# CS4201 P02 Report

**matriculation number: 160021429**

# 1. Overview

The aim of this practical is to implement the backend of the compiler for the Oreo language. There are 4 parts in this practical - 1) semantic analysis (variable scoping), 2) type system for the Oreo language, 3) generating the Three Address Code, and 4) generating High Level Assembly file.

## 1 - 1. Instruction

For this practical, I used the C++ language to implement the compiler backend. So, you need to use “g++” compiler to compile the source codes.

To compile : “g++ *.cpp -o output”
To execute : “./output”

# 2. Design and Implementation

Since parsing the Oreo source code is unnecessary part for this practical, I used the hand-coded trees. You could see the functions that generate hardcoded trees in the “oreoAST.cpp” file.

Basically, I modified the example codes that are uploaded on the studres to implement the syntax tree. When I tried to start generating example trees with the given structs and unions, I realised that it is impossible to check which type of the union is used. This is because that the members of the union shares the memory. Thus, I added new structs for each union, where each new struct has an attribute called “type”.

For example, I made a new struct Stmt for the union Statement. The Stmt has 2 attributes - “Statement *statement” and “char type”. The attribute “type” could have 6 possible values, since the union Statement has 6 members in it. For instance, the value of type is ‘a’ if the

union Statment uses the “Assign” type, and the value of type is ‘p’ if the “Print” type is used. Similarly, ‘v’ for “Variable”, ‘g’ for “Get”, ‘w’ for “While”, and ‘i’ for “If”.

So, by checking the value of type, the program could know which type of the union is currently used.

Furthermore, I added the integer variables “startLine” and “endLine” to the struct Compound. These will be used by the semantic analyser to check the scoping of the variables.

## 2 - 1. Semantic Analyser (Part 1)

The main aim of the semantic analyser is to check the scoping of variables. For example, the procedure cannot use the variable, which is declared in the other procedure. Furthermore, we should not assign value or get value of the variable that did not declared yet.

To implement this, I used the symbol table, which contains information about each variable. The class SymbolTable has an attribute called “list”, and the list contains the VarInfo instances. And each VarInfo contains the information about the variables in the source code. The VarInfo class has attributes called “name”, “val”, “type”, “declaredLine”, and “defined”.

So, by comparing the name of the variable, we could find the corresponding VarInfo instace from the SymbolTable. Then, we could check the scope of the variable by comparing the attribute “declaredLine”, which is the line number that the given variable is declared, with the starting line and end line of the current compound. By doing this, we could easily check the scoping issue for the semantic analyser.

## 2 - 2. Type Checker (Part 2)

The main aim of the part 2 is implement the type system for the Oreo, so that the compiler could correctly checks the type of expressions. According to the specification, the Oreo language has 3 types - integer, boolean, and string. We could only use the conditional operators (>=, ==, <=, etc) and arithmetic operators (+, -, *, /) with integers, and the operands of the logical operators (&&, ||, not) should be boolean type.

Basically, I made functions for each struct, and call the corresponding function recursively, so that the type checker could check the type of each expression easily. In the “TypeChecker.cpp” file, there are 2 types of functions - getType() and checkType().

The getType functions check the type of the given expression, and return the corresponding character. It will return ‘i’ for integer, ‘b’ for boolean, ‘s’ for string, 0 for undefined, and -1 for error. The reason that I added “undefined” type is because I realise that there is a possibility that the variable is not initialised. It is clear that using the uninitialised variable will generate

either error or undefined behaviour. Thus, I added the concept of “undefined” to my type system, so that the program could generate error message for using uninitialised variables. Perhaps, we could catch errors about using unintialised variables with parser in the frontend of the compiler, however, I just wanted to try to make my program to handle all possible errors that I could think.

If the given expression is a constant value, then it will first check if all characters of the constant are digits. If so, then the getType() function will return ‘i’. Otherwise, it will check if the given constant is either True or False. If so, ‘b’ will be returned. Otherwise, the getType() will return ‘s’. Similarly, if the given expression is a variable, the type checker will use the symbol table to get the type of the variable.

If the given expression contains the operator, then the getType() function will first check which operator is used. Then it will call the getType() method recursively to get the types of operands. Next, it will compare the types of 2 operands. If the types of 2 operands are same, then it will check if the correct types of operands are used for the operator. For example, if the expression contains the arithmetic operator, then the types of 2 operands should be integer. Similarly, if the expression contains the logical operator, then the types of 2 operands should be boolean. If the types of the operands are not suitable for the given operator, then the getType() function will return -1 to let the type system know that there is a type error in the syntax tree. Otherwise, it will return the corresponding character value (‘b’ for boolean, ‘i’ for integer, and ‘s’ for string).

And the checkType() functions check the type of the given expression. Basically, the checkType() functions return true if there is no type error, and return false if the program found the type error in the syntax tree. First, it checks if expression contains the operator. If so, it will call the getType() function for each operand. Then, it will check which operator is used in the given expression. If the operator is either an arithmetic operator or a conditional operator, then types of both operand expressions should be integer. Or if the operator is a logical operator, then the types of both operand should be boolean. By checking the types of the operand expressions, we could check if there is a type error in the given expression.

If there is a type error in the syntax tree, then the program will print out the error message. The error message that I implemented in this practical is not as good as the one that I implemented in the previous practical - this time, the error message does not show which line contains the error. However, still it could let the user know that there is a type error in the source code. Basically, I thought that implementing some good error message is not an essential part of this practical, thus, I just made my program to print out some straightforward and simple error message.

## 2 - 3. Three-Address-Code Generator (Part 3)

To implement the TAC(Three Address Code) generator, I wrote 2 classes - TAC and TACList. The TACList is a class that contains the list of TAC instances, and the TAC is the

class for each three address code. The TAC class has attributes called a1, a2 and a3, where each of them contains the string for three address code. Also, the TAC class has an attribute “op”, which will be used for the operator. Moreover, there are 3 boolean attributes in the TAC class - hasAssign, hasGoto, and isStartOfSection. The “hasAssign” lets the program know that the given TAC instance has an assignment, and the “hasGoto” lets the program know that the given TAC has the “goto” statement. And “isStartOfSection” attribute is used when the given TAC instance contains the label for the branching (i.e. L0 : ).

For the three address code, I made 3 cpp files - “TAC.cpp”, “TACGenerator.cpp” and “TACWriter.cpp”. The “TAC.cpp” contains the member methods of the TAC class and TACList class. The “TACGenerator.cpp” contains the functions that parses the AST and generate TAC instances. And the “TACWriter.cpp” contains the functions that write the TAC file.

The generateTACList() function in the “TACGenerator.cpp” file will generate the TACList instance for each procedure. Then, it will call the generateTAC() function to parse the statements in the procedure, and generate corresponding three address codes. It will check the statements and expressions, and call the corresponding function. For the syntax of the three address code, I referred to the TACExample file on the strudres.

To convert the nested expression to the three address code, I TAC generator will add the temporal variables, whose name is starting with “_TEMP_”. For example, if there is a example “a = (b + c) + (d + e)”, the TAC generator will generate “_TEMP_1 = b + c”, “_TEMP_2 = d + e”, and “a = _TEMP_1 + _TEMP_2”. To implement this, I made a function called “generateVarName”. This function uses the global variable varNum to count the number of temporal variables that it made, and generate an unique name for the three address code.

Similarly, I made a function “generateSectionNum”, which generates the unique label name for the branching. This function also uses the global variable “sectionNum” to count the number of labels that it generated, and generate an unique label name. For example, if we need 3 labels, then this function will generate “L0”, “L1” and “L2” for the branching. By using this function, the TAC generator could easily write the three address codes for conditional statements and loops, which use the conditional branching. For example, we could generate “If b Goto L0” and “Goto L1” for the if statement “if (b)”. Similarly, we could implement the loops with labels and Goto statement. Clearly, this will make the program to convert TAC to HLA easily.

After the functions in the “TACGenerator.cpp” file generates the TAC instances, then the program will use the functions in the “TACWriter.cpp” file to write the three address code file. Basically, the TAC writer uses the ofstream instance to use the file output stream to write the file. The functions in the “TACWriter.cpp” file simply checks the values of the attributes of the TAC instances, and write the corresponding TAC codes via file output stream.

While testing the TAC generator, I found that my TAC generator generates some unnecessary codes. For example, if the original source code is “a = b + b”, then my TAC

generator generates “_TEMP_1 = b; _TEMP_2 = b; a = _TEMP_1 + _TEMP_2”, which is definitely inefficient. Perhaps, this is because my program do not have the optimizer that optimizes the Intermediate Representation. I was not able to find any other way to improve my program to overcome this issue, and I think it is not the essential part of this practical. However, it is good experience that let me know that why we need the optimizer that optimizes the Intermediate Representations.

## 2 - 4. High-Level Assembly Generator (Extension)

The last component of my program is the HLA generator, which parses the TAC instances, and generate HLA codes. Basically, I referred to the HLA Reference to implement the HLA writer that generates the HLA codes with the correct syntax.

As you know, the HLA is an assembly language that supports some high level instructions such as loop or conditional statements. However, according to the specification, we are expected to implement the loops and conditional statements without using the high level instructions. This means that we need to implement them with branching method. This is the main reason that I implemented the generateSectionNum() function in the TAC generator. As my TAC instance uses the goto statement and label for the branching, it was easy to write the HLA codes with branching.

While implementing the HLA writer, I found that the way how the HLA is handling the conditional operators is quite different with the TAC. In TAC, I just make a new temporal variable to store the result of the conditional operation (i.e. _TEMP_1 = a < b). However, in the assembly and HLA, we only could use cmp and corresponding jump instruction to implement the conditional operation. For example, my TAC generator will convert “a <= b” to “_TEMP_1 = a <= b” to store the result of the conditional operator, however, the most easiest way to convert that statement to HLA is “cmp a b; jl L0; jmp L1”. This is because the cmp and jump instructions uses some predefined register to check the result of conditional statement. And I was not able to find which register does the cmp instruction uses to store the result. To overcome this issue, I found 2 solutions - 1) using branching to store the corresponding boolean value and 2) using the conditional instructions.

Again, we could use branching method to overcome this problem. So, by using the branching, we could jump to different label, and assign suitable boolean value. Clearly, this is the low level way to handle the conditional operations. However,in the HLA, there are instructions called conditional instructions, such as setle, setge, setl, etc. What these instructions do is they sets the result of the conditional operation to the given memory or register. For example, we could convert “a <= b” to “cmp a b; setle(“eax)” to store the result of the statement “a <= b” to the eax. Obviously, using conditional instruction is much easier than using branching method. However, I did not know about the conditional instructions until 5 hours before the deadline. Since I did not have enough time to change the codes, I decided not to rewrite the codes for conditional instructions. You could see this in the generated HLA file.

When I first run the HLA program, the program did not work as I intended. And I found that it is because I did not add the “@noframe” annotation to the procedure. According to the HLA Reference, we need to add @noframe next to the procedure declaration part. Otherwise, the HLA will automatically insert some additional codes, which might make the program to not work as intended.

Furthermore, I improved my HLA writer to generate the “var” section for procedures. The “var” section uses to declare the local variables. According to the chapter 10 of the HLA Reference, the HLA automatically references the variables in the var section using the negative offset from the EBP. This means that the variables in the var section will automatically inserted into the stack frame.

For the parameters of the procedures, I found that the HLA allows to add a list of parameters just like other high level programming languages. If you see the chapter 11 of the HLA Reference, you could find that the HLA supports the procedure arguments. Thus, I made a function called “writeParamsOfProcedures”, which checks if the given procedure has parameters, and adds a list of parameters next to the procedure name.

Apparently, handling the return statement with HLA was the hardest part. The HLA has a function called “ret()”, which supports the programmers to implement the return statement. However, for some reason, I was not able to get it to work. Thus, I just move the value to the “eax”, and exit the procedure by using “exit procedure_name” instruction. To make this work, I also used the “@return(“eax”) annotation, which lets the HLA know that this procedure returns the value that is stored in the “eax” register. To be honest, I still have no idea why the “ret()” function did not work. So, I am planning to have more research after submitting this practical to find the way to get “ret()” function to work.

# 3. Testing

To test the program, I made 3 example trees in the “oreoAST.cpp” file. There are 3 functions - getExampleTree, getExampleTree_error1, and getExampleTree_error2. The function getExampleTree() returns the AST that does not contain the error. So, the program should be able to generate both TAC file and HLA file properly. And the getExampleTree_error1() function returns the tree that has the variable scoping issue, thus, the semantic analyser should be able to produce an error message. Similarly, the getExampleTree_error2() function has a type error. Thus, the type checker should print out the suitable error message.

You could just run the program without any command line arguments. However, if you want to test the program with “getExampleTree_error1” function that returns the tree with variable scoping error, you need to use “1” as a first command line argument. Similarly, if you want to test the program with “getExampleTree_error2” function that returns the tree that has the type error, you need to put “2” as a first command line argument.

i.e.
“./output”

```
Program ends
```

“./output 1”

```
ScopeError::The scope of the variable n is invalid!
Error::Semantic analysis failed
```

“./output 2”

```
Error::Type checking failed - Type error is found in the function "main"
```

As you could see above, the program could print out suitable output. Furthermore, the program could generate suitable TAC file and HLA file.

# 4. Conclusion

By doing this practical, I was able to learn how the compiler backend compiles the Intermediate Representations such as AST to the executable files. I feel like the previous practical was related to the “design” of the programming language, and this practical is actually related to the “implementation” of the language. Last time I spent most of my time to analyse the Oreo codes lexically, and make parser to handle the grammar of the language. However, while doing this practical, I spent more time to think how the given Oreo code should run on the actual machine.

Definitely, I enjoyed this practical, and I could say that I learned a lot about the implementation of the programming language.

# 5. References

HLA Reference :
<http://www.plantation-productions.com/Webster/HighLevelAsm/HLADoc/HLARef/TOC_html/HLAReferenceTOC.htm?fbclid=IwAR1dE05HojnVFKRl7gB7wjkAq9mjpAYjhK7qiIkxzg7l9qPuA12WPKbwGRE>

TAC example:
<https://studres.cs.st-andrews.ac.uk/CS4201/Examples/TAC-Examples.pdf>