CSF363 - Compiler Construction: Assignment 4

In this assignment, you will be tasked to generate code for the additions you have made till now as well as make edits throughout the entire pipeline.

Task 1: Types

Add support for types. Currently we only support one type, which internally is a 32-bit integer. You will have to now add `short`(16-bit), `int`(32-bit), and `long`(64-bit)datatypes. Edit the lexer, parser so that variable declaration is changed to this:

```
let a : int = 25;
```

You will have to make changes to the AST to accommodate this new type information. You will also have to account for type coercion. Smaller types should be able to fit into bigger types, and a bigger type should not be able to fit into smaller types. Print appropriate error messages. Arithmetic operations should also handle type coercion correctly, result of an arithmetic operation between two different types should always be the larger type. The `dbg` statement should work with all three types.

Task 2: Control Flow

Add support for if statements. If statements should follow this syntax:

```
if a {
  // do something
} else {
  // do something else
}
```

Note that every if statement has an else statement attached to it. Braces are also mandatory. Remember to handle scoping issues when implementing the block of the if statement. The compiler must raise appropriate errors when encountering a variable that is out of scope:

```
if a {
  let b : int = 4;
} else {
  let c : int = b;
  // This should throw an error, if b is not defined in the outer scope
}
```

The compiler must not throw an error if a variable is redeclared in an inner scope:

```
let b : int = 5;
if a {
  let b : int = 6;
  // this is valid
} else { //...
}
```

For the `if` condition, 0 is considered as false and any non-zero number is considered as true.

Task 3: Functions

Add function support to the language. Functions should follow this syntax:

```
fun add(a : int, b : long) : long {
  ret a + b;
}
```

If there are no return statements, a function by default returns 0.

Perform type-checking when a function is called, i.e. the type of its formal parameters should equal to or bigger than the type of its actual arguments (same coercion rules as stated above). The return type of a function should also follow the same rules as mentioned in Task 1.

```
let a: int = 10;
let b: short = 20;
let c: long = 30;

let d: long = add(a, b); // b is coerced into a long type

// This should be an error:
let e: long = add(c, c);
// c is long, it cannot be coerced into an int
// This is also an error:
let f: short = add(a, c);
// add returns a long. A long cannot be coerced into a short
```

Currently in the baseline compiler, as they are no functions, all code is wrapped up automatically into a `main` function. Edit the code generation so that there should be an explicit `main` function from now on. The compiler throws an error if the `main` function is not present. The `main` function should not take any arguments, and returns an `int`.

Task 4: Optimizations

These optimizations should run just after parsing and before code generation, i.e. optimizations are performed **on the AST**. After your compiler is done with the optimizations, it should write the optimized AST into a text file called `opt.txt` in the bin directory of the project (you may use the provided `to_string` method of the AST class to get the `string` form of the tree).

Add the following two optimizations to the compiler.

Constant Folding

Operations that involve only literals, should be folded into one constant. The following code:

```
let b: int = 3 * 4 + 5 - 6;
```

has the following AST:

```
(let (b int) (- (+ (* 3 4)) 6))
```

After doing constant folding you should get:

```
(let (b int) 11)
```

Branch Elimination

Remove branches of the if statement that are never reached. For example, here the if condition is always zero, and the if branch is never executed.

```
if 0 {
  dbg a;
} else {
  dbg b;
}
```

has the AST:

```
(if-else 0
  (dbg a)
  (dbg b)
)
```

After doing branch elimination, the AST should look like:

```
(dbg b)
```

At the end of these tasks, the compiler should run code that looks like this:

```
fun factorial(n : long) : long {
  if n {
    ret factorial(n - 1) * n;
  } else {
    ret 1;
  }
}

fun main(): int {
  let n : int = 20;
  let res : long = factorial(n);
  dbg res;
}
```