Compilers Project 3

Κώστας Χασιαλής - 1115201600195

1 First pass visitor (ClassDefinitions.java)

The first visitor collects every definitions and model them using classes.
For example, a ClassDefintion has a ClassIdentifier and ClassBody, ClassBody has methods and a method consists of a ClassMethodDeclaration and a ClassMethodBody etc.

2 Second pass visitor (IntermidiateRepresentation.java)

The second visitor produces the LLVM IR code while also doing semantic analysis on the program which is described on the previous assignment.

In order to accomplish this task, each visit method should have get some information from its the visitor method that called it. This information is packed on Argument class.

More specifically, we declare the following booleans with the corresponding properties.

- **isMethodDeclaration** : If this variable is set to true, it means that we have a VariableDeclaration inside a method, thus we need to allocate space on the stack, otherwise (its a variable declaration as Class fields) do nothing.
- **produceCode** : This is an indicator to the method "Identifier" that tells it that it should produce code that loads this identifier to a temporary register and return it to the method that called it. To get the address of this variable, it checks if its a local variable or a class field and either loads from local variable or loads from the heap.

Every expression returns an object of type ObjectType which includes information about this type and the temporary register that this expression stored its result.

Thus, for consistency in code, every expression, even if its a simple one, (like INTEGER LITERAL) stores its value on a temporary register even though its not always necessary.

(NOTE: Because we have ThisExpression, and we need to follow the above approach, we allocate space on stack for a register **%.this** and load the value of the paramater **%this** to this register).

For while/if labels the approach is the same with the examples (*.ll) given. For checking out of bounds either on array allocation and/or array lookups and array assignments, we perform unsigned compare in order to also catch negative bounds error.

For types, we declare class references and boolean arrays as i8*, booleans as i1, integers as i32.
This means, that if we have a condition like ***if (x[1])*** and x is boolean array, we need to convert the value returned (i8) to i1. To do this, we use **trunc**.
When there is an assignment statement on boolean arrays, like ***x[1] = true or x[1] = y*** where y is boolean, use **zext**.

When we allocate an array, we allocate 4 more bytes than we actual size of the array in order to store its size. We store the size on the first 4 bytes, but the array pointer points after these bytes. Thus, in order to find the size of the array we use offset -1.
(NOTE: On boolean arrays, using offset -1 will not give the right result, so, we temporarily convert this array to i32, access the size, check if its valid and then access the array).

For AndExpression we use short-circuiting using the command "phi".
For example, the following code :

```
class Main {
    public static void main(String[] args) {
        if (false && ((new boolean[0 - 1])[0])) {
            /*Code*/
        } else {
            /*Code*/
        }
    }
}
```

will not throw out of bounds.

But this one will throw:

```
class Main {
    public static void main(String[] args) {
        if (true && ((new boolean[0 - 1])[0])) {
            /*Code*/
        } else {
            /*Code*/
        }
    }
}
```