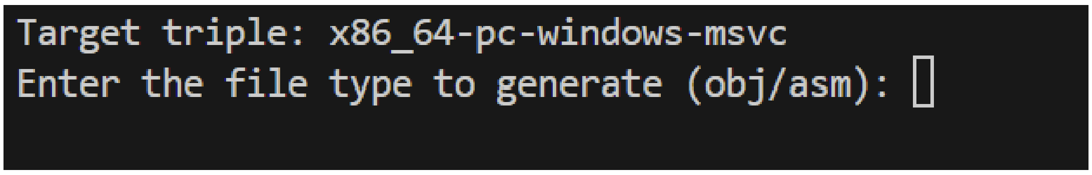
Results and Future Work

a) Test Results

The test results from the test cases pass successfully. The compiler is only tested for integers, boolean, floats and strings. Other types of data may result in lexer error. The compiler is also able to output IR code and assembly code based on the input. However, the user needs to give command to the compiler whether to output assembly or object files.



```
Target triple: x86_64-pc-windows-msvc
Enter the file type to generate (obj/asm):
```

asm or obj should be inserted in this prompt to continue into the code generator.

b) Performance Analysis

- Compilation time

  The Ngebski compiler has seven phases of compilation, including lexer, parser, abstract syntax tree generator, code generator for intermediate code, IR optimization for .obj and .s and machine code generator. Ngebski compiler favours runtime performance over compilation time to ensure performance for the targeted code. Therefore, the runtime complexity is linear O(1).

- Memory usage

  The memory for Ngebski compiler depends on the nodes in AST and does not depend on the input. Therefore the space complexity of this compiler is O(1) which is quite efficient.

- Generated code quality

  There are three types of codes generated by Ngebski compiler:

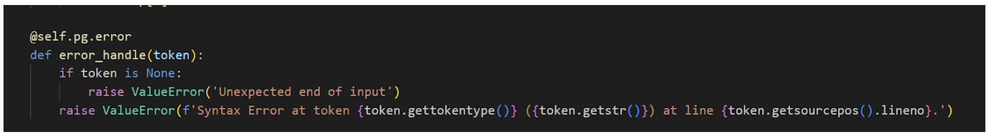
  - Assembly code
  - Intermediate representation code
  - Human readable output

  Generally, the code quality follows the syntax of IR and assembly. For human readable output, it follows the high-level language style output.

- Error handling

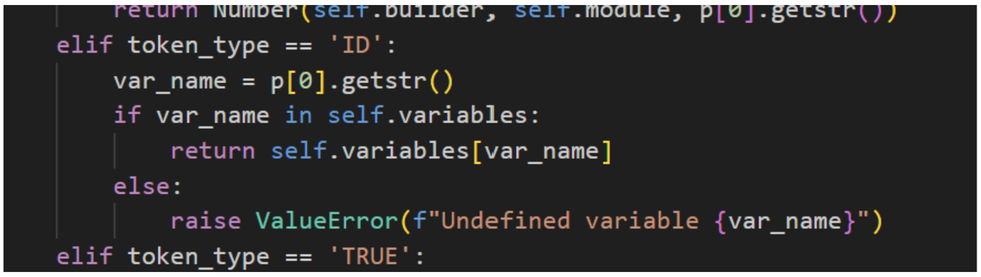
  There are several error handling inserted in the compiler, defined as error messages to tell the user which error results in compilation failure.

  - Syntax error



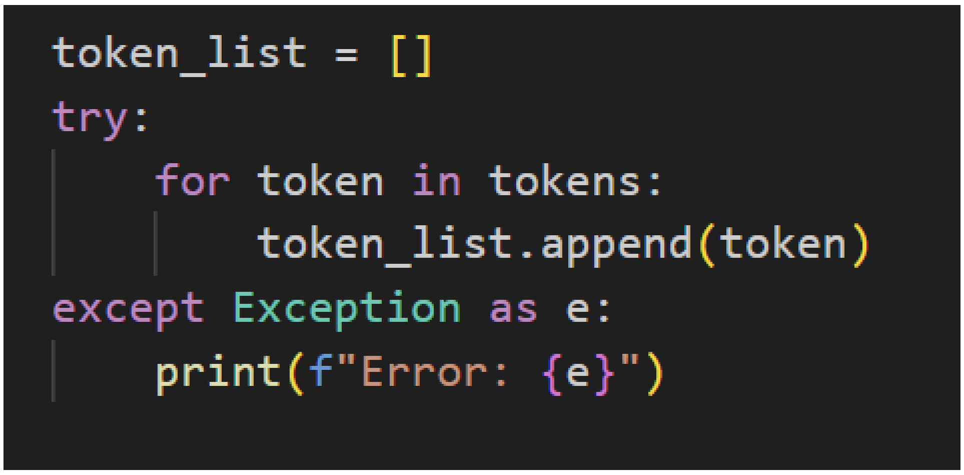
```python
@self.pg.error
def error_handle(token):
    if token is None:
        raise ValueError('Unexpected end of input')
    raise ValueError(f'Syntax Error at token {token.gettokentype()} ({token.getstr()}) at line {token.getsourcepos().lineno}.')
```

- Undefined variable error



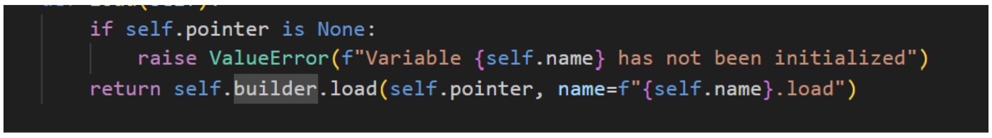
```python
        return Number(self.builder, self.module, p[0].getstr())
    elif token_type == 'ID':
        var_name = p[0].getstr()
        if var_name in self.variables:
            return self.variables[var_name]
        else:
            raise ValueError(f"Undefined variable {var_name}")
    elif token_type == 'TRUE':
```

- Token error/undefined tokens



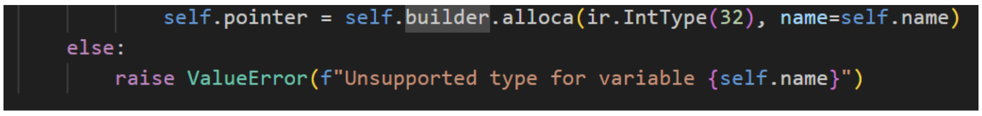
```python
token_list = []
try:
    for token in tokens:
        token_list.append(token)
except Exception as e:
    print(f"Error: {e}")
```

- Variable initialization error



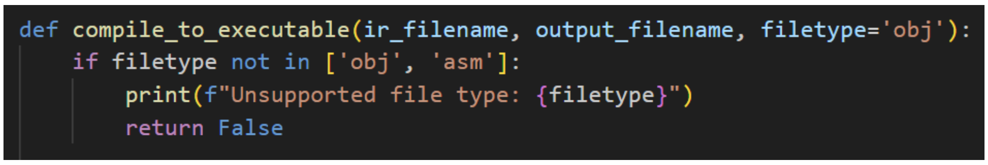
```python
    if self.pointer is None:
        raise ValueError(f"Variable {self.name} has not been initialized")
    return self.builder.load(self.pointer, name=f"{self.name}.load")
```

- Unsupported variable error



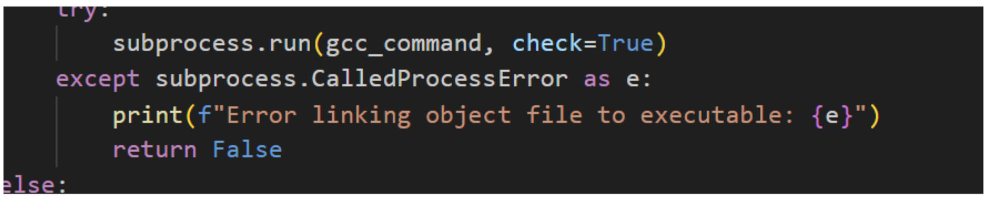
```python
            self.pointer = self.builder.alloca(ir.IntType(32), name=self.name)
    else:
        raise ValueError(f"Unsupported type for variable {self.name}")
```

- File type error



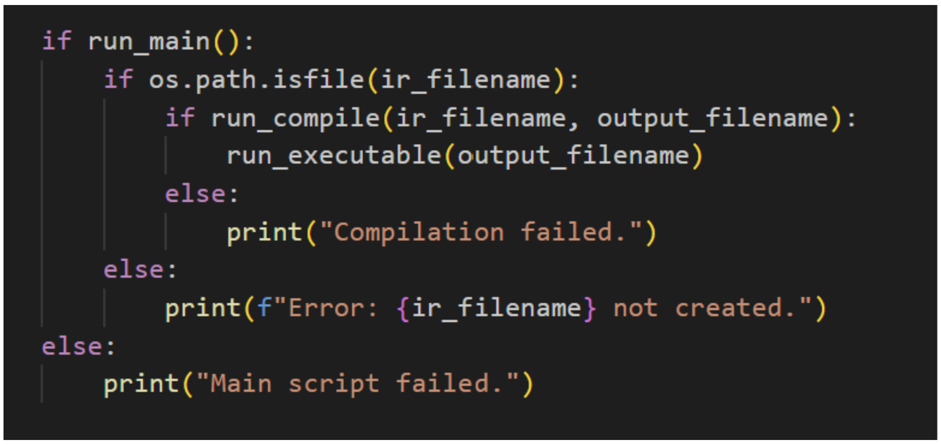
```python
def compile_to_executable(ir_filename, output_filename, filetype='obj'):
    if filetype not in ['obj', 'asm']:
        print(f"Unsupported file type: {filetype}")
        return False
```

- Linking error



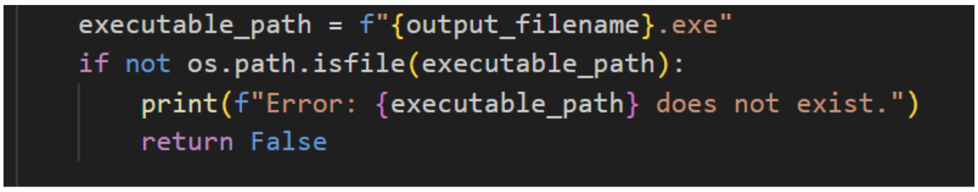
```
    try:
        subprocess.run(gcc_command, check=True)
    except subprocess.CalledProcessError as e:
        print(f"Error linking object file to executable: {e}")
        return False
else:
```

- Compilation error



```
if run_main():
    if os.path.isfile(ir_filename):
        if run_compile(ir_filename, output_filename):
            run_executable(output_filename)
        else:
            print("Compilation failed.")
    else:
        print(f"Error: {ir_filename} not created.")
else:
    print("Main script failed.")
```

- File location error



```
    executable_path = f"{output_filename}.exe"
    if not os.path.isfile(executable_path):
        print(f"Error: {executable_path} does not exist.")
        return False
```

c) Future Enhancements

There are some features that can be added to improve the Ngebski language, including

- Function calls
- For loop
- Integration with other operating systems aside from Windows and MacOS
- Type system enhancement
- Complex mathematical functions

This concludes the Ngebski programming language compiler project. By applying compiler principles from lexer analysis, LL tables, syntax definer and BNF, the Ngebski compiler is finished successfully. We applied rigorous testing to ensure there are no ambiguity and errors in the compiler. Additionally, We designed the compiler to be available in both Windows and MacOS to ensure its flexibility.