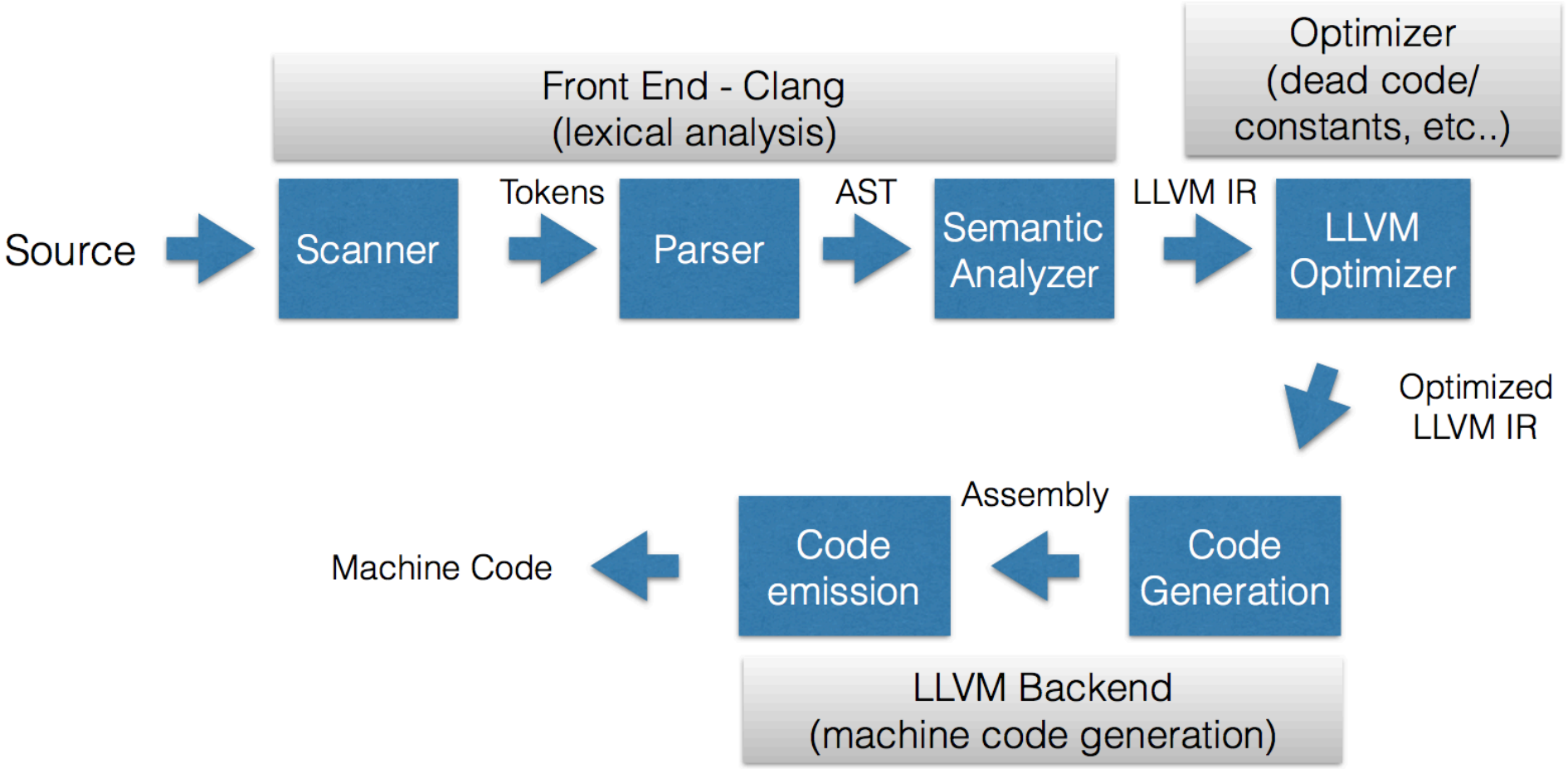
- Clang provides great support for source analysis tools (http://clang.llvm.org/docs/ClangTools.html), refactoring, and code generation.
- GCC is still a monolithic application, which makes it hard to extract pieces of GCC without pulling in most of the compiler.
- Error messages are easier to understand compared to GCC

```
$ gcc-4.2 -fsyntax-only t.c
t.c:7: error: invalid operands to binary + (have 'int' and 'struct A')
$ clang -fsyntax-only t.c
t.c:7:39: error: invalid operands to binary expression ('int' and 'struct A')
  return y + func(y ? ((SomeA.X + 40) + SomeA) / 42 + SomeA.X : SomeA.X);
                      ~~~~~~~~~~~~~~~ ^ ~~~~~~
```

Clang

- Clang is a compiler front end for C, C++ and Objective-C for the LLVM compiler (back end)
- Has a modularized design allowing it to be reused by source analysis tools.
- Clang has comparable speed compared to GCC (for the most part).
- In this project, we'll need to emit the **LLVM IR** of the source code!

Three-Phase Design of LLVM-based Compiler



Why is this useful?

LLVM IR

- Used to represent code in the compiler.
- Bitcode is similar to Java's ByteCode
- It can be executed with the help of LLVM's interpreter
- Serves as a "snapshot" of LLVM's AST

How to generate bitcode?

- The same as if you are generating an object:
  - clang/gcc -c foo.c -o foo.o
- Simply add the argument to emit the bitcode:
  - clang **-emit-llvm** -c foo.c -o foo.bc
- Executing the bit code file:
  - lli foo.bc

How to generate call graphs from the bitcode?

- Use the LLVM optimizer, ‘man opt’, from the man page
- Run “opt -print-callgraph foobar.bc”
  - By default, the call graph prints to **stderr** and the content of bitcode is printed to **stdout**
  - You can redirect the output of opt, or even discard information that you don’t need by redirecting garbage to /dev/null

opt [options] [filename] 2>&1 1>/dev/null

Project

- **Objective:**

You'll build a tool that detects bug based on likely-invariants (i.e., malloc-free bug) from call graphs.

Let's start off with an example!

Likely Invariant

- Remove duplicated functions
- Count pairs in all functions
- Take a pair, say B-D
- Called together 4 times
- B is called once w/o D
- B in scope3 is likely a bug
- Confidence(B→D) = 4/(4+1)=80%
- Support (B-D) = 4

```
void scope1(){
   A(); B(); C(); D();
}
void scope2(){
        A(); C(); D();
}
void scope3(){
       A(); B();
}
void scope4(){
      B(); D(); scope1();
}
void scope5(){
     B(); D(); A();
}
void scope6(){
     B(); D();
}
```

bug: B in scope3, pair: (A, B), support: 3, confidence: 75.00%

What to submit?

- Three items only (do not submit anything from the skeleton):
  - Makefile
  - Source code
  - pipair (if it is not an executable)

Test Suite

- There are a total of six tests, but only three are given to you. Your task is to make sure your program can pass all the tests.
- You write a program called “pipair”, and our script, “verify.sh” will compile your program and enumerate through the test cases:
  - Execute “make” in the test directory to generate the bitcode file
  - Run your program with the appropriate command line arguments
  - Wait for your program to finish execution and collect output.
  - Compare the output of your program against the “golden” files using “sort” and “diff”

Demo Code

- This will show you how “pipair” should interact with “verify.sh”
  - Create a binary called “pipair” and place it where “verify.sh” is located
  - Run “verify.sh” and observe each test case's status (pass/fail)
  - See Lab3_demo.c
  - “pipair” can be a bash wrapper script if you choose to implement your tool in Java, for example:

```
#!/bin/bash
java YourProgram $@
```

Tip: Use a hash function!

- We will evaluate the performance and scalability of your tool. Avoid string comparisons at all cost because it is a very $$$$ operation!
- Example: Given a string of names, [john, tom, sam, peter, john], print the frequency map of all the names. How can you program this efficiently without any string comparisons?

How to start?

- Read the PDF instructions and README file in the skeleton files carefully.
- Create LLVM Docker Container you can use the image from Lab2 or use another image that has LLVM installed.
- Mount one of the directories in your host machine to the Docker Image.
- Unzip the skeleton files on the mounted directory, and start with the demo code (Lab3_demo.c).
- Send me an email (bjarnil10@ru.is) and I will do our best to help you.