Mini Assignment 3
cs19btech11020

Code 1
Factorial :

```
int main(){
   int i,n;
   int res = 1;
         for (i = 2; i <= n; i++)
                  res *= i;
         return res;
}
```

AST:

```
(base) sharanya@sharanya-Swift-SF314-55G:~/Desktop/Compilers mini assign 3$ clang -Xclang -ast-dump -fsyntax-only code1.c
TranslationUnitDecl 0xb782e8 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0xb78b80 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
| `-BuiltinType 0xb78880 '__int128'
|-TypedefDecl 0xb78bf0 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
| `-BuiltinType 0xb788a0 'unsigned __int128'
|-TypedefDecl 0xb78ef8 <<invalid sloc>> <invalid sloc> implicit __NSConstantString 'struct __NSConstantString_tag'
| `-RecordType 0xb78cd0 'struct __NSConstantString_tag'
|   `-Record 0xb78c48 '__NSConstantString_tag'
|-TypedefDecl 0xb78f90 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list 'char *'
| `-PointerType 0xb78f50 'char *'
|   `-BuiltinType 0xb78380 'char'
|-TypedefDecl 0xb79288 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list 'struct __va_list_tag [1]'
| `-ConstantArrayType 0xb79230 'struct __va_list_tag [1]' 1
|   `-RecordType 0xb79070 'struct __va_list_tag'
|     `-Record 0xb78fe8 '__va_list_tag'
`-FunctionDecl 0xbd7fd0 <code1.c:1:1, line:8:1> line:1:5 main 'int ()'
  `-CompoundStmt 0xbd84d0 <line:2:1, line:8:1>
    |-DeclStmt 0xbd81d0 <line:3:2, col:9>
    | |-VarDecl 0xbd80d0 <col:2, col:6> col:6 used i 'int'
    | `-VarDecl 0xbd8150 <col:2, col:8> col:8 used n 'int'
    |-DeclStmt 0xbd8288 <line:4:2, col:13>
    | `-VarDecl 0xbd8200 <col:2, col:12> col:6 used res 'int' cinit
    |   `-IntegerLiteral 0xbd8268 <col:12> 'int' 1
    |-ForStmt 0xbd8450 <line:5:5, line:6:16>
    | |-BinaryOperator 0xbd82e0 <line:5:10, col:14> 'int' '='
    | | |-DeclRefExpr 0xbd82a0 <col:10> 'int' lvalue Var 0xbd80d0 'i' 'int'
    | | `-IntegerLiteral 0xbd82c0 <col:14> 'int' 2
    | |-<<<NULL>>>
    | |-BinaryOperator 0xbd8370 <col:17, col:22> 'int' '<='
    | | |-ImplicitCastExpr 0xbd8340 <col:17> 'int' <LValueToRValue>
    | | | `-DeclRefExpr 0xbd8300 <col:17> 'int' lvalue Var 0xbd80d0 'i' 'int'
    | | `-ImplicitCastExpr 0xbd8358 <col:22> 'int' <LValueToRValue>
    | |   `-DeclRefExpr 0xbd8320 <col:22> 'int' lvalue Var 0xbd8150 'n' 'int'
    | |-UnaryOperator 0xbd83b0 <col:25, col:26> 'int' postfix '++'
    | | `-DeclRefExpr 0xbd8390 <col:25> 'int' lvalue Var 0xbd80d0 'i' 'int'
    | `-CompoundAssignOperator 0xbd8420 <line:6:9, col:16> 'int' '*=' ComputeLHSTy='int' ComputeResultTy='int'
    |   |-DeclRefExpr 0xbd83c8 <col:9> 'int' lvalue Var 0xbd8200 'res' 'int'
    |   `-ImplicitCastExpr 0xbd8408 <col:16> 'int' <LValueToRValue>
    |     `-DeclRefExpr 0xbd83e8 <col:16> 'int' lvalue Var 0xbd80d0 'i' 'int'
    `-ReturnStmt 0xbd84c0 <line:7:5, col:12>
      `-ImplicitCastExpr 0xbd84a8 <col:12> 'int' <LValueToRValue>
        `-DeclRefExpr 0xbd8488 <col:12> 'int' lvalue Var 0xbd8200 'res' 'int'
```

- At first we have a function declaration of the main function of return type int.
- The next node is compound staments corresponding to the statement block in the code ie. contents of main.
- Next we have 2 Declaration statements containing VarDecl for i and n of type int.
- The other declaration statement is for variable res,and its value 1 an integer literal .
- Next we have node for FOR loop, containing binary operator ir assignment(=), and declare refexpression that is assignment of 2 to i,, and implicit coast expression to check for condition and the unary operator for incrementing iterator in the for loop
- Then inside for loop we have compound assignment operator *= where the result is computed.
- At last we have the return stmt of the main function returning int.

IR:

```
; ModuleID = 'code1.c'
source_filename = "code1.c"
```

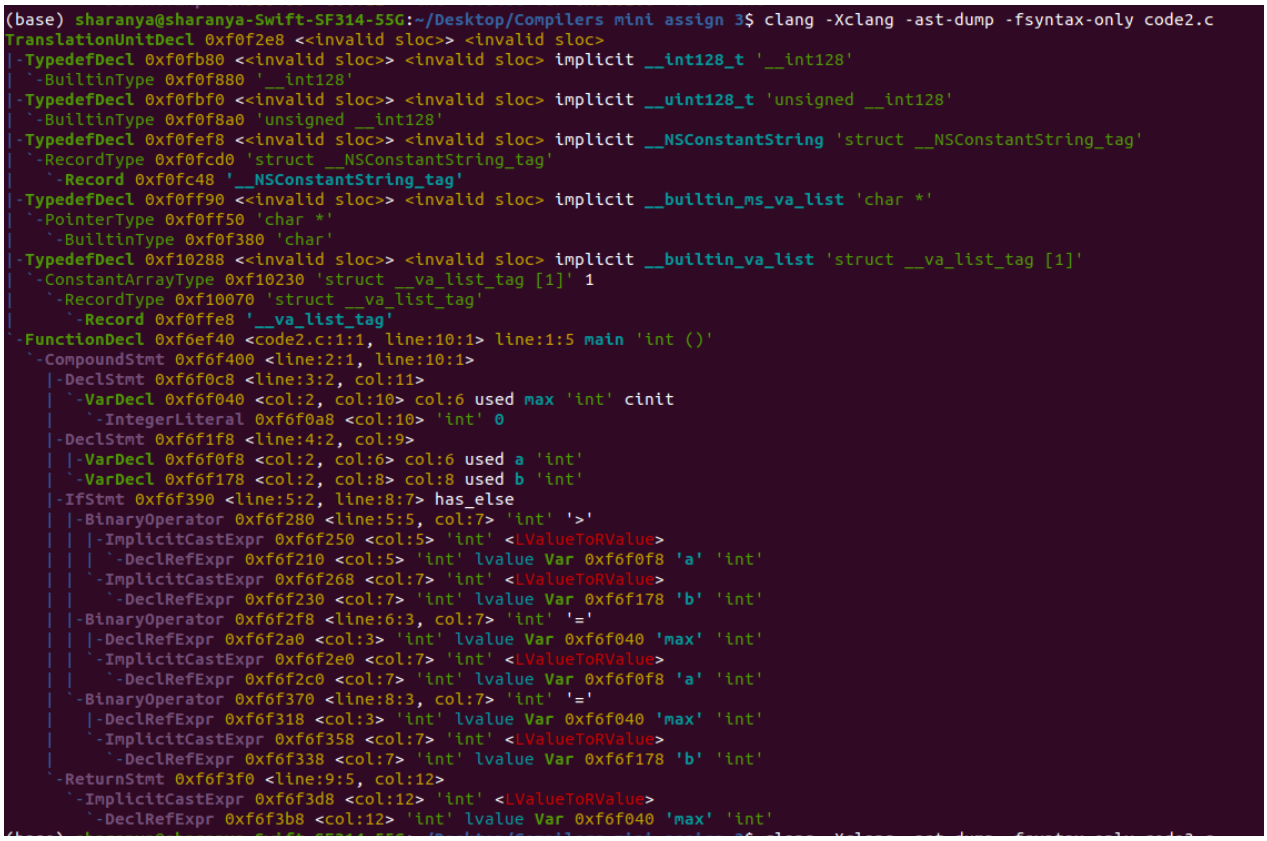
```
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"
; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 1, i32* %4, align 4
  store i32 2, i32* %2, align 4
  br label %5

5:                                                ; preds = %13, %0
  %6 = load i32, i32* %2, align 4
  %7 = load i32, i32* %3, align 4
  %8 = icmp sle i32 %6, %7
  br i1 %8, label %9, label %16

9:                                                ; preds = %5
  %10 = load i32, i32* %2, align 4
  %11 = load i32, i32* %4, align 4
  %12 = mul nsw i32 %11, %10
  store i32 %12, i32* %4, align 4
  br label %13

13:                                               ; preds = %9
  %14 = load i32, i32* %2, align 4
  %15 = add nsw i32 %14, 1
  store i32 %15, i32* %2, align 4
  br label %5

16:                                               ; preds = %5
  %17 = load i32, i32* %4, align 4
  ret i32 %17
}
attributes #0 = { noinline nounwind optnone uwtable "correctly-rounded-divide-sqrt-fp-math"="false"
"disable-tail-calls"="false" "frame-pointer"="all" "less-precise-fpmad"="false" "min-legal-vector-width"="0"
"no-infs-fp-math"="false" "no-jump-tables"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false"
"no-trapping-math"="false" "stack-protector-buffer-size"="8" "target-cpu"="x86-64"
"target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 10.0.0-4ubuntu1 "}
```

- `define dso_local i32 @main() #0` is to declare main function. Then we allocate space for the variables i,n, res, 1 using alloca.
- We store the 2 in variable i in %2 and n in %3. And we store 1 in res ie %4.
- We have for loop and break statement, so in for loop if the condition i<=n is true we jump to 9 or if the condition fails we come to label 16: `br i1 %8, label %9, label %16`, now if true we load and update the res by multiplying with i and storing it back in res `%12 = mul nsw i32 %11, %10` and we break and go to for again to 15: and increment the value of i in `%15 = add nsw i32 %14, 1,` and go back to 5 and compare the i value `%8 = icmp sle i32 %6, %7`.
- %17 is for res ,we load the value %4.
- In 16: we have rt %17 that is returning the result value.

Code 2
If else:

```
int main(){
   int max=0;
   int a,b;
   if(a>b)
          max=a;
   else
          max=b;
         return max;
}
```

AST:



```
(base) sharanya@sharanya-Swift-SF314-55G:~/Desktop/Compilers mini assign 3$ clang -Xclang -ast-dump -fsyntax-only code2.c
TranslationUnitDecl 0xf0f2e8 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0xf0fb80 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
| `-BuiltinType 0xf0f880 '__int128'
|-TypedefDecl 0xf0fbf0 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
| `-BuiltinType 0xf0f8a0 'unsigned __int128'
|-TypedefDecl 0xf0fef8 <<invalid sloc>> <invalid sloc> implicit __NSConstantString 'struct __NSConstantString_tag'
| `-RecordType 0xf0fcd0 'struct __NSConstantString_tag'
|   `-Record 0xf0fc48 '__NSConstantString_tag'
|-TypedefDecl 0xf0ff90 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list 'char *'
| `-PointerType 0xf0ff50 'char *'
|   `-BuiltinType 0xf0f380 'char'
|-TypedefDecl 0xf10288 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list 'struct __va_list_tag [1]'
| `-ConstantArrayType 0xf10230 'struct __va_list_tag [1]' 1
|   `-RecordType 0xf10070 'struct __va_list_tag'
|     `-Record 0xf0ffe8 '__va_list_tag'
`-FunctionDecl 0xf6ef40 <code2.c:1:1, line:10:1> line:1:5 main 'int ()'
  `-CompoundStmt 0xf6f400 <line:2:1, line:10:1>
    |-DeclStmt 0xf6f0c8 <line:3:2, col:11>
    | `-VarDecl 0xf6f040 <col:2, col:10> col:6 used max 'int' cinit
    |   `-IntegerLiteral 0xf6f0a8 <col:10> 'int' 0
    |-DeclStmt 0xf6f1f8 <line:4:2, col:9>
    | |-VarDecl 0xf6f0f8 <col:2, col:6> col:6 used a 'int'
    | `-VarDecl 0xf6f178 <col:2, col:8> col:8 used b 'int'
    |-IfStmt 0xf6f390 <line:5:2, line:8:7> has_else
    | |-BinaryOperator 0xf6f280 <line:5:5, col:7> 'int' '>'
    | | |-ImplicitCastExpr 0xf6f250 <col:5> 'int' <LValueToRValue>
    | | | `-DeclRefExpr 0xf6f210 <col:5> 'int' lvalue Var 0xf6f0f8 'a' 'int'
    | | `-ImplicitCastExpr 0xf6f268 <col:7> 'int' <LValueToRValue>
    | |   `-DeclRefExpr 0xf6f230 <col:7> 'int' lvalue Var 0xf6f178 'b' 'int'
    | |-BinaryOperator 0xf6f2f8 <line:6:3, col:7> 'int' '='
    | | |-DeclRefExpr 0xf6f2a0 <col:3> 'int' lvalue Var 0xf6f040 'max' 'int'
    | | `-ImplicitCastExpr 0xf6f2e0 <col:7> 'int' <LValueToRValue>
    | |   `-DeclRefExpr 0xf6f2c0 <col:7> 'int' lvalue Var 0xf6f0f8 'a' 'int'
    | `-BinaryOperator 0xf6f370 <line:8:3, col:7> 'int' '='
    |   |-DeclRefExpr 0xf6f318 <col:3> 'int' lvalue Var 0xf6f040 'max' 'int'
    |   `-ImplicitCastExpr 0xf6f358 <col:7> 'int' <LValueToRValue>
    |     `-DeclRefExpr 0xf6f338 <col:7> 'int' lvalue Var 0xf6f178 'b' 'int'
    `-ReturnStmt 0xf6f3f0 <line:9:5, col:12>
      `-ImplicitCastExpr 0xf6f3d8 <col:12> 'int' <LValueToRValue>
        `-DeclRefExpr 0xf6f3b8 <col:12> 'int' lvalue Var 0xf6f040 'max' 'int'
```

- Here also we have function declaration in beginning followed by compound statement block which contains declaration statements, if statement and return statement.
- In declaration statements max, a, b are variable declarations
- the if statement node is invoked by a binary operator > of the condition, if it's true then we do assignment , update max value to a. Otherwise we move to the else block and since we are assigning max as b we invoke a binary operator, implicit cast expression.
- At last we have return stmt to return int value

IR:

```
source_filename = "code2.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-pc-linux-gnu"

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
```

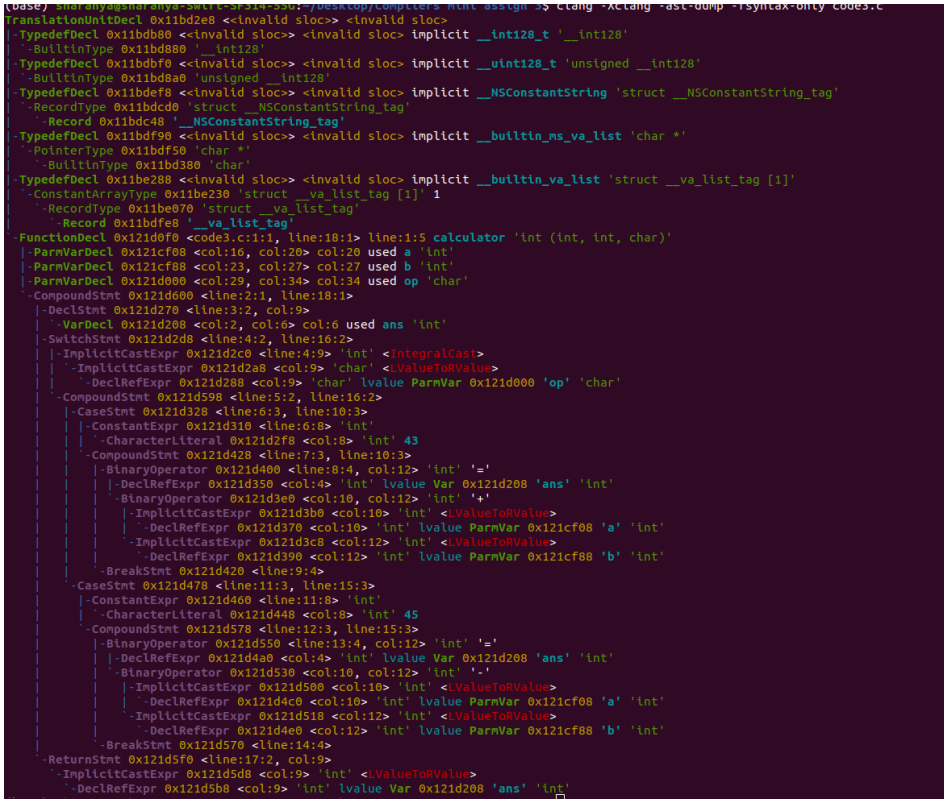
```
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 0, i32* %2, align 4
  %5 = load i32, i32* %3, align 4
  %6 = load i32, i32* %4, align 4
  %7 = icmp sgt i32 %5, %6
  br i1 %7, label %8, label %10

8:                                                ; preds = %0
  %9 = load i32, i32* %3, align 4
  store i32 %9, i32* %2, align 4
  br label %12

10:                                               ; preds = %0
  %11 = load i32, i32* %4, align 4
  store i32 %11, i32* %2, align 4
  br label %12

12:                                               ; preds = %10, %8
  %13 = load i32, i32* %2, align 4
  ret i32 %13
}

attributes #0 = { noinline nounwind optnone uwtable "correctly-rounded-divide-sqrt-fp-math"="false"
"disable-tail-calls"="false" "frame-pointer"="all" "less-precise-fpmad"="false" "min-legal-vector-width"="0"
"no-infs-fp-math"="false" "no-jump-tables"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false"
"no-trapping-math"="false" "stack-protector-buffer-size"="8" "target-cpu"="x86-64"
"target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 10.0.0-4ubuntu1 "}
```

- Here we have function declaration of main in beginning `define dso_local i32 @main().`
- Then we allocate space for max,0,a,b. Then we store 0 in max i e %2 `store i32 0, i32* %2, align 4.`
- Then we load a,b, max into %5,%6 registers. Using load
- In %7 we store the result of comparison of the if condition ie a>b `%7 = icmp sgt i32 %5, %6.`
- Then we check for the condition if its true we jump to label 8 otherwise we jump to label 10 ie else block. `br i1 %7, label %8, label %10`
- In label 8 we load a in %9, and we update max ny calling store.`store i32 %9, i32* %2, align 4`
- Similarly for label 10 also,
- After updating we jump to label 12 here we load max value to %13 and return it `ret i32 %13.`

Code 3
Switch case:

```
int calculator(int a, int b , char op){
    int ans;
    switch(op) {
            case '+':
            {
                    ans = a+b;
                    break;
            }
            case '-':
            {
                    ans = a-b;
                    break;
            }
    }
    return ans;
}
```

AST:



- First we have function declaration of name calculator returning type int and h=take two integers and a character.
- In function declaration we have parameter variables declarations of a,b,op followed by function body i.e is compound stmts.
- In the function block we have variable declaration of ans of type int; then we have switch statement.
- Next we have switch statement so we invoke SwitchStmt over x so we invoke DeclRefExpr .

- Next we have 2 cases so we invoke caseStmt we have char +,- so invoke charLiteral 43 ascii of + is 43 . In cse + we have a+b so we binaryoperator. We have assignment to we invoke declrefexpr. Finally we have break statement so we invoke BreakStmt.
- After switch stmt we have return statement of the function.

IR:

```
; ModuleID = 'code3.c'
source_filename = "code3.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-pc-linux-gnu"

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @calculator(i32 %0, i32 %1, i8 signext %2) #0 {
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  %6 = alloca i8, align 1
  %7 = alloca i32, align 4
  store i32 %0, i32* %4, align 4
  store i32 %1, i32* %5, align 4
  store i8 %2, i8* %6, align 1
  %8 = load i8, i8* %6, align 1
  %9 = sext i8 %8 to i32
  switch i32 %9, label %18 [
            i32 43, label %10
            i32 45, label %14
  ]

10:                                               ; preds = %3
  %11 = load i32, i32* %4, align 4
  %12 = load i32, i32* %5, align 4
  %13 = add nsw i32 %11, %12
  store i32 %13, i32* %7, align 4
  br label %18

14:                                               ; preds = %3
  %15 = load i32, i32* %4, align 4
  %16 = load i32, i32* %5, align 4
  %17 = sub nsw i32 %15, %16
  store i32 %17, i32* %7, align 4
  br label %18

18:                                               ; preds = %3, %14, %10
  %19 = load i32, i32* %7, align 4
  ret i32 %19
}

attributes #0 = { noinline nounwind optnone uwtable "correctly-rounded-divide-sqrt-fp-math"="false"
"disable-tail-calls"="false" "frame-pointer"="all" "less-precise-fpmad"="false" "min-legal-vector-width"="0"
"no-infs-fp-math"="false" "no-jump-tables"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false"
"no-trapping-math"="false" "stack-protector-buffer-size"="8" "target-cpu"="x86-64"
"target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 10.0.0-4ubuntu1 "}
```

- First is the function declaration of calculator efine dso_local i32 @calculator(i32 %0, i32 %1, i8 signext %2) of parameter of type int int and char. Ie i32 adn i8 signext
- We have a in %0, b in %1 and op in %2
- We allocate space for ans, op a, b using alloca.
- We store values of a , b, to %4,%5 store i32 %0, i32* %4, align 4 and op to %6
- Now we load op to check for case in switch stmt.
- We call switch statement check for conditions if case + ie ascii 43 we goto label 10 else if - we got to label 14 otherwise we goto label 18 come out of switch block switch i32 %9, label %18 [32 43, label %10 i32 45, label %14]
- In label 10 we load a,b values and add them and store them in ans store i32 %17, i32* %7, align 4, after that we call break
- In label 14 we load a,b values and subract them %17 = sub nsw i32 %15, %16 and store them in ans store i32 %17, i32* %7, align 4, after that we call break

Code 4
Find gcd using while loop

```
int main(){
        int n1, n2;
        while(n1!=n2){
        if(n1 > n2)
                n1 -= n2;
        else
                n2 -= n1;
        }
        int gcd=n1;

        return 0;
}
```

● In our code we have declared n1,n2 and then we have a while loop over n!=n2 inside which we increment value of n1 or n2.
● So functionDecl() is invoked for “int main()” . then we have compound statements below so we invoke Compoundstmt then we declare variable n1 , so we invoke Vardecl. Similarly for n2.
● Then we have while loop() so we invoke whilestmt we have n!=n2 inside while
which is binary operator i.e, != on a so declRefExpr.
● Next we have a compound statement containing if statements hence we invoke Compoundstmt under which be have if stmt that has binary operator > to compare n1 and n2 so we invoke binaryOperator “>”, then we have compound assign operator to update n1 and n2 values.
Then we have a variable decls statement for variable gcd and declrefexpr for assigning it value . Finally a return statement for the main function.

IR:

```
; ModuleID = 'code4.c'
source_filename = "code4.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-pc-linux-gnu"

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  br label %5

5:                                                ; preds = %21, %0
  %6 = load i32, i32* %2, align 4
  %7 = load i32, i32* %3, align 4
  %8 = icmp ne i32 %6, %7
  br i1 %8, label %9, label %22

9:                                                ; preds = %5
  %10 = load i32, i32* %2, align 4
  %11 = load i32, i32* %3, align 4
  %12 = icmp sgt i32 %10, %11
  br i1 %12, label %13, label %17

13:                                               ; preds = %9
  %14 = load i32, i32* %3, align 4
  %15 = load i32, i32* %2, align 4
  %16 = sub nsw i32 %15, %14
  store i32 %16, i32* %2, align 4
  br label %21

17:                                               ; preds = %9
  %18 = load i32, i32* %2, align 4
  %19 = load i32, i32* %3, align 4
  %20 = sub nsw i32 %19, %18
  store i32 %20, i32* %3, align 4
  br label %21

21:                                               ; preds = %17, %13
```

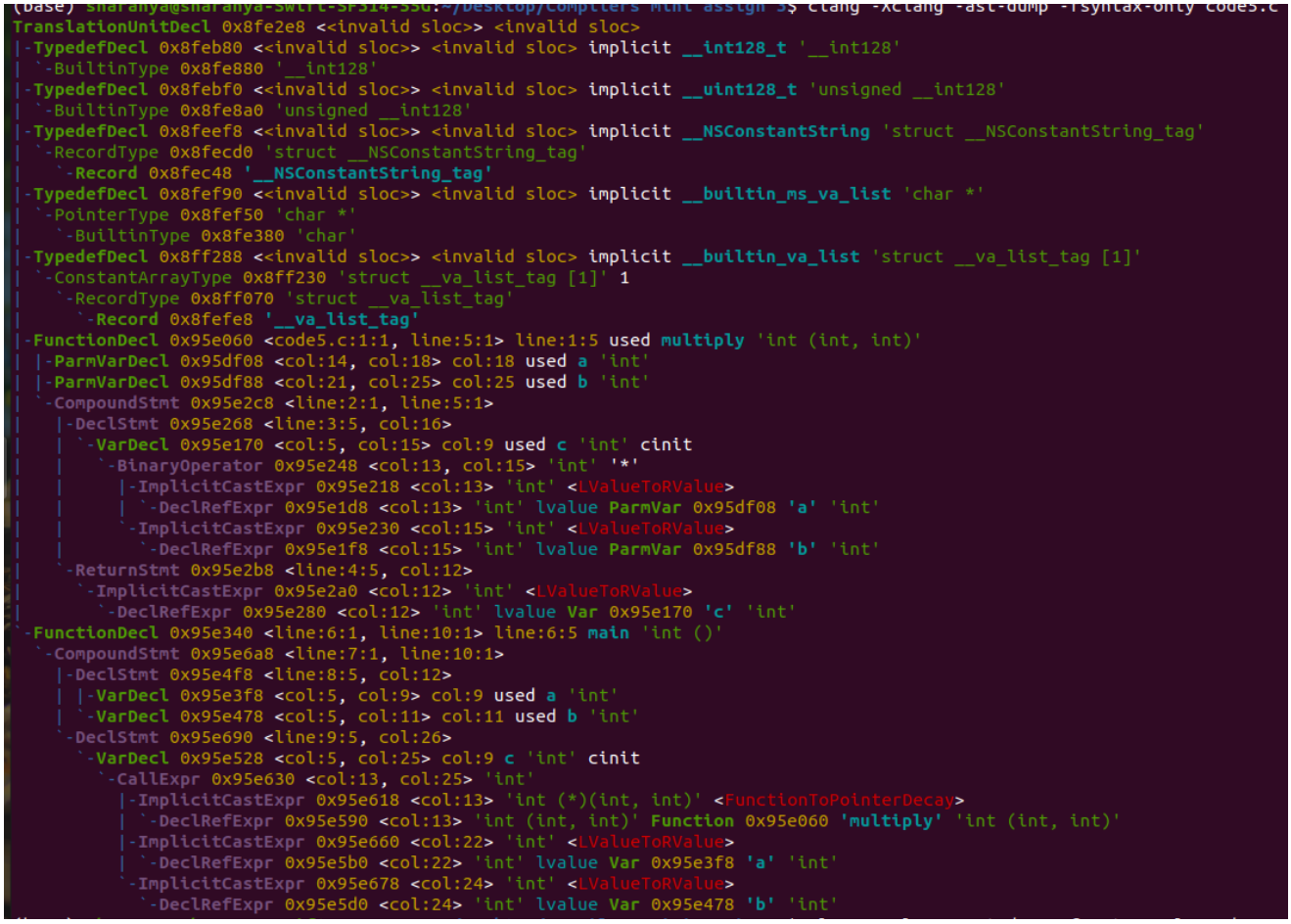
```
  br label %5

22:                                               ; preds = %5
  %23 = load i32, i32* %2, align 4
  store i32 %23, i32* %4, align 4
  ret i32 0
}

attributes #0 = { noinline nounwind optnone uwtable "correctly-rounded-divide-sqrt-fp-math"="false"
"disable-tail-calls"="false" "frame-pointer"="all" "less-precise-fpmad"="false" "min-legal-vector-width"="0"
"no-infs-fp-math"="false" "no-jump-tables"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false"
"no-trapping-math"="false" "stack-protector-buffer-size"="8" "target-cpu"="x86-64"
"target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 10.0.0-4ubuntu1 "}
```

- First we have function declaration of main
- Then we allocate space for n1,n2 and variables
- Then we have while loop , we loas n1, n2 and `%8 = icmp ne i32 %6, %7` compare if they are not equal, if they are not equal then we go to label 9 otherwise we go to label 22 `br i1 %8, label %9, label %22` that is outside while loop
- In label 9 we have if condition so here we load n1 and n2 again from %2,%3 and `%12 = icmp sgt i32 %10, %11` compare them `br i1 %12, label %13, label %17` if the comparison is true we jump to label 13 else we jump to label 17.
- In label 13 we loadn1 and n2 and subtract from n1 and store the value in n1.
- Similarly in label 17 also after that we jump to label 21
- In label 21 we jump to label 5 `br label %5` ie while condtiion again.
- Finally we return 0.

Code 5

```
int multiply(int a, int b){
        int c = a*b;
        return c;
}
int main(){
        int a,b;
        int c = multiply(a,b);
}
```

AST:



●control flow: function dels-> paramsdecl -> compundstmt -> Declstmt …..

● FunctionDecl is on the top level of a tree, next containing ParmVarDecl nodes followed by block of statements ,and compound statement contains variable declarations, binary operations and other nodes such as ReturnStmt.

● functionDecl() is an ast matcher that is invoked for every function declaration we have defined two functions multiply() , main() in above code so it is invoked twice.

● This matcher will focuses only on function declarations which have the name “multiply”

● The middle column indicates the name of each matcher, and the first column indicates the kind of matcher that has been nested in.

● Next we have parmVarDecl that is Parameter declarations. We can match only parameter variable declarations by using the respective AST node matcher.

● We next have compound statement i.e, c = a*b here we have variable declaration of c so we have VarDecl and also we use binary operator “ * “ so we have binaryoperator.

IR:

```
; ModuleID = 'code5.c'
source_filename = "code5.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-pc-linux-gnu"

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @multiply(i32 %0, i32 %1) #0 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  store i32 %1, i32* %4, align 4
  %6 = load i32, i32* %3, align 4
  %7 = load i32, i32* %4, align 4
  %8 = mul nsw i32 %6, %7
  store i32 %8, i32* %5, align 4
  %9 = load i32, i32* %5, align 4
  ret i32 %9
}

; Function Attrs: noinline nounwind optnone uwtable
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = load i32, i32* %1, align 4
  %5 = load i32, i32* %2, align 4
  %6 = call i32 @multiply(i32 %4, i32 %5)
  store i32 %6, i32* %3, align 4
  ret i32 0
}

attributes #0 = { noinline nounwind optnone uwtable "correctly-rounded-divide-sqrt-fp-math"="false"
"disable-tail-calls"="false" "frame-pointer"="all" "less-precise-fpmad"="false" "min-legal-vector-width"="0"
"no-infs-fp-math"="false" "no-jump-tables"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false"
"no-trapping-math"="false" "stack-protector-buffer-size"="8" "target-cpu"="x86-64"
"target-features"="+cx8,+fxsr,+mmx,+sse,+sse2,+x87" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 10.0.0-4ubuntu1 "}
```

● line 6: The function declaration is similar to C syntax (@multiply(i32 %0, i32 %1)
) This function returns a value of the type i32 and has two i32 arguments,
%0 and %1 these correspond to a, b parameters in the code.. Local identifiers need % as prefix whereas global needs @. The alloca instruction reserves space on the stack frame of the current function

● In multiply function we have a,b,c so we need to reserve space for these variable we do this using %3 = alloca i32, align 4 then we store a,b(from the parameters of function) into %3 & %4,

using store i32 %0, i32* %3, align 4 then we again load these two values into %6 & %7 and their product into %8 using %7 = load i32, i32* %4, align 4 , %8 = mul nsw i32 %6, %7 respectively and finally return %9 value using ret i32 %9.

● Next we have the main function with no parameters. We do allocation and load and then call the multiply function using %6 = call i32 @multiply(i32 %4, i32 %5) .

Assembly output:

```
        .text
    .file   "code1.c"
    .globl  main                    # -- Begin function main
    .p2align    4, 0x90
    .type   main,@function
main:                               # @main
    .cfi_startproc
# %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
    .cfi_def_cfa_register %rbp
    movl    $0, -16(%rbp)
    movl    $1, -8(%rbp)
    movl    $2, -4(%rbp)
.LBB0_1:                            # =>This Inner Loop Header: Depth=1
    movl    -4(%rbp), %eax
    cmpl    -12(%rbp), %eax
    jg  .LBB0_4
# %bb.2:                            #   in Loop: Header=BB0_1 Depth=1
    movl    -4(%rbp), %eax
    imull   -8(%rbp), %eax
    movl    %eax, -8(%rbp)
# %bb.3:                            #   in Loop: Header=BB0_1 Depth=1
    movl    -4(%rbp), %eax
    addl    $1, %eax
    movl    %eax, -4(%rbp)
    jmp     .LBB0_1
```

The compilers change the names of variable to registers and they us $0,$1,$2 to represent the values of variables.
They use subl, imul, addl for subtract,multiply and addition.
They use move to store one value to another
They have offset to keep track of memory ,location.
Temporary (callee can change these):%rax, %r10, %r11
Parameters to function calls: %rdi, %rsi, %rdx, %rcx, %r8, %r9
%rbp is typically used as the “frame” pointer to the current function’s local variables
Return values %rax, %rdx