# CONTRIBUTIONS:

| name | utorid | contribution |
|---|---|---|
| Nagee Elghassein | elghasse | everything |
| Timur Borkhodoev | borkhodo | everything |
| Theodor Poenaru | poenarut | None |
| Yu Sing Wong | wongyu7 | None |

| name | utorid | contribution |
|---|---|---|
| Hong Zhi Yew | yewhong | None |

:(

# Design

## A4 v A5

For every AST node, we wrote a `machine_visit` function, which generates the machine code required for it and returns it, calling `machine_visit` on children when necessary. The accumulation of all the machine code happens in the `Program::machine_visit`, which then returns all of the code necessary.

As mentioned in A4, both `Ident` and `Subs` have a `machine_lhs_visit`, which generates code for when an identifier is being written to, and for those (as with everything), `machine_visit` is simply to access.

# Policy

While generating the code, we also keep track of an offset for how far we are from the first machine instruction. Each `machine_visit` or `machine_lhs_visit` maintains the following policy

> The offset for any code is only incremented when a =new Instructior created, and a function does not increment on behalf of a function inside it.

# Testing process:

## Output

While developing features, we would create one of the new files, set it as TEST.488, and use `src/compiler488/testing/TestCodeGen.java` to run the test. We would then move `TEST.488` to one of the other new test files, and then give it a descriptive name.

## Machine code

We tested the specific code outputted by printing out the resulting `ArrayList<Instruction>`, and diffing it with prewritten machine code. These were done for very "small" test programs, like declaring one variable and storing it, or looping and exiting directly.

One thing we should have done was regressive tests. Our `TestCodeGen.java` should have taken a directory of files, ran the compiler on each of those files, and outputted the failures/passes in some nice format.

# Machine Code Generation

## MachineUtils.java

## ApplyOperator

Handles +,-,*,/,<,<=,>,>=,=,!=,and,or for arithmetic and logical operators

1. Negation

- $|TRUE - 1| = FALSE$
- $|FALSE - 1| = TRUE$

```
negation.add(new Instruction(Machine.PUSH, 1));
negation.add(new Instruction(Machine.SUB));
negation.add(new Instruction(Machine.NEG));
```

2. $a > b$

   Optimize to remove swap

```
ordered_instructions.addAll(b);
ordered_instructions.addAll(a);
ordered_instructions.add(new Instruction(Machine.LT));
```

3. $a \geq b$

   $A \geq B = \neg (A < B)$

```
ordered_instructions.addAll(a);
ordered_instructions.addAll(b);
ordered_instructions.add(new Instruction(Machine.LT));
ordered_instructions.addAll(generateNegation());
```

4. $a \leq b$

   $A \leq B = \neg (B < A)$

```
ordered_instructions.addAll(b);
ordered_instructions.addAll(a);
ordered_instructions.add(new Instruction(Machine.LT));
ordered_instructions.addAll(generateNegation());
```

5. $a \neq b$

   De Morgan's laws $A \neq B = = \neg (A = B)$

```
ordered_instructions.addAll(a);
ordered_instructions.addAll(b);
ordered_instructions.add(new Instruction(Machine.EQ));
ordered_instructions.addAll(generateNegation());
```

6. a ∧ b

   De Morgan's laws $s \neg \neg A \wedge B = = \neg ((\neg A) \vee (\neg B))$

```
ordered_instructions.addAll(a);
ordered_instructions.addAll(generateNegation());
ordered_instructions.addAll(b);
ordered_instructions.addAll(generateNegation());
ordered_instructions.add(new Instruction(Machine.OR));
ordered_instructions.addAll(generateNegation());
```

# Program

Each program generates the code for the scope inside it, and then adds the `halt` instruction.

# Scopes

Each scope visits the declarations inside it, setting the address in the symbol table for the identifier, and then procedes to generate all of the statements. For scopes in functions (minor scopes), we do some additional work for return statments and displays.

# Binary operations

# Unary operations

1. Array indexing

   We take the absolute value of the array lower bound and add it the the result of the expression within the brackets. For subs, we do the same but add the neccessary code for storing.

2. Minus

   We simply throw in `NEG` after generating the machine code for the value without the minus.

# Assignment

Assignments look like the following: `lhs := rhs`, where `rhs` is some value, and `lhs` is either an identifier or an array index. We use `machine_visit` on the `rhs` and `machine_lhs_visit` on the `lhs`. In total, we do `LOCATION; VALUE; STORE`, where `STORE` is the operation, and the first two are the result of the visits on `lhs` and `rhs`, respectivly.

# looping

Both looping statements are handled in `MachineUtils::whileDo` and `MachineUtils::repeatUntil`, and the individual machine vists on looping statements just generate machine code for the condition and the body. In machine utils, we calculate offsets and break when necessary, then put everything together depending on what type of loop it is.

# if statements

This is handled in `MachineUtils::ifThen` and `MachineUtils::ifThenElse`, where offsets and labels and handled, and the `IfStmt::machine_visit` just handled generating the code for the condition, true block, and when necessary, the false block.

# I/O operations

These call `printi` wherever `printi` works, and `printc` on strings, where the characters of the strings are written into the machine code in reverse order.

# returns

These are done by taking the return value and `MachineUtils::SwapPop`'ing it to the return address. If there is no return value, like for procedures, then we exit.

# Routines

We generate these in place, creating space for returns, and for the arguments that are passed.

# Changes: what we should have done differently

Our symbol table implementation is mostly duct tape at this point. A5 caught a lot of bugs, and the types of bugs wouldve been better fixed in a rewrite. We did not choose to rewrite because there was only two of us...

Make each instruction its own type that understands how many arguments it takes. We didn't really run into any issue with our `Instruction` class, but having a type garuntee that the correct number of args are being passed in is nice.

We implemted labels by counting out the number of lines generated in the first visit, we should have created place holders and visited a second time to fill in the correct line numbers.

Fixing our communication problems early in the course... :/

Oh well, we had fun :)