Group 4:

- Dilan Shaminda
- Sadaf Alvani
- Server Khalilov
- Tareq Nasif

1. Translation of MiniJava Program

In order to translate the MiniJava Program to the LLVM, we are once again using Visitor Pattern (extending **DefaultVisitor**), which goes through the MiniJava AST. There we override visit methods with parameters of the following types: **MJProgram, MJBlock, MJMainClass**. Each of these objects we are matching with corresponding instructions in LLVM.

2. Translation of Blocks of MiniJava Program

However, beyond that, we have to take care about blocks, different kind of statements and different kind of expressions that we can encounter in the MiniJava Program. To do that, we benefit from using Matcher in the following methods **getTheInstruction, getExpr, getOperator, getUnaryOperator, getType**. It helps to distinguish between different kind of statements, expressions, etc. The logic of translation is usually pretty simple: mostly, we defined the global variable **currentBlock**, to which we assign **new BasicBlock** whenever we change the block. And each of them we have to add to the variable **blocks** in order to pass to the returned program afterwards.

3. Translation of **if** statement

The idea is the following: we are creating two different blocks, which will cover true part of the **if statement** and false part of it. After that, we pass both of them to the **Branch** instruction, so both of the situations are translated.

4. Translation of **While** statement

The idea is similar to the **if** statement. Again, we are using the branch to make the decision: either we **Jump** back to the block, which is used when condition holds or we go to the rest of the code.

5. Translation of Arrays

In order to translate arrays, we referred to **struct** types in **C**. So, the idea of representing arrays in our implementation is the following:

1. We have struct Array {
   ```
       int size; // size of the array;
       int* data; // pointer to the data;
   }
   ```
2. Then, we have to calculate the amount of memory, which we need to allocate for that kind of structure. To calculate it, we have to keep in mind that sizeOf(int) equals to 4 bytes.

Therefore, our structure will require 4 bytes for the size and another 4 bytes for every element in the array (because we implement IntArrays). That means **sizeOf(int) * (1+sizeOf(Array)**).

We created our struct by using **TypeStruct** from the LLVM AST. In order to fill it with fields, we also created **ArrayList** of **StructFields** and added them to **StructFieldList** structure. After that, everything is set. To access length we can use the **GetElementPtr** with **OperandList**, which would contain **ConstInt(0)** for the referring to the array and **ConstInt(1)** for the referring to the **data int*** pointer. This information we can store in some **TemporaryVar.** In case after accessing **data,** we want to shift to specific **index** of array, again we can use **GetElementPtr** with corresponding **index.**

The important thing that we kept in mind is that because first 4 bytes in the allocated memory refer to the length of the array, whenever we use array lookups, we had to add to the index value **1,** to shift in memory correctly.