# LLVM

When source code is written by a programmer in a human readable form, it needs to be translated to machine language. There are three main concepts in achieving that goal:

- Don't compile. Instead, interpret code every time it's executed.
- Compile to byte code common for various devices and then translate to machine code on a target.
- Compile to machine code.

In our focus is the third approach.

The **LLVM Project** is a collection of modular and reusable compiler and toolchain technologies like clang, lldb, llc, lld, etc. In order to compile source code into binary form, compilers perform various analyzes:

- Lexical - Splits source code into tokens (parentheses, identifiers, constants, key words etc.)
- Syntax - Checks if braces are correctly paired, is there comma, semicolon or similar sign missing...
- Semantic - Code is syntactically valid, but doesn't make sense. For instance, there is no point in writing code in function after it returns.

Yet, many errors stay undetected by both compiler and a programmer. In order to reduce them, LLVM compiler uses certain optimizations called **passes**.

## Build Configuration

For detailed building of LLVM, one may consult official documentation. One of possibilities:

```
git clone 'https://github.com/llvm/llvm-project.git'
cd llvm-project
mkdir build && cd build
cmake -G Ninja -DLLVM_ENABLE_PROJECTS="clang;lld;lldb;llvm"
↪  -DCMAKE_INSTALL_PREFIX="$HOME/builds/llvm-release" -DLLVM_CCACHE_BUILD='ON'
↪  -DCMAKE_BUILD_TYPE="Release" -DENABLE_ASSERTIONS="ON" ../llvm
ninja
ninja install
```

## Implementing a Custom Pass

Let's suppose we have the following code saved in file *f-never-returns.c*. Note that function f never returns because the *for* loop never ends: unsigned variable will never be less than zero.

```
#include <iostream>

using namespace std;

unsigned f()
{
        unsigned cnt = 0, i = 1000;
```

```
        for(; i >= 0; i--)
            cnt++;

        return cnt;
}

int g()
{
        int a = 5;
        for(int j = 0; j < 10; j++)
            a *= -1;

        return a;
}

int main()
{
        int rv_g = g();
        cout << "Main: g returned " << rv_g << endl;
        unsigned rv_f = f();
        cout << "Main: f returned" << rv_f << endl;
        return 0;
}
```

It would be nice if we'd have compiler optimization that detects such errors.
The following steps implement solution. It is supposed that you have already cloned llvm-project repo and built llvm.

Add the following content to the file *llvm-project/llvm/include/llvm/Transforms/Utils/FunCantReturn.h*:

```
#ifndef LLVM_TRANSFORMS_FUNCANTRETURN_H
#define LLVM_TRANSFORMS_FUNCANTRETURN_H

#include "llvm/IR/PassManager.h"

namespace llvm {

class FunCantReturnPass : public PassInfoMixin<FunCantReturnPass> {
        public:
        PreservedAnalyses run(Function &F, FunctionAnalysisManager &AM);
        };
} // namespace llvm

#endif // LLVM_TRANSFORMS_FUNCANTRETURN_H
```

and the following to the *llvm-project/llvm/lib/Transforms/Utils/FunCantReturn.cpp*:

```
#include "llvm/Transforms/Utils/FunCantReturn.h"

using namespace llvm;

PreservedAnalyses FunCantReturnPass::run(Function &F,
FunctionAnalysisManager &AM) {
        if (F.doesNotReturn())
                errs() << "WARNING: " << F.getName() << " can't return." << "\n";
        return PreservedAnalyses::all();
}
```

Register your pass by adding the following to *llvm-project/llvm/lib/Passes/PassRegistry.def* in the *FUNCTION_PASS* section:

```
FUNCTION_PASS("fun-cant-return", FunCantReturnPass())
```

Add the proper include directive in llvm-project/llvm/lib/Passes/PassBuilder.cpp:

```
#include "llvm/Transforms/Utils/FunCantReturn.h"
```

and finally, add new pass to *llvm-project/llvm/lib/Transforms/Utils/CMakeLists.txt*:

```
FunCantReturn.cpp
```

Build LLVM optimizer called *opt* which performs passes on **Intermediate Representation** (IR) code:

```
cd llvm-project/build
ninja opt
ninja install # just if no errors reported
```

After installation is complete, we can see our pass is available in just built *opt* version:

```
build/opt -print-passes | grep fun-cant-return

# following line will be printed only if our pass has
# successufully added to opt. If not, return back and try to
# find error.
fun-cant-return
```

Let's test it on our code:

```
# get IR (.ll) from source
build/clang -emit-llvm -S f-cant-return.c -g -O2

# perform pass on IR code
build/opt f-cant-return.ll -passes=fun-cant-return

# opt's output. '_Z1fv' is name of function f in produced .ll code
# main never returns because it calls f which has an infinite for-loop
WARNING: _Z1fv can't return.
WARNING: main can't return.
```

One may write another pass in a similar way. The most important part is *run* method in *llvm-project/llvm/lib/Transforms/Utils/*.cpp* file because it defines what pass actually does. Registering, rebuilding opt and other steps remain the same.