Objective
In this lab, you will implement a constraint-based analysis on simple C programs. You will discover relevant facts about LLVM IR Instructions, feed those facts into the Z3 constraint solver, then implement the rules of reaching definitions and liveness analysis in Z3's C++ API.

Resources
● Z3 tutorial and C++ API
- https://www.philipzucker.com/z3-rise4fun/
- https://z3prover.github.io/api/html/group__cppapi.html
- https://github.com/Z3Prover/z3/blob/master/examples/c%2B%2B/example.cpp
● Important classes
- https://z3prover.github.io/api/html/classz3_1_1fixedpoint.html
- https://z3prover.github.io/api/html/classz3_1_1expr.html

Setup
VM - sudo password is student.
https://drive.google.com/file/d/11NvIFIdFi3SGkhPbGPBGWFxEC2DVYUU-/view?usp=sharin
g

Download **datalog.zip** from Canvas and unzip in the home directory on the VM. This will create a datalog directory.
https://drive.google.com/file/d/1YhcSniKufeSJ3GNd9inBr63F5lzCQRay/view?usp=sharing

Build
Please refer to the **build.sh** located in the **datalog.zip** file

Instructions
In this lab, you will design a reaching definition analysis, then a live variables analysis, using Z3. The main tasks are to design the analysis in the form of Datalog rules through the Z3 C++ API, and implement a function that extracts logical constraints in the form of Datalog facts for each LLVM instruction.

We will then feed these constraints, along with your datalog rules, into the Z3 solver. The main function of src/Constraint.cpp ties this logic together, and provides comments to explain how the main components work together.

In short, the following tasks:
1. Write Datalog rules in the initialize function in Extractor.cpp to define the reaching definition analysis and live variable analysis.
2. Write the extractContraints function in Extractor.cpp that extracts Datalog facts from LLVM IR Instruction . You will likely need to extract different facts for reaching definitions analysis and liveness analysis.

Relations for Datalog Analysis. The skeleton code provides the definitions of necessary Datalog relations over LLVM IR in Extractor.h . In the following subsection, we will show how to represent a LLVM IR program using these relations.

The relations for the reaching definition analysis are as follows:

- Kill(X,Y) : Definition Y is killed by instruction X
- Gen(X,Y) : Definition Y is generated by instruction X
- Next(X,Y) : Instruction Y is an immediate successor of instruction X
- In(X,Y) : Definition Y may reach the program point immediately before instruction X
- Out(X,Y) : Definition Y may reach the program point immediately after instruction X

You will use these relations to build rules for both analyses in Extractor.cpp .

Defining Datalog Rules from C++ API. You will write your Datalog rules in the function initialize using the relations above. Consider an example Datalog rule:

A(X, Y) :- B(X, Z), C(Z, Y).

This rule corresponds to the following formula:

$\forall X, Y, Z.\ B(X, Z) \wedge C(Z, Y) \Rightarrow A(X, Y)$.

In Z3, you can specify the formula in the following sequence of APIs in *initialize* . Assume *ctx* and *Solver* are configured as shown in Extractor.cpp. They represent instances of a Z3Context and a fixed point constraint solver, respectively.


```
/* Declare functions */
z3::func_decl A = ctx.function("A", ctx.bv_sort(32));
z3::func_decl B = ctx.function("B", ctx.bv_sort(32));
z3::func_decl C = ctx.function("C", ctx.bv_sort(32));
/* Declare quantified variables */
z3::expr X = ctx.bv_const("X", 32);    // encode X as a 32-bit bitvector (bv)
z3::expr Y = ctx.bv_const("Y", 32);
z3::expr Z = ctx.bv_const("Z", 32);
/* Define and register rules */
z3::expr R0 = z3::forall(X, Y, Z, z3::implies(B(X,Z) && C(Z, Y), A(X,Y)));
Solver->add_rule(R0, ctx.str_symbol("R0"));
```

Study the above pattern with forall and implies closely, as you can adapt it to express all the Datalog relations required to complete this lab. (Note: the above code is not a fully functioning example and is more or less for conceptual demonstration purposes only.)

Extracting Datalog Facts. You will need to implement the function extractConstraints in Extractor.cpp to extract Datalog facts for each LLVM instruction. The skeleton code provides a couple of auxiliary functions in src/Extract.cpp and src/Utils.cpp help you with this task:

- void addX(const InstMapTy &InstMap, ...)
    - X denotes the name of a relation. These functions add a fact of X to the solver. It takes InstMap that encodes each LLVM instruction as an integer. This map is initialized in the main function

- vector<Instruction*> getPredecessors(Instruction *I)
    - Returns a set of predecessors of a given LLVM instruction I

● bool isDef(Instruction *I)
  ○ it returns true iff instruction I defines a variable

Miscellaneous.
● For convenience for easy debugging, you can use the toString(Value *) function in Utils.cpp

● If the --debug option is passed through the command line constraint -ReachDef --debug), it will print out several relations. You can extend the print function in Extractor.h for your local development purposes, but you cannot submit Extractor.h so use caution if you change it

In this Lab, you are passing bitcode for those programs into the constraint executable.

A Makefile is provided in the test directory to run both sample programs through Reaching Definitions Analysis and Liveness Analysis, redirecting output to files. You can use make to test everything, or you can invoke constraint individually like so:


```
$ cd ~/datalog/test
$ clang -emit-llvm -c -o Greatest.bc Greatest.c
$ clang -emit-llvm -c -o ArrayDemo.bc ArrayDemo.c
$ ../build/constraint -ReachDef Greatest.bc
$ ../build/constraint -ReachDef ArrayDemo.bc
$ ../build/constraint -Liveness Greatest.bc
$ ../build/constraint -Liveness ArrayDemo.bc
```


constraint will produce output formatted just like the cpt pass you built in Lab 2. If you build and run the unmodified skeleton, you'll see lots of empty IN and OUT sets per instruction:

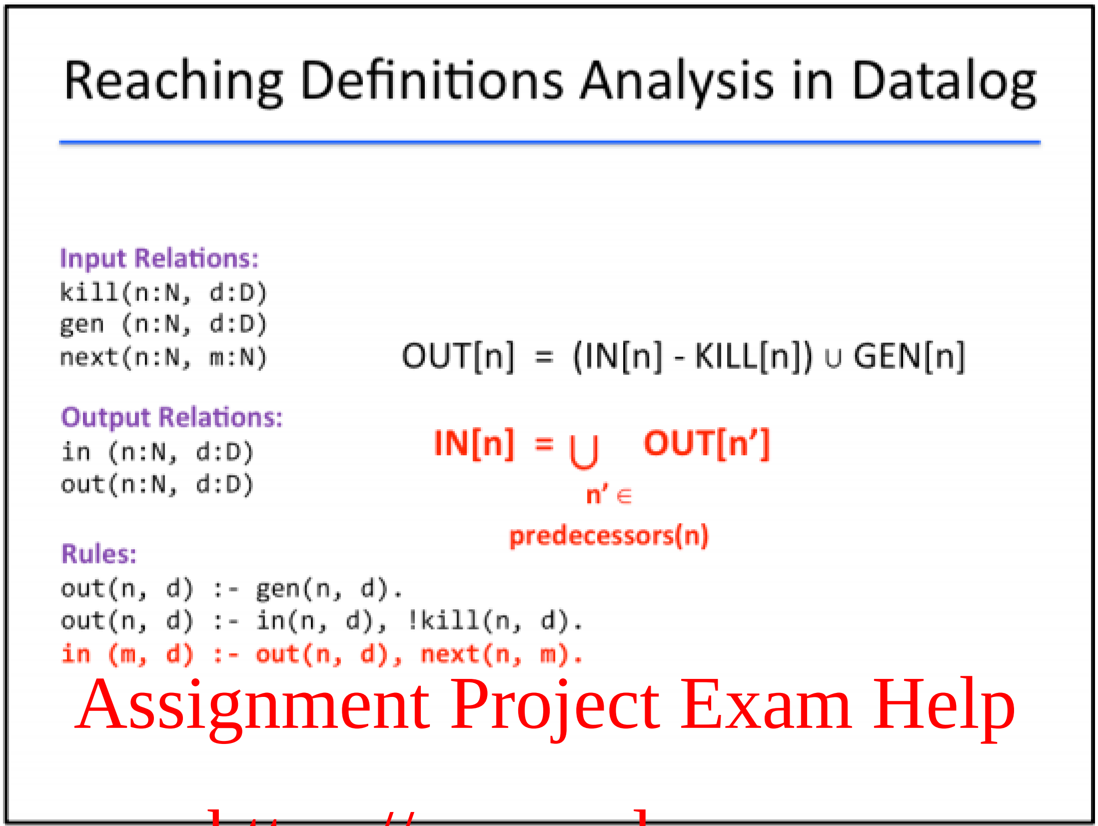
```
Instruction:   %1 = alloca i32, align 4
In set:
[]
Out set:
[]
```

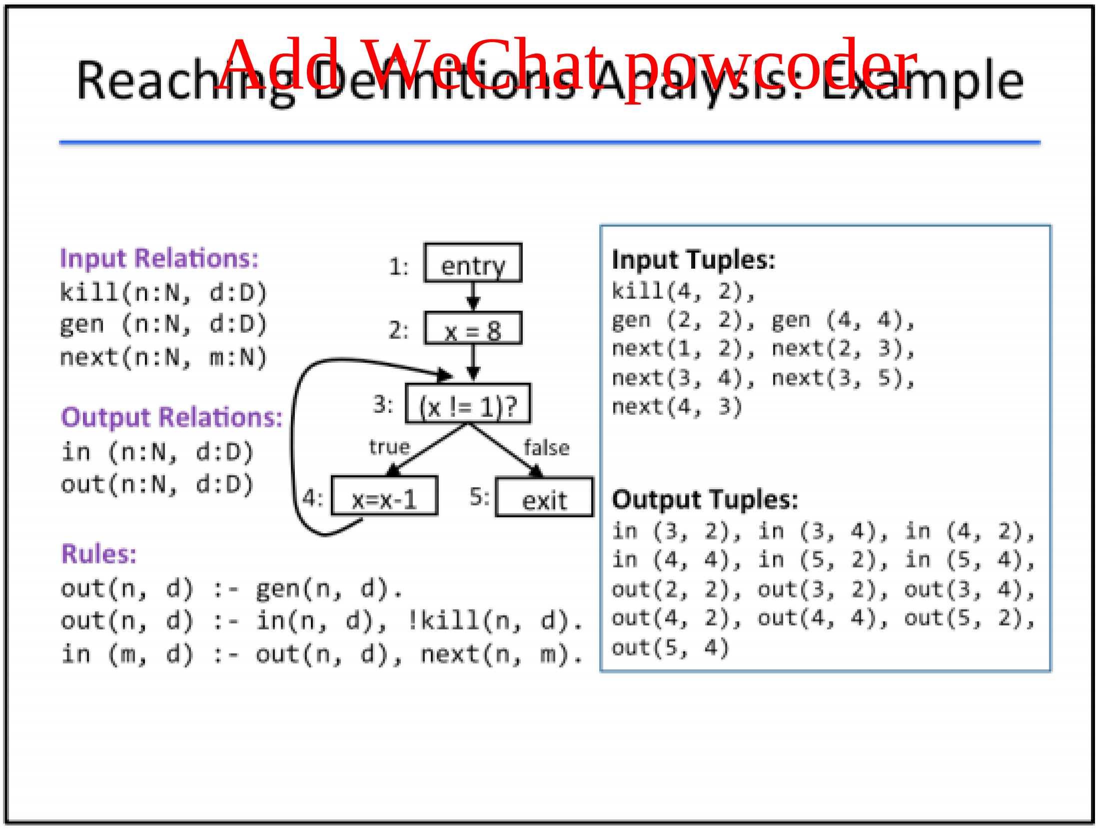
The contents of your IN and OUT sets matter, the order does not. Do not worry if the elements of your IN and OUT sets appear in a different order than the provided reference output.

If the Extractor.cpp code implementation is correct the output console should match the following files:

- ArrayDemo_Liveness
- ArrayDemo_ReachDef
- Greatest_Liveness
- Greatest_ReachDef

Tips


Reaching Definitions Analysis in Datalog
Input Relations:
kill(n:N, d:D)
gen (n:N, d:D)
next(n:N, m:N)
OUT[n] = (IN[n] - KILL[n]) ∪ GEN[n]
Output Relations:
in (n:N, d:D)
out(n:N, d:D)
IN[n] = ∪ OUT[n']
n' ∈
predecessors(n)
Rules:
out(n, d) :- gen(n, d).
out(n, d) :- in(n, d), !kill(n, d).
in (m, d) :- out(n, d), next(n, m).


Reaching Definitions Analysis: Example
Input Relations:
kill(n:N, d:D)
gen (n:N, d:D)
next(n:N, m:N)
Output Relations:
in (n:N, d:D)
out(n:N, d:D)
Rules:
out(n, d) :- gen(n, d).
out(n, d) :- in(n, d), !kill(n, d).
in (m, d) :- out(n, d), next(n, m).
1: entry
2: x = 8
3: (x != 1)?
true
false
4: x=x-1
5: exit
Input Tuples:
kill(4, 2),
gen (2, 2), gen (4, 4),
next(1, 2), next(2, 3),
next(3, 4), next(3, 5),
next(4, 3)
Output Tuples:
in (3, 2), in (3, 4), in (4, 2),
in (4, 4), in (5, 2), in (5, 4),
out(2, 2), out(3, 2), out(3, 4),
out(4, 2), out(4, 4), out(5, 2),
out(5, 4)

Writing Datalog rules with Z3++

The goal:

```
scc(n1, n2) :- path(n1, n2), path(n2, n1).
```

We give:

```
    "Solver" as a Z3 engine
    "SCC" and "Path" configured properly as Z3 relations
    "N1" and "N2" configured properly as Z3 variables
```

You provide:

```
z3::expr yourRule =
    z3::forall(N1, N2,
        z3::implies(Path(N1, N2) && Path(N2, N1), SCC(N1, N2)));

Solver->addRule(yourRule, "yourRule");
```

Deliverables

Extractor.cpp