EXERCISE #3

REVIEW: ABSTRACTING CODE

Write your name and answer the following on a piece of paper

- Draw the Control Flow Graph for the following code

```
void v(int a){
  if (a < 2){
    while (c < 3){
      c++;
    }
    if (b > 3){
      c = 12;
    }
  }
  return;
}
```

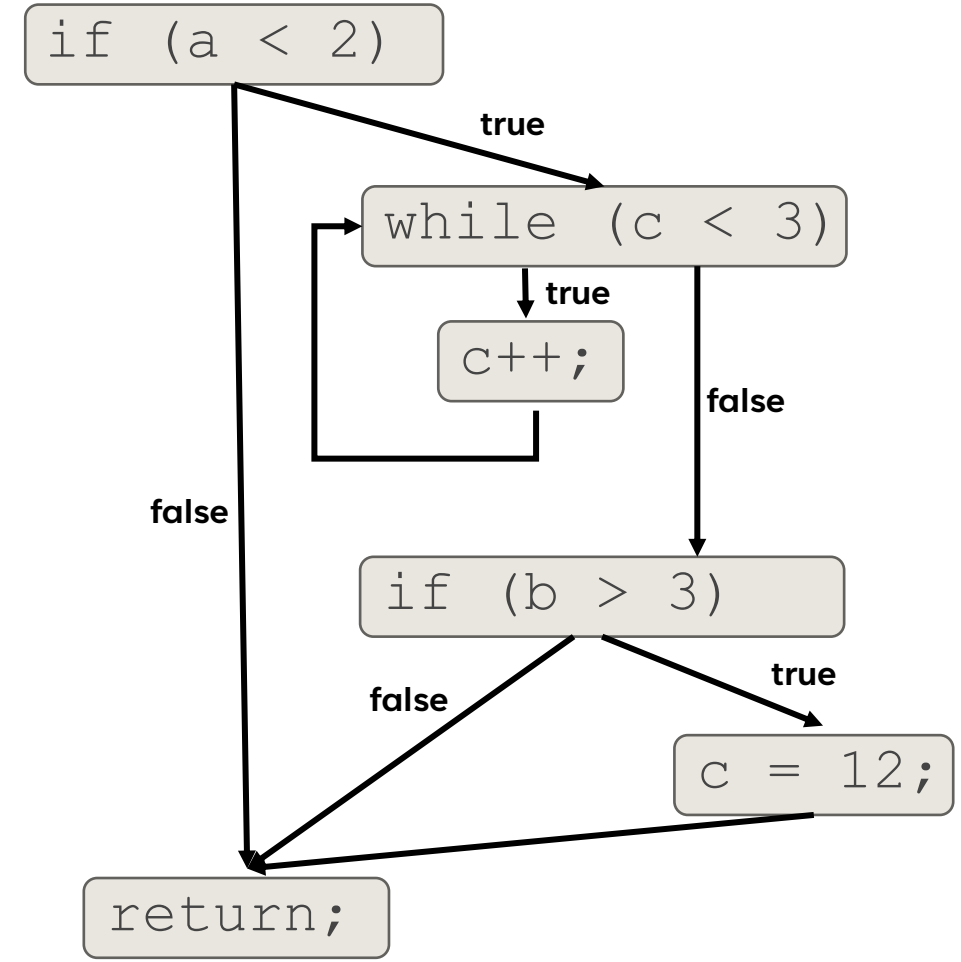
EXERCISE #3

REVIEW: ABSTRACTING CODE

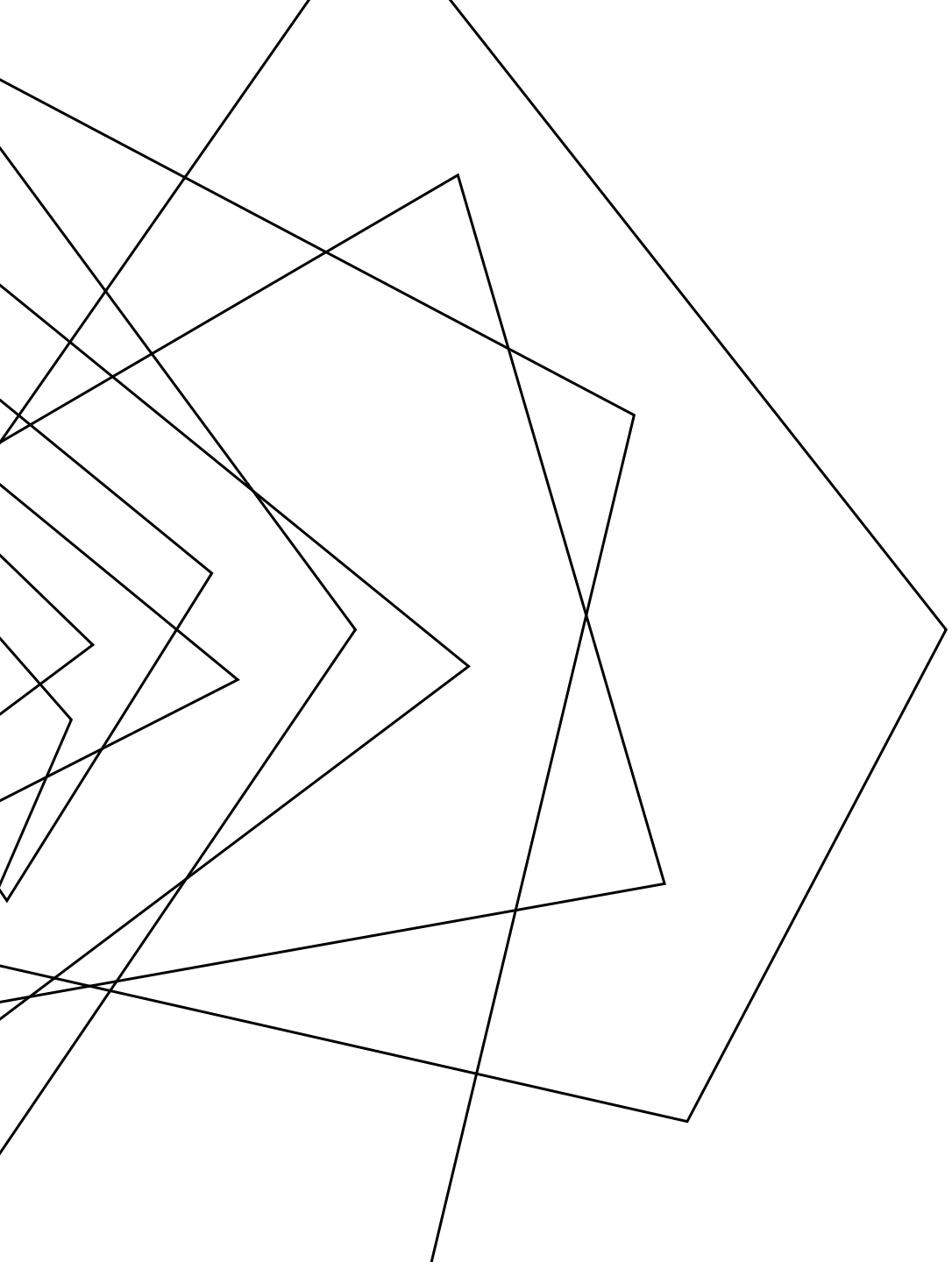
Write your name and answer the following on a piece of paper

- Draw the Control Flow Graph for the following code

```
void v(int a){
  if (a < 2){
    while (c < 3){
      c++;
    }
    if (b > 3){
      c = 12;
    }
  }
  return;
}
```



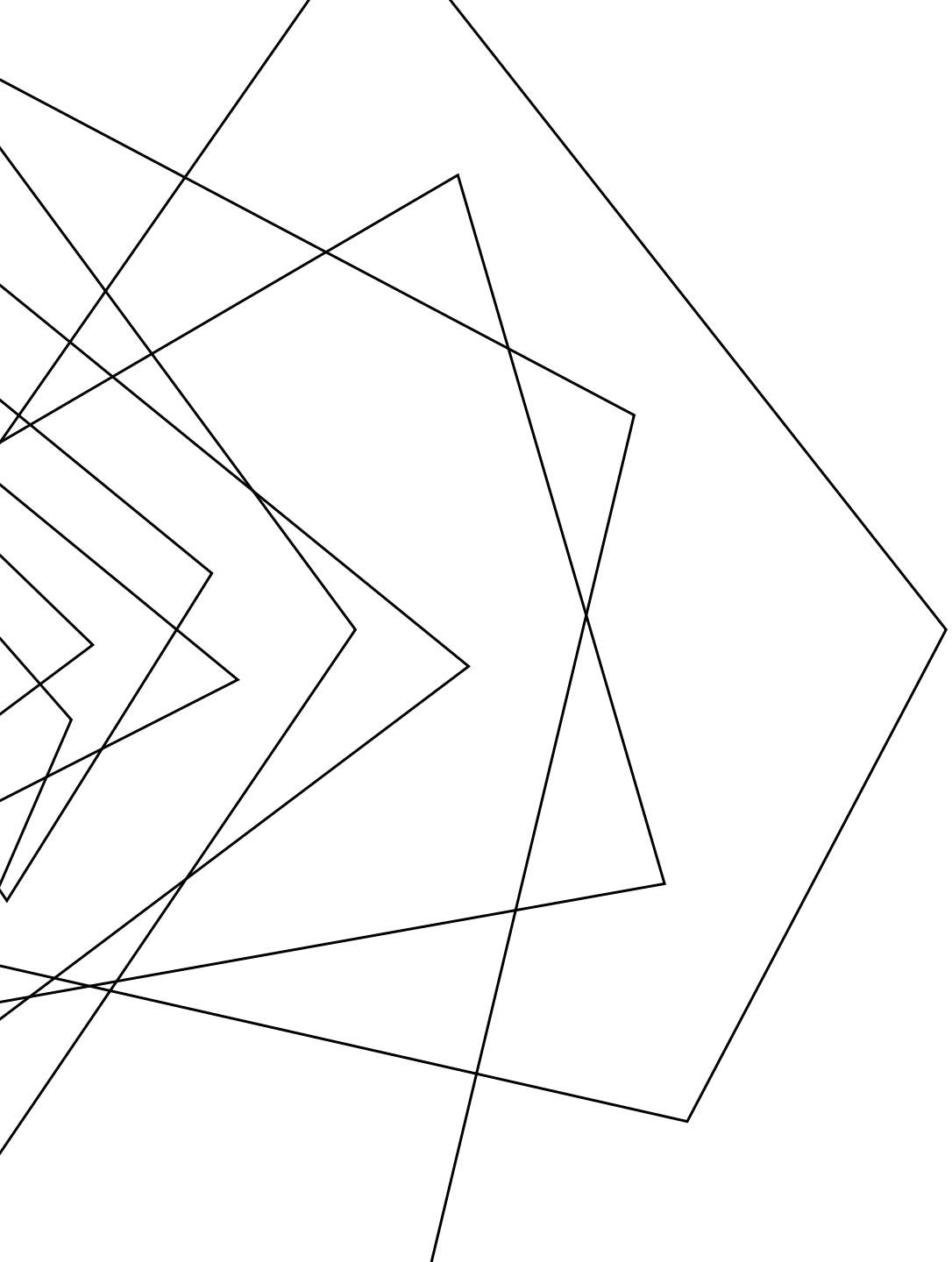
ADMINISTRIVIA AND ANNOUNCEMENTS



CLASS PROGRESS

EXPLORING PROGRAM ABSTRACTION

- SOME BASIC DATA STRUCTURES (CONTROL-FLOW GRAPHS)
- BUT WHAT DO THE DATA STRUCTURES HOLD?



LAST LECTURE: ABSTRACTING CODE

WHY WE MODEL/ABSTRACT PROGRAMS

ONE WAY TO ABSTRACT PROGRAMS

- THE CONTROL-FLOW GRAPH

REVIEW: BASIC BLOCKS

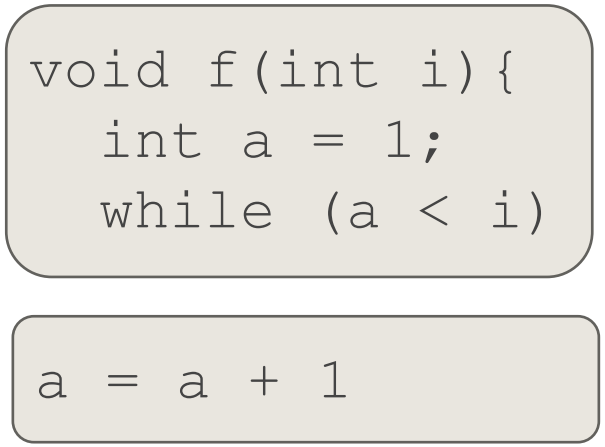
LAST LECTURE: ABSTRACTING CODE

VISUALIZE THE *FLOW OF CONTROL* IN THE PROGRAM

Use the Control-Flow Graph (CFG) to make flow explicit

Guaranteed sequential “fall-through” code is glommed into the same node (called a basic block)

```
void f(int i){
   int a = 1;
   while (a < i){
      a = a + 1;
   }
}
```



REVIEW: CONTROL-FLOW GRAPHS

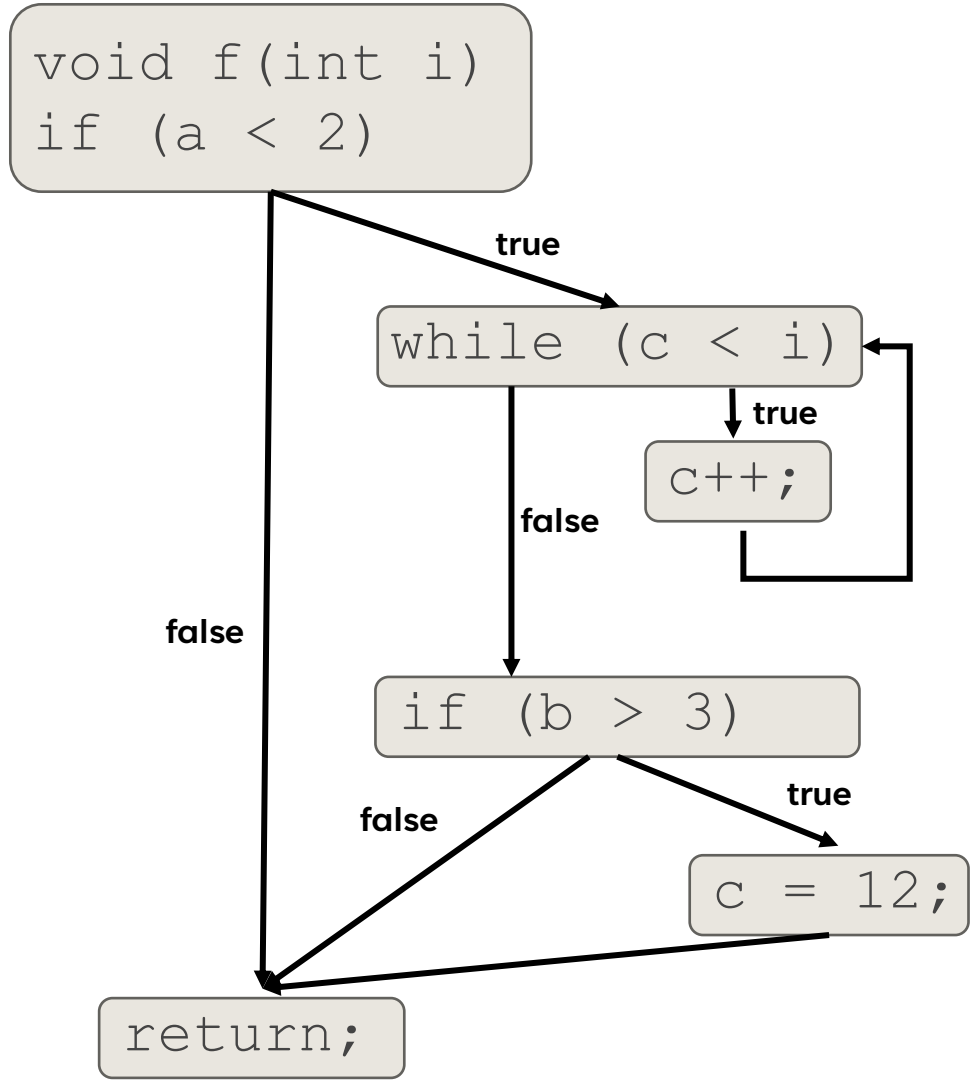
LAST LECTURE: ABSTRACTING CODE

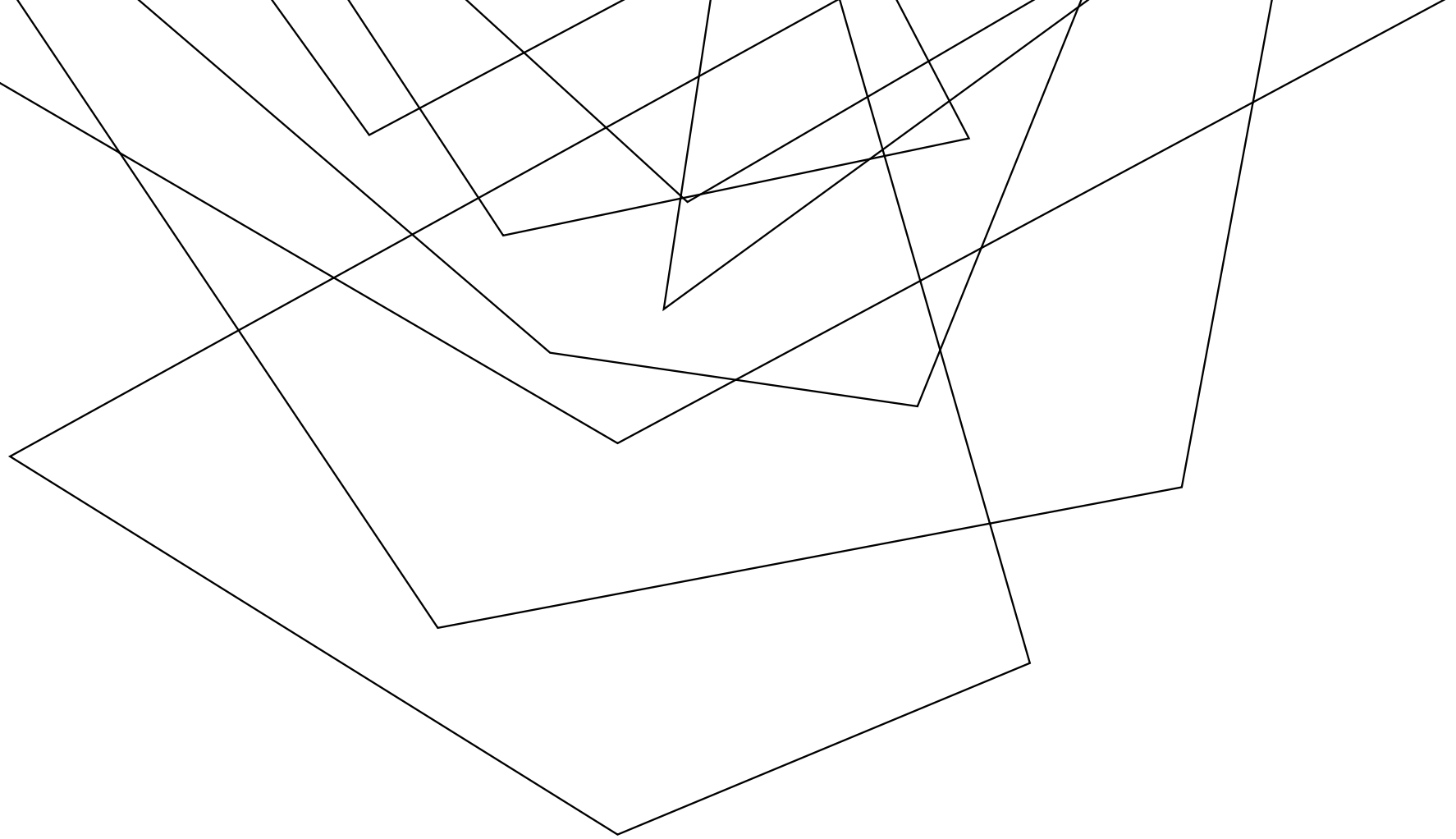
VISUALIZE *FLOW OF CONTROL* IN A PROCEDURE

Use the Control-Flow Graph (CFG) to make flow explicit

Connect basic blocks according to possible successors

```
void f(int i){
  if (a < 2){
    while (c < i){
      c++;
    }
  }
  if (b > 3){
  }
  return;
}
```





LLVM BITCODE

EECS 677: Software Security Evaluation

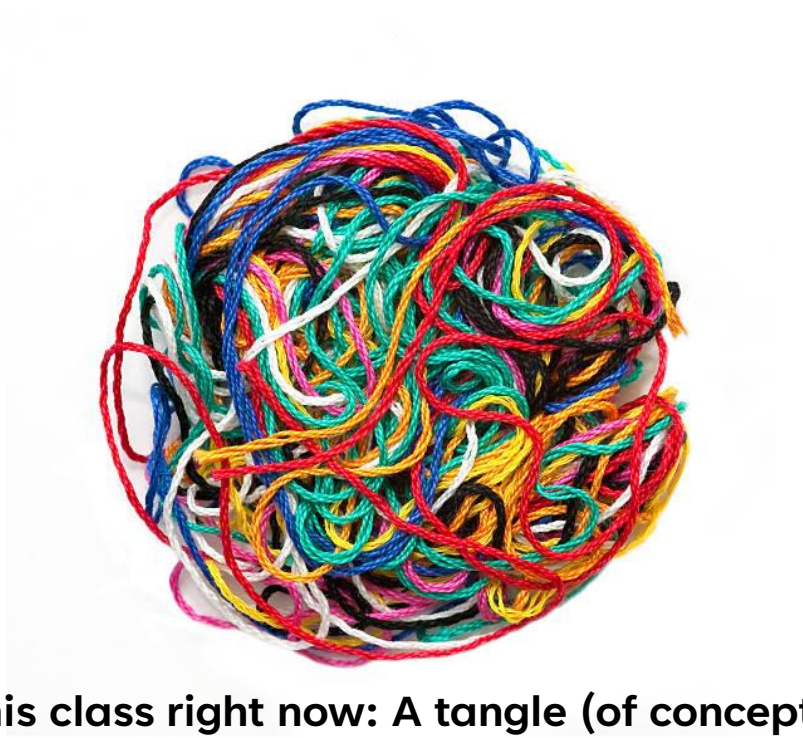
Drew Davidson

QUICK DISCLAIMER

VIBE CHECK

Entangled Concepts

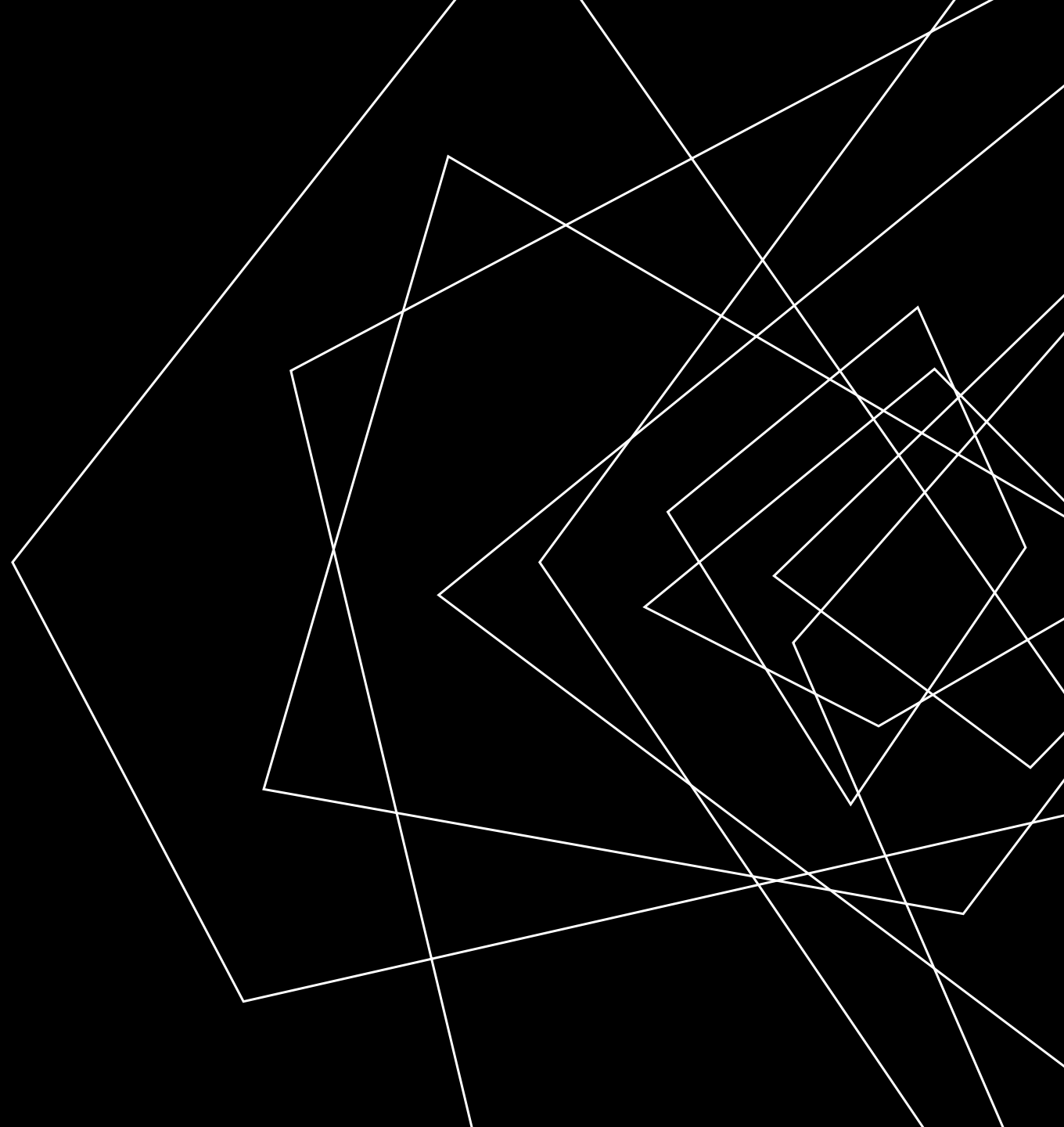
- There are a lot of deep concepts that are interdependent
- We’re skimming through a couple of ideas just enough to dig in
- We’ll come back to



This class right now: A tangle (of concepts)

LECTURE OUTLINE

- LLVM Bitcode Format
- Very simple examples
- SSA Format

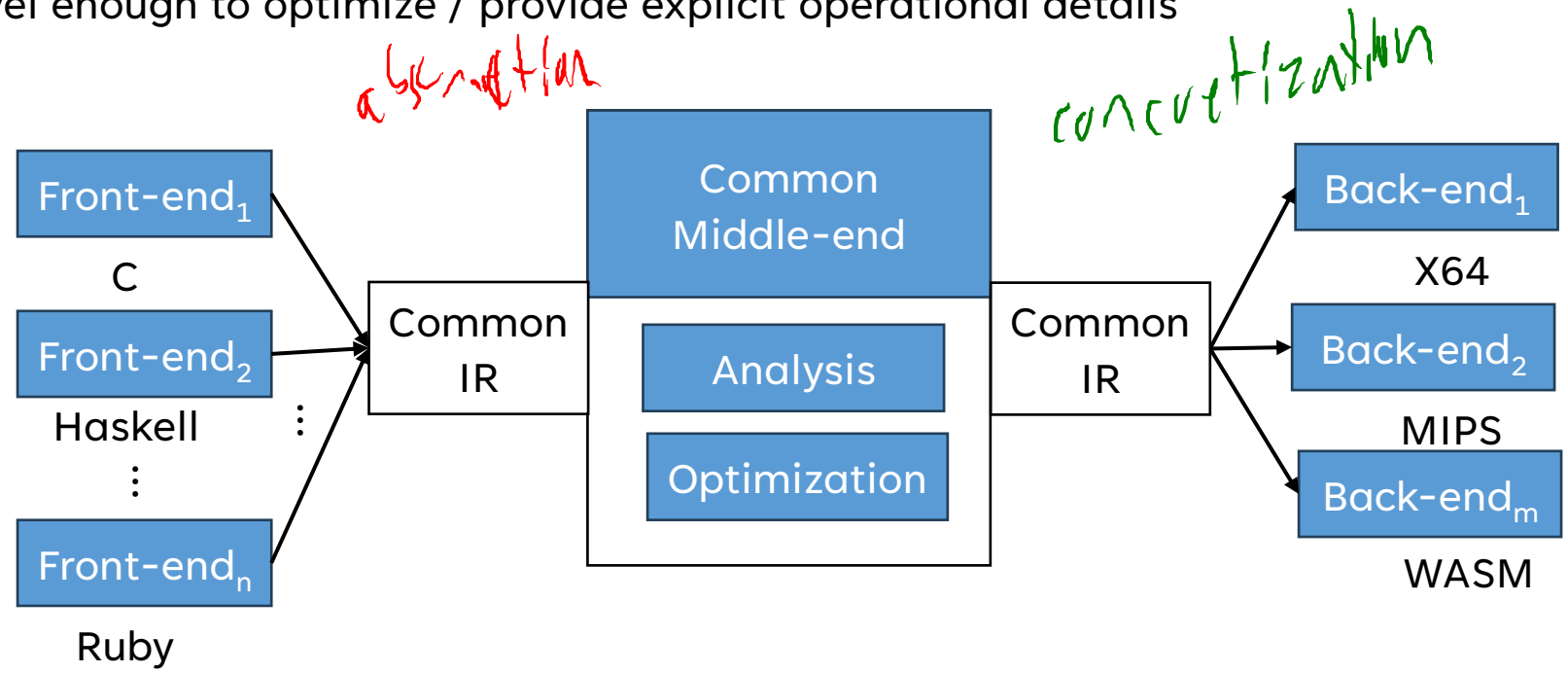


BEHOLD: LLVM

LLVM BITCODE FORMAT

A SET OF PROGRAM MANIPULATION TOOLS BUILT AROUND A "MID-LEVEL" ABSTRACT INSTRUCTION SET

- Called an intermediate representation (IR) because it sits between source code and executable
- High level enough to avoid architecture lock-in
- Low level enough to optimize / provide explicit operational details



LLVM'S "UNIVERSAL IR"

LLVM BITCODE

Bit-Code Language Design Goals

An in-memory compiler IR

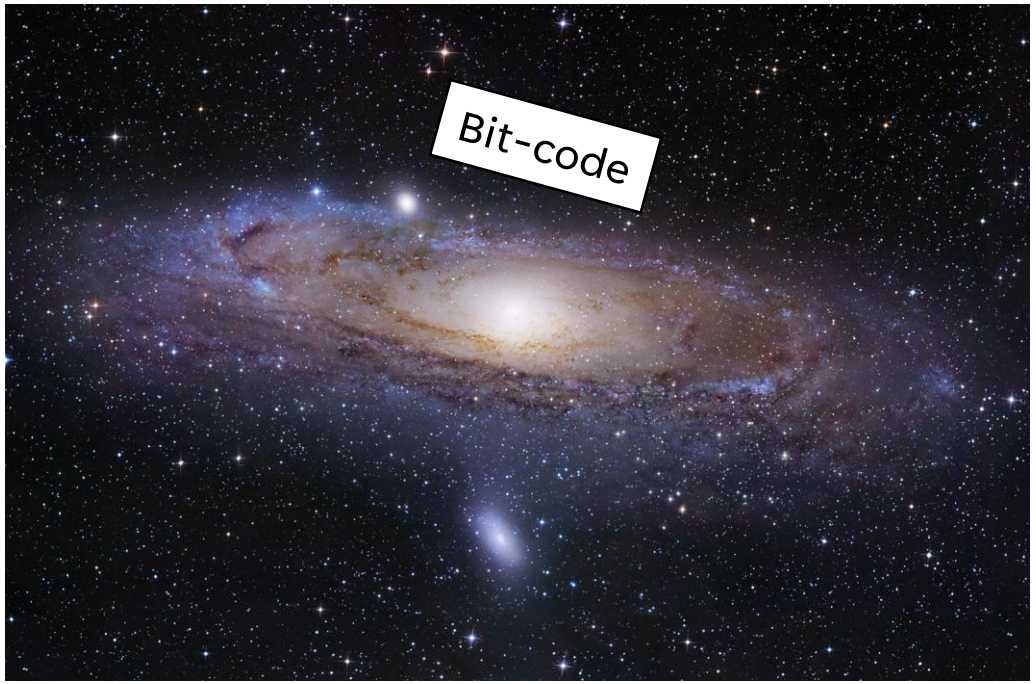
An on-disk program representation

A human readable assembly language

A Compiler's Representation

Relatively generic

Relatively easy to analyze



BITCODE STRUCTURE

LLVM BITCODE

Nested Structure

Modules

Individual translation unit (can be a whole program)

Functions

Invokable execution units

Global variables (globals)

Regions of statically-allocated memory

Local variables

Regions of dynamically-allocated memory

Instructions

Data transformers

Registers

Value holders

AN ABSTRACT COMPUTER

LLVM BITCODE

NO REAL COMPUTER RUNS BITCODE NATIVELY*

Abstract representation of memory

Highly-explicit instructions

*Without some additional translation software

LLVM'S ABSTRACT MEMORY

LLVM BITCODE

Named Memory Objects

No explicit layout between objects

Sized field within the object

Highly-explicit instructions

Abstract registers

Infinite number of registers

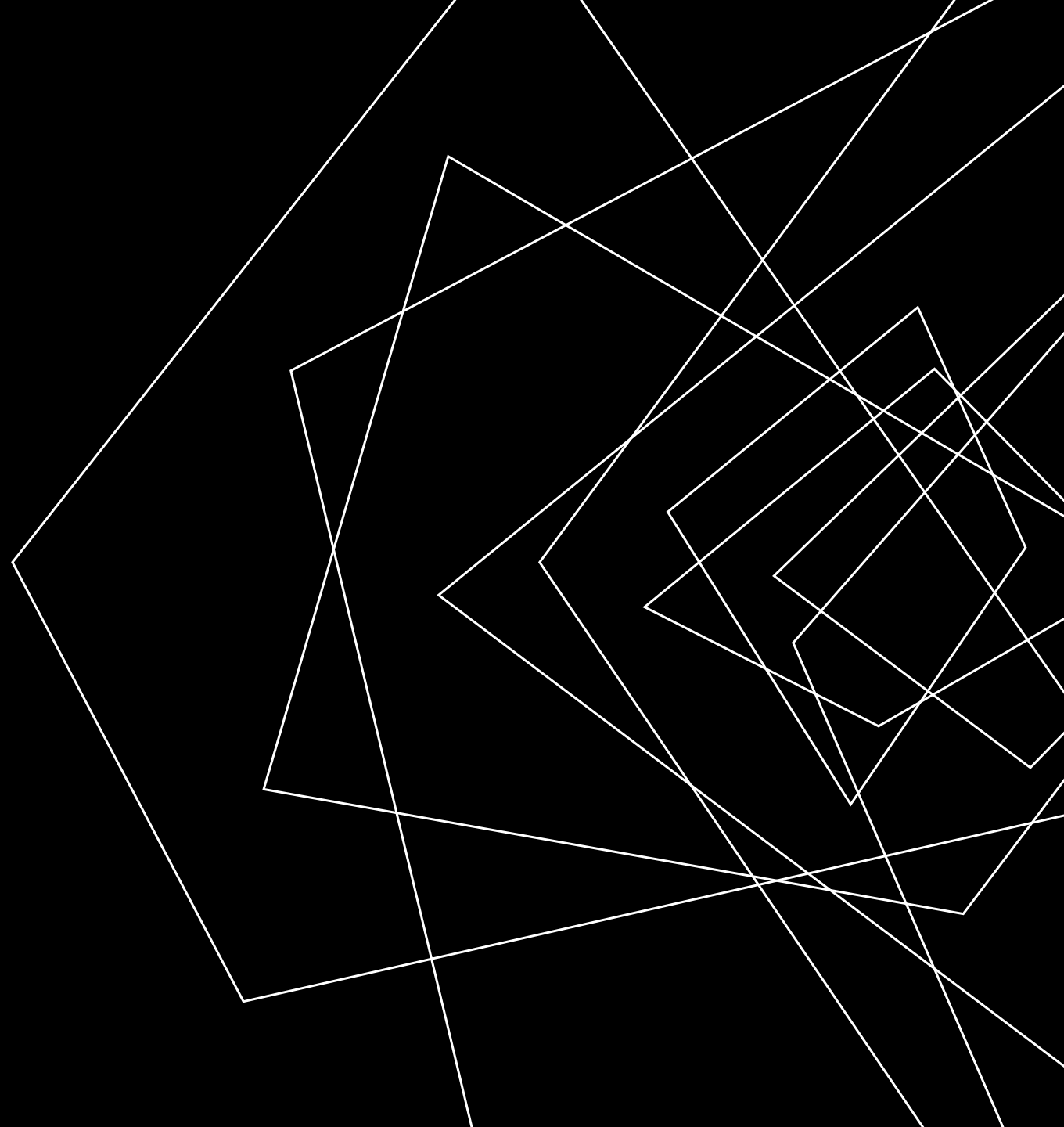
EXAMPLE-DRIVEN LEARNING

LLVM BITCODE

Before we get too lost in the details, let’s explore bit-code with an example

LECTURE OUTLINE

- LLVM Bitcode Format
- Very simple examples
- SSA Format



AN EXAMPLE PROGRAM

LLVM BITCODE

Source code

```
int main(){
        return 7;
}
```

Basically-equivalent bit-code

```
define i32 @main(){
        ret i32 7
}
```

@ precedes a function Name

Types explicitly denote their bit size (i32)

Operands are prefixed by a type annotation `ret i32 7`

AN EXAMPLE PROGRAM – MATH

LLVM BITCODE

Source code

```
int main(int argc){
        return argc + 5;
}
```

Basically-equivalent bit-code

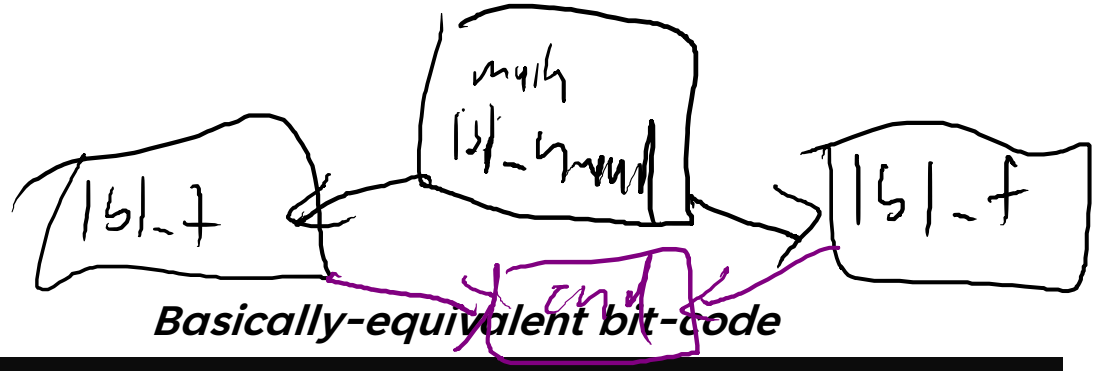
```
define i32 @main(i32 %argc) {
        %val = add i32 %argc, 5
        ret i32 %val
}
```

% precedes a register name

No complex operands (the operand of the return cannot be the add)

AN EXAMPLE PROGRAM - JUMPS

LLVM BITCODE



Source code

```
int main(int argc){
        if (argc == 1){
                return 1;
        } else {
                return 2;
        }
}
```

Basically-equivalent bit-code

```
define i32 @main(i32 %argc) {
lbl_head:
        %noArgs = icmp eq i32 %argc, 1
        br i1 %noArgs, label %lbl_t, label %lbl_f
lbl_t:
        ret i32 1
lbl_f:
        ret i32 2
}
```

All blocks must end in a terminator instruction

SIMPLE INSTRUCTION SET

LLVM BITCODE – VERY SIMPLE EXAMPLES

MATH

The `add` instruction for addition
The `mul` instruction for multiplication
The `sub` instruction for subtraction
The `div` instruction for division

CONTROL FLOW

The `br` instruction for branching

- Predicate + multiple targets for conditional branch
- No predicate + 1 target for unconditional branch

COMPARISON

The `icmp <kind>` for integer comparison
Where kind is...
`eq`: equal
`ne`: not equal
`ugt`: unsigned greater than
`uge`: unsigned greater or equal
`ult`: unsigned less than
`ule`: unsigned less or equal
`sgt`: signed greater than
`sge`: signed greater or equal
`slt`: signed less than
`sle`: signed less or equal

RUNNING BITCODE PROGRAMS

LLVM BITCODE – VERY SIMPLE EXAMPLES

LLI – A RUNTIME ENVIRONMENT FOR BIT-CODE PROGRAMS!

RUNNING BITCODE PROGRAMS

LLVM BITCODE – VERY SIMPLE EXAMPLES

```
define i32 @main(){
        ret i32 7
}
```

```
$: lli ret7.l
$: echo $?
7
```

LLI – A RUNTIME ENVIRONMENT FOR BIT-CODE PROGRAMS!

SECTION SUMMARY

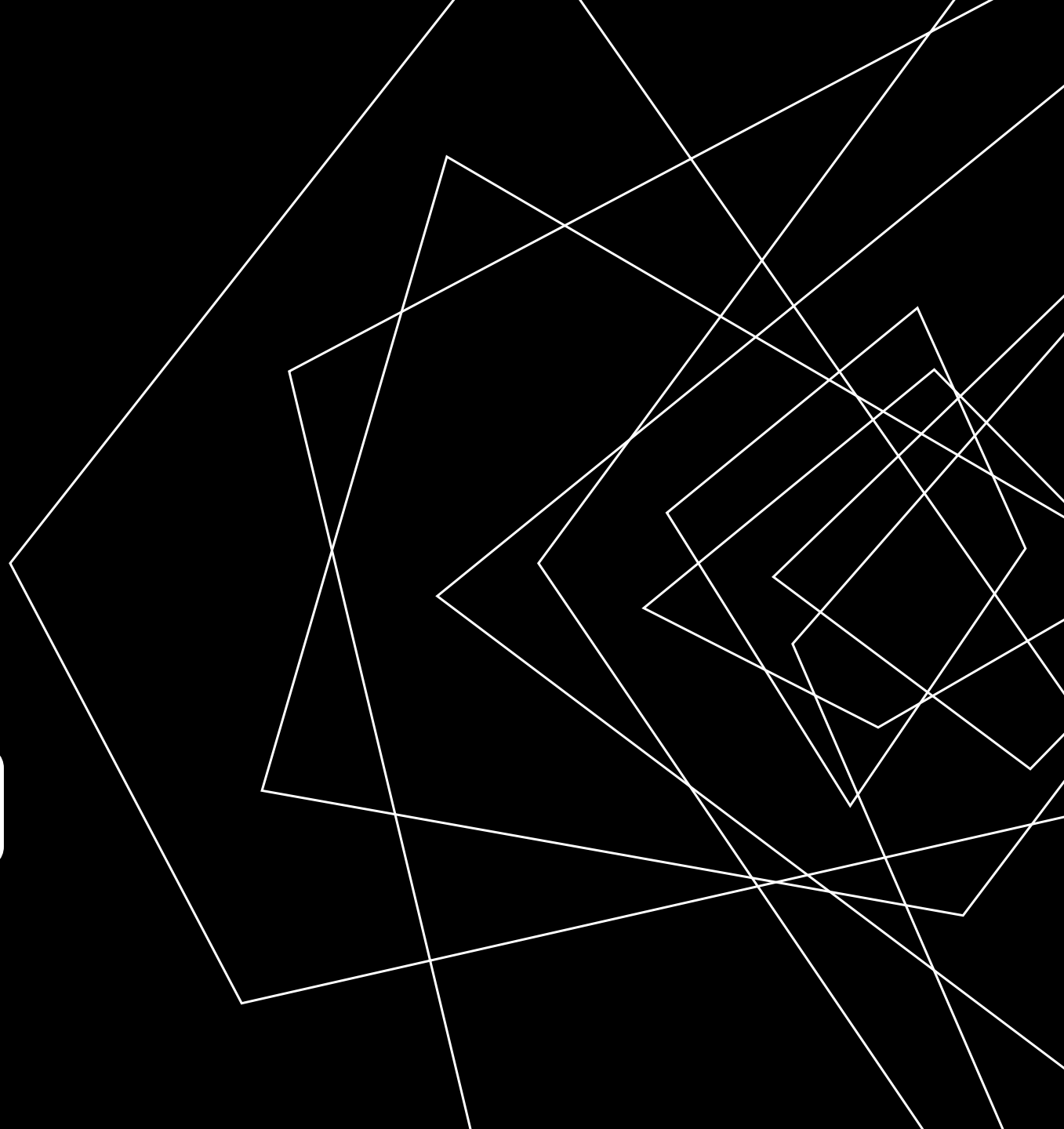
LLVM BITCODE – VERY SIMPLE EXAMPLES

We can write Simple programs using the instructions given

We can write Run simple programs using LLI

LECTURE OUTLINE

- LLVM Bitcode Format
- Very simple examples
- Format Constraints - SSA



AN INCORRECT PROGRAM

LLVM BITCODE –FORMAT CONSTRAINTS: SSA

THIS PROGRAM IS INVALID!

```
define i32 @main(i32 %0) {
        %reg = add i32 %0, 5
        %reg = add i32 %0, 5
        ret i32 %2
}
```

THE REGISTER %REG IS NOT IS NOT IN **SSA FORM**

```
lli: badSSA.ll:3:2: error: multiple def
inition of local value named 'reg'
        %reg = add i32 %0, 5
        ^
```

SSA FORM

LLVM BITCODE –FORMAT CONSTRAINTS: SSA

In static single assignment form, a variable (here, register) may be assigned in at most one program point

```
define i32 @main(i32 %argc) {
        %v1 = add i32 %argc, 1
        %v1 = mul i32 %v1, 7
        %v1 = sub i32 %v1, 2
        ret i32 %v1
}
```

```
define i32 @main(i32 %argc) {
        %v1 = add i32 %argc, 1
        %v2 = mul i32 %v1, 7
        %v3 = sub i32 %v2, 2
        ret i32 %v3
}
```

SSA FORM

LLVM BITCODE –FORMAT CONSTRAINTS: SSA

IN STATIC SINGLE ASSIGNMENT FORM, A VARIABLE (HERE, REGISTER) MAY BE ASSIGNED IN AT MOST ONE PROGRAM POINT

Is this program in SSA form? **Yes!**

Remember static means “before runtime”
only one static assignment
(many dynamic assignments)

```
define i32 @main(i32 %argc) {
loop:
        %v1 = add i32 %argc, 1
        br label %loop
}
```

Is this program in SSA form? **No!**

var is assigned at two program points

```
define i32 @main(i32 %argc) {
lbl_head: %noArgs = icmp eq i32 %argc, 1
        br i1 %noArgs, label %lbl_t, label %lbl_f
lbl_t: %var = add i32 1, 0
        br label %end
lbl_f: %var = add i32 2, 0
        br label %end
end: ret i32 %var
}
```

PHI FUNCTIONS

LLVM BITCODE –FORMAT CONSTRAINTS: SSA

DEALING WITH "JOINED" VALUES

Can the following program be made into SSA?

```
1 int main(int argc){
2     while (argc > 0){
3         argc = argc - 1;
4     }
5     return 0;
6 }
```

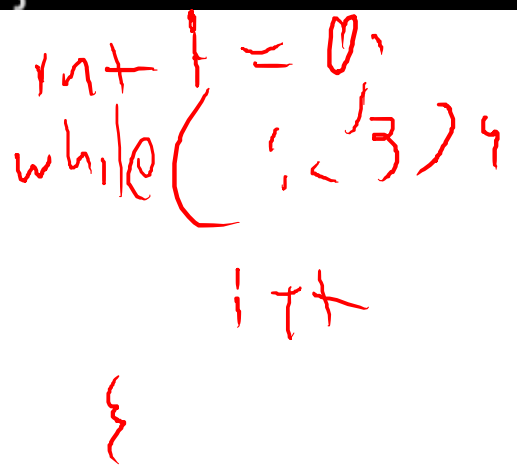
PHI FUNCTIONS

LLVM BITCODE –FORMAT CONSTRAINTS: SSA

`%res = phi <type> [val1, bbl1], [val2, bbl2], … [valn, bbln]`

Set %res to val_i if the block was entered from bbl_i

```
1 int main(int argc){
2     while (argc > 0){
3         argc = argc - 1;
4     }
5     return 0;
6 }
```




```
1  define i32 @main(i32 %argc) {
2  entry:
3      %i_init = add i32 0, 0
4      br label %loop_head
5  loop_head:
6      %i_join = phi i32 [%i_init, %entry], [%i_loop, %loop_body]
7      %done = icmp slt i32 %i_join, 3
8      br i1 %done, label %loop_body, label %loop_after
9  loop_body:
10     %i_loop = add i32 %i_join, 1
11     br label %loop_head
12 loop_after:
13     ret i32 %i_join
14 }
```

SECTION SUMMARY

STATIC ANALYSIS

LLVM CONSTRAINS HOW VALUES CAN BE SET

ONE SOLUTION IS TO USE PHI INSTRUCTIONS
TO UNIFY DISPARATE VALUES

ASIDE: FANCY SYNTAX HIGHLIGHTING

DREW'S COOL TOOLS

TIRED:

```
define i32 @main(i32 %argc) #0 {
entry:
        %res_entry = add i32 0, 1
        %cond = icmp sgt i32 %argc, 1
        br i1 %cond, label %if_body, label %after_if
if_body:
        %res_body = add i32 0, 7
        br label %after_if
after_if:
        %res_join = phi i32 [%res_entry, %entry], [%res_body, %if_body]
        ret i32 %res_join
}
```

WIRED:

```
define i32 @main(i32 %argc) #0 {
entry:
        %res_entry = add i32 0, 1
        %cond = icmp sgt i32 %argc, 1
        br i1 %cond, label %if_body, label %after_if
if_body:
        %res_body = add i32 0, 7
        br label %after_if
after_if:
        %res_join = phi i32 [%res_entry, %entry], [%res_body, %if_body]
        ret i32 %res_join
}
```

ASIDE: FANCY SYNTAX HIGHLIGHTING

DREW'S COOL TOOLS

TURNS OUT SYNTAX HIGHLIGHTERS ARE AVAILABLE FOR SEVERAL EDITORS

- vim (my personal choice)
- emacs
- vscode

FILES ARE IN THE GIT REPO

https://github.com/llvm/llvm-project
Under the directory llvm/utils

If you don't want to download all that code, check
https://analysis.cool/llvm-syntax.tgz

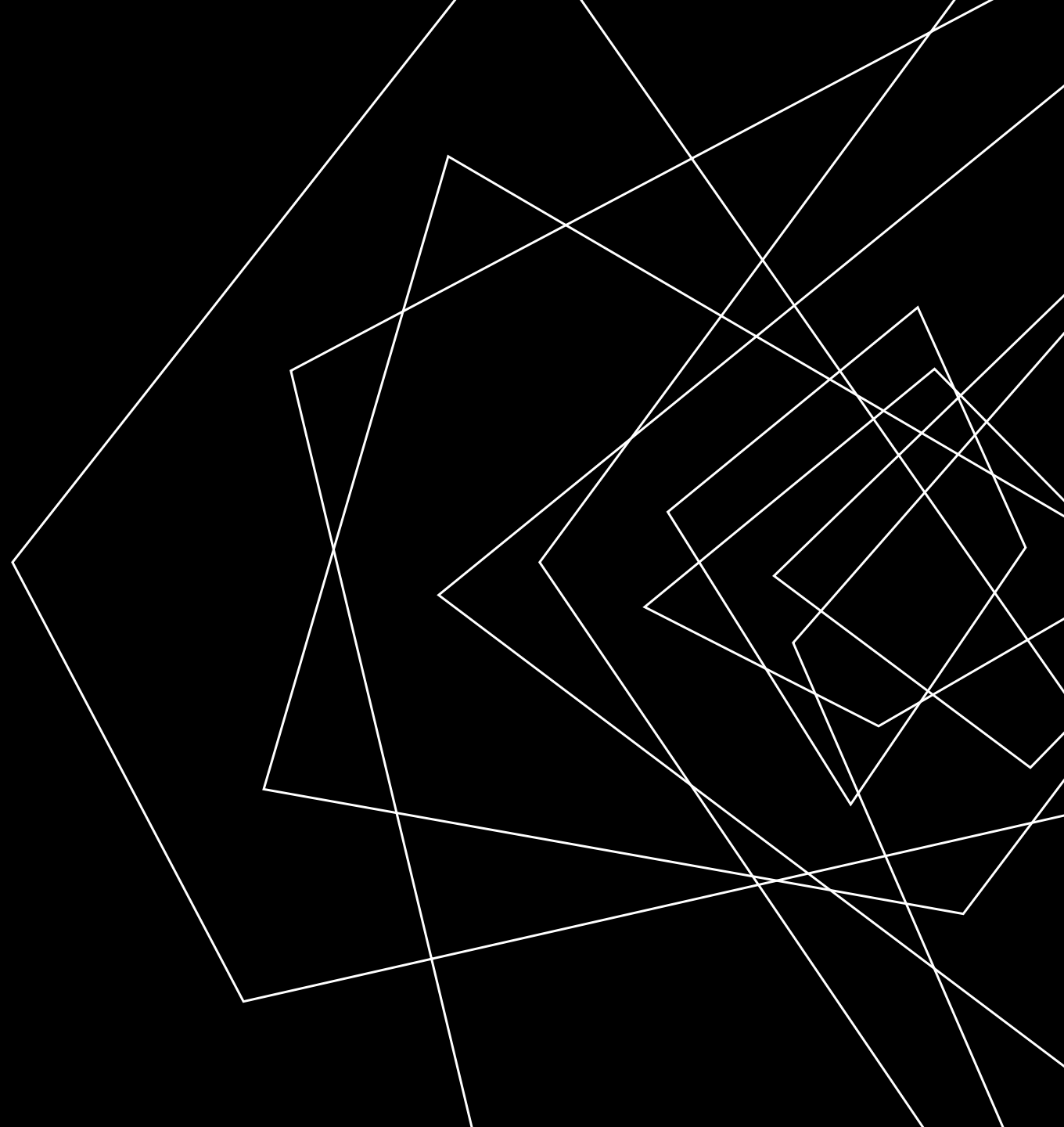
TIRED:

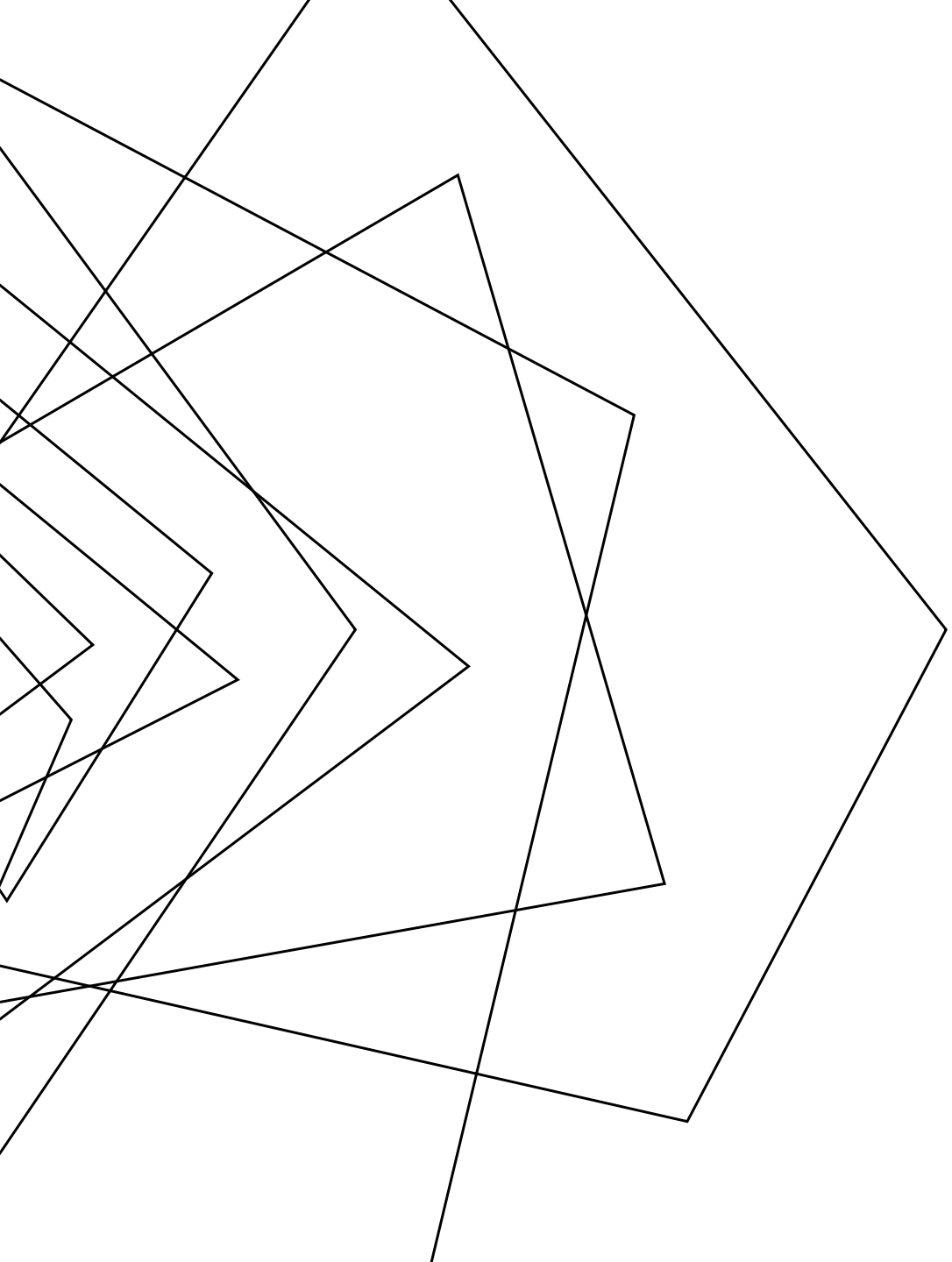
```
define i32 @main(i32 %argc) #0 {
entry:
        %res_entry = add i32 0, 1
        %cond = icmp sgt i32 %argc, 1
        br i1 %cond, label %if_body, label %after_if
if_body:
        %res_body = add i32 0, 7
        br label %after_if
after_if:
        %res_join = phi i32 [%res_entry, %entry], [%res_body, %if_body]
        ret i32 %res_join
}
```

WIRED:

```
define i32 @main(i32 %argc) #0 {
entry:
        %res_entry = add i32 0, 1
        %cond = icmp sgt i32 %argc, 1
        br i1 %cond, label %if_body, label %after_if
if_body:
        %res_body = add i32 0, 7
        br label %after_if
after_if:
        %res_join = phi i32 [%res_entry, %entry], [%res_body, %if_body]
        ret i32 %res_join
}
```

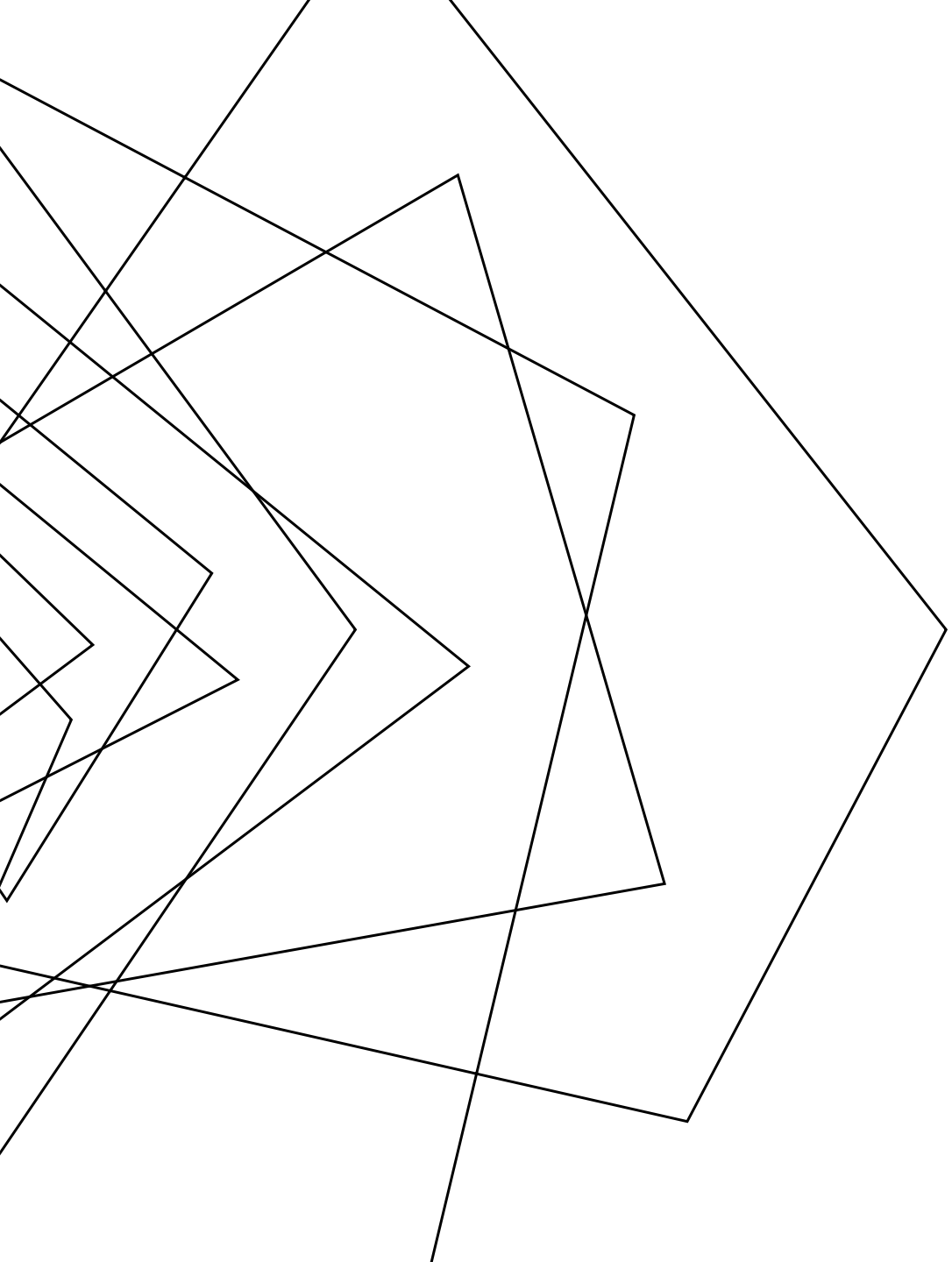
WRAP-UP





HOMEWORK 1
DUE FRIDAY, 9/4

WRITE AN LLVM PROGRAM THAT ITERATIVELY COMPUTES THE K^{TH} FIBONACCI NUMBER WHERE K IS THE ARG COUNT TO THE PROGRAM



NEXT TIME

LOOK AT SOME MORE COMPLEX LLVM EXAMPLES

START LOOKING AT MANIPULATING MEMORY:

- POINTERS / REF+DEREF
- STRUCTURES / ARRAYS