HW1 Data Dependency

Homework 1 - Data Dependency

In this project, we will use the LLVM intermediate form to perform data dependence analysis on C programs with for loops. You need to report the dependence information within the for loop. The data dependence in this project only concerns the dependencies between array variables. You do not need to report the dependencies for scalar variables.

Case 1 - test1.c

This case considering the array index style i + c, as shown below, for one-dimensional arrays and a single-level loop. The format is A[f(i)], where f(i) is defined as i + c.

- test1.c

```
1  //test1.c
2  int main(){
3      int i;
4      int A[20], B[20], C[20];
5      for (i = 4; i < 20; i++) {
6          A[i] = C[i];
7          B[i] = A[i - 4];
8      }
9      return 0;
10 }
```

- Output

```
====Flow Dependency====
(i=4,i=8)
A:S1 -----> S2
(i=5,i=9)
A:S1 -----> S2
(i=6,i=10)
A:S1 -----> S2
(i=7,i=11)
A:S1 -----> S2
(i=8,i=12)
A:S1 -----> S2
(i=9,i=13)
A:S1 -----> S2
(i=10,i=14)
A:S1 -----> S2
(i=11,i=15)
A:S1 -----> S2
(i=12,i=16)
A:S1 -----> S2
(i=13,i=17)
A:S1 -----> S2
(i=14,i=18)
A:S1 -----> S2
(i=15,i=19)
A:S1 -----> S2
====Anti-Dependency====
====Output Dependency====
```

Case 2 - test2.c

In this case, we are also considering the array index style c i + d for one-dimensional arrays and a single-level loop. The format is A[f(i)], where f(i) is defined as c i + d.

- test2.c:

```
//test2.c
int main(){
    int i;
    int A[40], C[40], D[40];
    for (i = 2; i < 20; i++) {
        A[i] = C[i];
        D[i] = A[3 * i - 4];
        D[i - 1] = C[2 * i];
    }
    return 0;
}

```

- Output

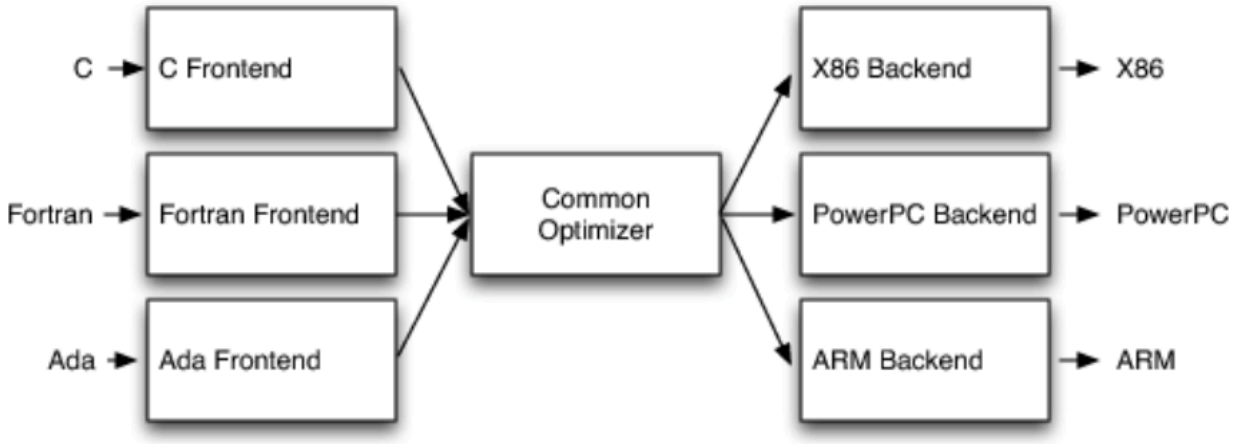
```
====Flow Dependency====
(i=2,i=2)
A:S1 -----> S2
====Anti-Dependency====
(i=3,i=5)
A:S2 --A--> S1
(i=4,i=8)
A:S2 --A--> S1
(i=5,i=11)
A:S2 --A--> S1
(i=6,i=14)
A:S2 --A--> S1
(i=7,i=17)
A:S2 --A--> S1
====Output Dependency====
(i=2,i=3)
D:S2 --O--> S3
(i=3,i=4)
D:S2 --O--> S3
(i=4,i=5)
D:S2 --O--> S3
(i=5,i=6)
D:S2 --O--> S3
(i=6,i=7)
D:S2 --O--> S3
(i=7,i=8)
D:S2 --O--> S3
(i=8,i=9)
D:S2 --O--> S3
(i=9,i=10)
D:S2 --O--> S3
(i=10,i=11)
D:S2 --O--> S3
(i=11,i=12)
D:S2 --O--> S3
(i=12,i=13)
D:S2 --O--> S3
(i=13,i=14)
D:S2 --O--> S3
(i=14,i=15)
D:S2 --O--> S3
(i=15,i=16)
D:S2 --O--> S3
(i=16,i=17)
D:S2 --O--> S3
(i=17,i=18)
D:S2 --O--> S3
(i=18,i=19)
D:S2 --O--> S3
```

Output format

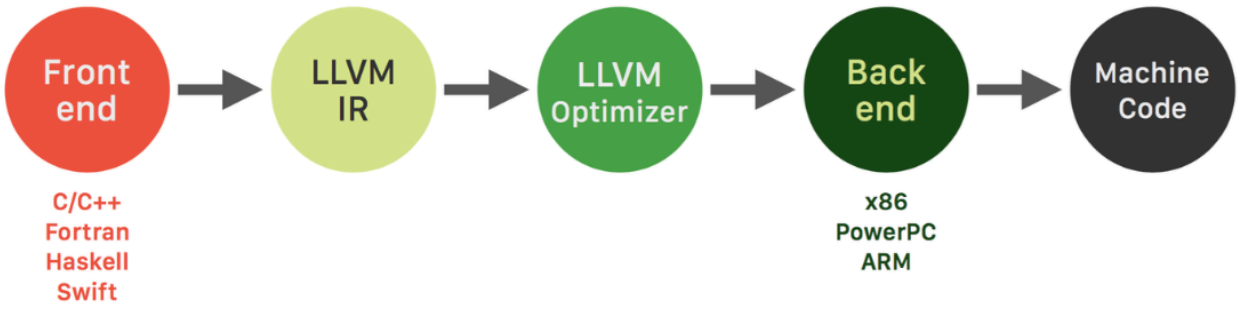
- general format:
  (i=[i of Sa], i=[i of Sb])
  [Array variable name]: Sa [dependency] Sb
- dependency representation:
  - flow: ——>
  - anti: –A–>
  - output: –O–>
- output order of types of dependency should follows:
  1. flow
  2. anti
  3. output
- output order within the same type is irrelevant
- output to either standard output or standard error is acceptable

Brief introduction to LLVM

- The LLVM Compiler Infrastructure (https://llvm.org/)



(http://www.aosabook.org/en/llvm.html (http://www.aosabook.org/en/llvm.html))



LLVM IR Overview

- Low level assembly like language
- Register machine, infinite number of registers
- Each instruction defines a new register (SSA form)
- Load/Store Architecture
- LLVM Language Reference Manual (http://llvm.org/docs/LangRef.html)

LLVM Program Structure

C program

```c
// foo.c
int foo(int a, int b) {
    int sum = a + b;
    if (sum < 0) sum = 0;
    return sum;
}
```

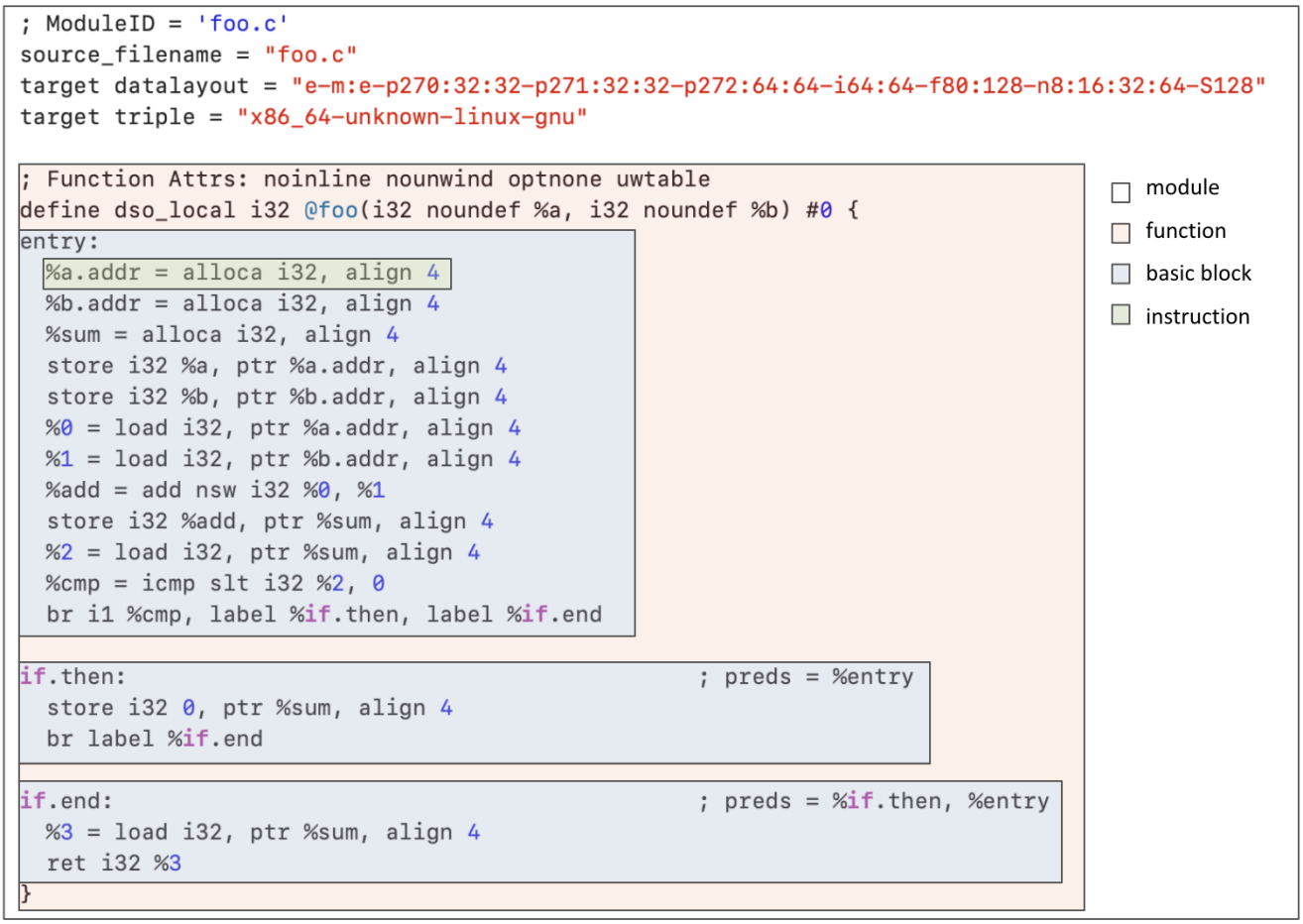
LLVM IR

```
; ModuleID = 'foo.c'
source_filename = "foo.c"
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @foo(i32 noundef %a, i32 noundef %b) #0 {
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  %sum = alloca i32, align 4
  store i32 %a, ptr %a.addr, align 4
  store i32 %b, ptr %b.addr, align 4
  %0 = load i32, ptr %a.addr, align 4
  %1 = load i32, ptr %b.addr, align 4
  %add = add nsw i32 %0, %1
  store i32 %add, ptr %sum, align 4
  %2 = load i32, ptr %sum, align 4
  %cmp = icmp slt i32 %2, 0
  br i1 %cmp, label %if.then, label %if.end

if.then:                                          ; preds = %entry
  store i32 0, ptr %sum, align 4
  br label %if.end

if.end:                                           ; preds = %if.then, %entry
  %3 = load i32, ptr %sum, align 4
  ret i32 %3
}
```



- LLVM Modules
  - An LLVM program consists of one or more modules, each representing a translation unit or a collection of source files. A module contains various entities, such as functions, global variables, and type definitions.
- Functions
  - Functions in LLVM IR are similar to functions in high-level languages. Each function comprises basic blocks that contain a sequence of instructions.
- Basic Blocks
  - Basic blocks are sequences of instructions with a single entry point and a single exit point. They are essential for control flow analysis and optimization.
- Instructions
  - Instructions are the fundamental building blocks of LLVM IR. Each instruction performs a specific operation, such as arithmetic, memory access, or control flow.

How to write an LLVM pass

Hello Pass

The following sample pass will invoke the run() method every time it encounters a function inside an LLVM module, printing out the name of the function.

hw1 pass sample (hw1.cpp)

```
#include "llvm/IR/PassManager.h"
#include "llvm/Passes/PassBuilder.h"
#include "llvm/Passes/PassPlugin.h"

using namespace llvm;

namespace {

struct HW1Pass : public PassInfoMixin<HW1Pass> {
  PreservedAnalyses run(Function &F, FunctionAnalysisManager &FAM);
};

PreservedAnalyses HW1Pass::run(Function &F, FunctionAnalysisManager &FAM) {
  errs() << "[HW1]: " << F.getName() << '\n';
  return PreservedAnalyses::all();
}

} // end anonymous namespace

extern "C" ::llvm::PassPluginLibraryInfo LLVM_ATTRIBUTE_WEAK
llvmGetPassPluginInfo() {
  return {LLVM_PLUGIN_API_VERSION, "HW1Pass", LLVM_VERSION_STRING,
          [](PassBuilder &PB) {
            PB.registerPipelineParsingCallback(
                [](StringRef Name, FunctionPassManager &FPM,
                   ArrayRef<PassBuilder::PipelineElement>) {
                  if (Name == "hw1") {
                    FPM.addPass(HW1Pass());
                    return true;
                  }
                  return false;
                });
          }};
}
```

Iterate over LLVM IR

```
for (BasicBlock &BB : F) {
    for (Instruction &I : BB) {
        processInst(I);
    }
}
```

Check pointer type

```
for (Instruction &I : BB) {
    if (auto *LI = dyn_cast<LoadInst>(&I)) {
        processLoadInst(LI);
    } else if (auto *SI = dyn_cast<StoreInst>(&I)) {
        processStoreInst(SI);
    }
}
```

Get Loop Information by LoopAnalysis Pass (Optional)

```
auto &LI = FAM.getResult<LoopAnalysis>(F);
for (const auto &L : LI) {
    processLoop(L);
}
```

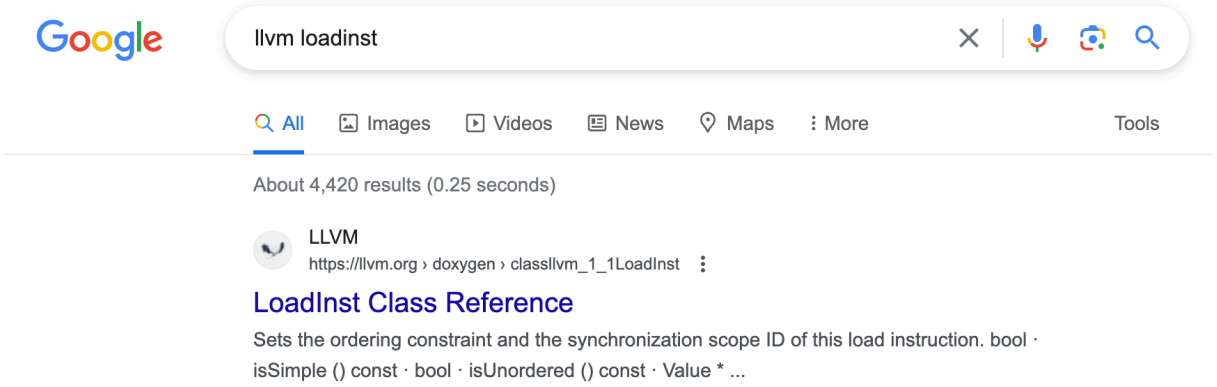
- llvm::Loop Class Reference (https://llvm.org/doxygen/classllvm_1_1Loop.html)
- Hint: getBounds()

Navigate through LLVM API Reference

- LLVM API Reference (https://llvm.org/doxygen/)

- For example, if you want to know how to find the pointer in a LoadInst:

    1. Search "LLVM LoadInst" on your search engine and go to the API documentation.



2. See whether its Public Member Functions list has the method you want to use.

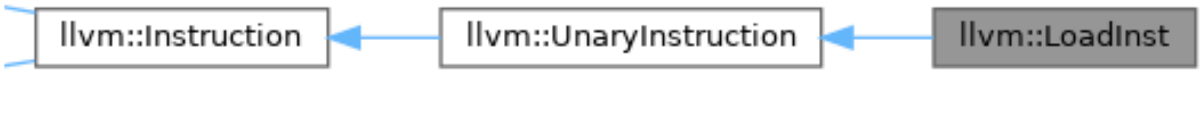
Public Member Functions

	LoadInst (**Type** *Ty, **Value** ***Ptr**, **const Twine** &NameStr, **Instruction** *InsertBefore)
	LoadInst (**Type** *Ty, **Value** ***Ptr**, **const Twine** &NameStr, **BasicBlock** *InsertAtEnd)
	LoadInst (**Type** *Ty, **Value** ***Ptr**, **const Twine** &NameStr, **bool isVolatile**, **Instruction** *InsertBefore)
	LoadInst (**Type** *Ty, **Value** ***Ptr**, **const Twine** &NameStr, **bool isVolatile**, **BasicBlock** *InsertAtEnd)
	LoadInst (**Type** *Ty, **Value** ***Ptr**, **const Twine** &NameStr, **bool isVolatile**, **Align Align**, **Instruction** *InsertBefore=nullptr)
	LoadInst (**Type** *Ty, **Value** ***Ptr**, **const Twine** &NameStr, **bool isVolatile**, **Align Align**, **BasicBlock** *InsertAtEnd)
	LoadInst (**Type** *Ty, **Value** ***Ptr**, **const Twine** &NameStr, **bool isVolatile**, **Align Align**, **AtomicOrdering** Order, **SyncScope::ID** SSID=**SyncScope::System**, **Instruction** *InsertBefore=nullptr)
	LoadInst (**Type** *Ty, **Value** ***Ptr**, **const Twine** &NameStr, **bool isVolatile**, **Align Align**, **AtomicOrdering** Order, **SyncScope::ID** SSID, **BasicBlock** *InsertAtEnd)
bool	**isVolatile** () **const** Return true if this is a load from a volatile memory location.

There is a getPointerOperand() function that seems to work!

bool	**isSimple** () **const**
bool	**isUnordered** () **const**
Value *	**getPointerOperand** ()
const Value *	**getPointerOperand** () **const**
Type *	**getPointerOperandType** () **const**
unsigned	**getPointerAddressSpace** () **const** Returns the address space of the pointer operand.

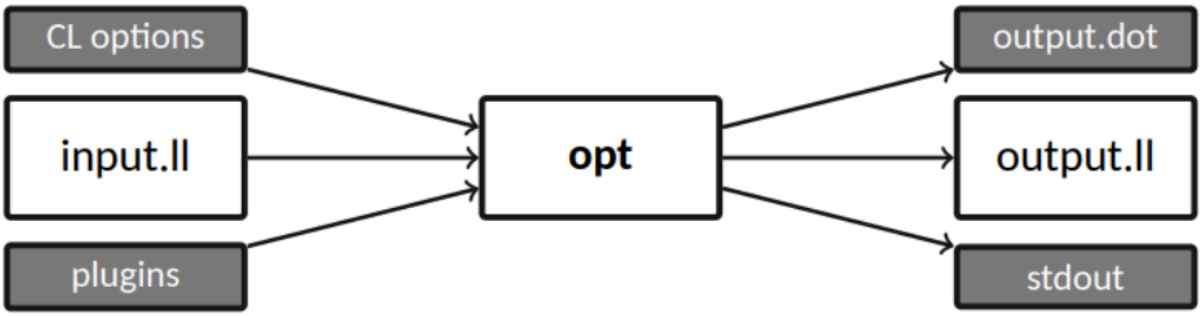
3. You can also try to search its base/derived class.



- Another tip for finding out how to use an API is to search the LLVM codebase and observe how others uesd it.

```
# In llvm-project-17.0.2.src/llvm
$ grep -r LoadInst
```

Run the pass on LLVM IR using opt



opt workflow

```
$ opt -S -load-pass-plugin=./mypass.so -passes=mypass-name input.ll -o output
```

Homework 1 - Development Environment

Download llvm-project source

- using curl

```
curl -L -O https://github.com/llvm/llvm-project/releases/download/llvmorg-17.(
```

- using wget

```
$ wget https://github.com/llvm/llvm-project/releases/download/llvmorg-17.0.2/
```

Extract files from the archive

```
$ tar -xf llvm-project-17.0.2.src.tar.xz
```

Build Clang and LLVM

- Create a build directory (select your host target for -DLLVM_TARGETS_TO_BUILD)

```
$ mkdir llvm_build && cd llvm_build
$ cmake ../llvm-project-17.0.2.src/llvm \
    -DLLVM_ENABLE_PROJECTS="clang" \
    -DLLVM_TARGETS_TO_BUILD="X86" \
    -DCMAKE_BUILD_TYPE=Release
```

- Optional: Build LLVM with Ninja (alternative for line 2)

```
$ cmake -G Ninja ../llvm-project-17.0.2.src/llvm \
    -DLLVM_ENABLE_PROJECTS="clang" \
    -DLLVM_TARGETS_TO_BUILD="X86" \
    -DCMAKE_BUILD_TYPE=Release
```

- Start the build in the build directory

```
$ cmake --build .
```

Create hw1 directory

- Your directory structure will look like:

```
advanced_compiler/
|-- hw1/
|   |-- hw1.cpp
|   |-- Makefile
|   |-- test1.c
|   |-- test2.c
|-- llvm-project-17.0.2.src/
|-- llvm_build/
```

Makefile sample

```
LLVM_CONFIG=/path/to/your/llvm_build/bin/llvm-config

CXX=`$(LLVM_CONFIG) --bindir`/clang
CXXFLAGS=`$(LLVM_CONFIG) --cppflags` -fPIC -fno-rtti
LDFLAGS=`$(LLVM_CONFIG) --ldflags`
IRFLAGS=-Xclang -disable-O0-optnone -fno-discard-value-names -S -emit-llvm
OPT=`$(LLVM_CONFIG) --bindir`/opt

.PHONY: all test run clean
all: hw1.so test

test: test1.ll

hw1.so: hw1.cpp
	$(CXX) -shared -o $@ $< $(CXXFLAGS) $(LDFLAGS)

test1.ll: test1.c
	$(CXX) $(IRFLAGS) -o $@ $<

run: test1.ll hw1.so
	$(OPT) -disable-output -load-pass-plugin=./hw1.so -passes=hw1 $<

clean:
	rm -f *.o *.ll *.so
```

If you encounter any problem, please contact TA(hmlai@pllab.cs.nthu.edu.tw (mailto:hmlai@pllab.cs.nthu.edu.tw)) or post your question on the eeclass discussion.

Homework 1 - Requirements

- LLVM version:
  - 17.0.2 (https://github.com/llvm/llvm-project/releases/tag/llvmorg-17.0.2)
  - You are required to implement the data dependency analysis algorithm on your own and should not use existing LLVM passes. (except LoopAnalysis)
- Deadline:
  - 2023/11/9 23:59
- Please upload the following files to NTHU eeclass:
  - Source Code
    - hw1.cpp (pass source code)
    - Makefile (TA will run "make hw1.so (http://hw1.so)" to build the .so)
  - Report

    - hw1_<student_id>_report.pdf
    - limited to 4 pages (additional 3 pages for Bonus part)
    - content:
        1. How to run your program (e.g. any paths that TA should modify in your Makefile)
        2. Describe the cases that you can handle
        3. Experiment report – how you implement the pass
        4. Bonus (see Homework 1 - Bonus)
- Late submission points deduction:
    - 10 points for every week late, up to a maximum of 50 points.

Homework 1 - Grading Policy

Rank	Request
D	Report only
C- ~ B-	Cannot pass any testcases
B	Pass test1.c or test2.c
B+	Pass test1.c and test2.c
A-	Pass test3.c (hidden)
A	Pass test4.c (hidden)
A+	Find depndency by solving diophantine equation - sample equation sovler (https://drive.google.com/file/d/1R9UMQL_K4Qj_YAlq1e9BtA4VxLtDUydr/view?usp=sharing) (should be mentioned in your experiment report)

Homework 1 - Bonus (10%)

Mixin pattern (5%)

We've seen in the sample pass that an LLVM pass class is derived from a template instantiation, using itself as a template argument. This is known as the Curiously Recurring Template Pattern (CRTP). Additionally, we can notice that the template name suggests this is a mixin pattern:

```
struct HW1Pass : public PassInfoMixin<HW1Pass>
```

- Please survey and explain why LLVM adopted this pattern for writing pass classes in your report.

Utilize ADT (5%)

LLVM provides a set of data structures/STL that are optimized for specific scenarios. These purpose-built data types are designed to enhance the efficiency and performance of LLVM-based applications.
For more information on ADT, please refer to Picking the Right Data Structure for a Task (https://llvm.org/docs/ProgrammersManual.html#picking-the-right-data-structure-for-a-task).

- Try to utilize ADT in your project and write down the experience in your report.

Reference

Padua, David A., and Michael J. Wolfe. “Advanced compiler optimizations for supercomputers.” Communications of the ACM 29.12 (1986): 1184-1201. (https://dl.acm.org/doi/abs/10.1145/7902.7904)