भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

CS3423 - COMPILERS-2

PROJECT REPORT

TEAM MEMBER NAME	ROLL NUMBER
Sai Mahesh Abburi	CS18BTECH11001
Bandi Chiranjeevi Sai Ganesh	CS18BTECH11004
Bhargav Siva Phaneendra	CS18BTECH11005
Krishna Pawan Darapaneni	CS18BTECH11008
Deepak Reddy Jillela	CS18BTECH11016
Vijay Kumar Reddy	CS18BTECH11017
Sai Krishna Kuchi	CS18BTECH11025

Introduction:

A compiler is a program that takes the source code written in a particular language as input and returns a machine-level code which can be understandable by the computer. After writing a program in a particular language, the programmer runs a compiler corresponding to that language by specifying the name of the source code file. There are many stages in compiling. The compiler can be broadly divided into two phases. They are:

1) **Analysis phase:** In this phase, the compiler reads source code and checks for lexical, grammar, syntax errors. This phase outputs an intermediate code. Lexical analysis, syntax analysis(parser), semantic analysis and intermediate code generation are done in this phase.
2) **Synthesis phase:** In the phase, the compiler generates the target machine code with the help of intermediate code. Code optimization and code generation are done in this phase.

At first, Lexer takes source code as input and generates the tokens and removes white spaces and comments simultaneously. These tokens are further processed by parser and parser generates a parse tree, and it also checks some of the syntax rules of programming language. Parse tree generated by the parser is analyzed by the semantic analyzer. Semantic analyser checks whether the parse tree follows the rule of language or not. After semantic analysis, the compiler generates intermediate code for the target machine. This intermediate code is further optimised, and the code

generator takes optimised intermediate code and generates target machine-level code.

Language Tutorial:

Let's start with a simple C+ program "hello_world.cp". The extension for C+ program file is .cp

```
+ "io.h"

int main() {

   print("Hello World\n")

   return 0;

}
```

Let us look at this code step by step:

io.h (stands for Input Output header). **+** says our compiler to compile our program with the **"io.h"** header file included. Header files contain the declarations for specific functions and save us the cost of rewriting huge code again. io.h header here contains the definitions for input and output functions (print() in this example code).

int main() is the main function where execution of the program starts and is necessary for successful compilation of our program. All the code belonging to any function is to be included in **curly braces '{ }'**.

print("Hello World\n") says to print "Hello world" with a newline on the console. All functions are made of two parts - function name and parameter

list. For example, in this case, print is the function name and “Hello World\n” is a single parameter of the parameter list.

return 0 indicates termination of function with return value 0. Normally, even without any return statement, the program would end at the end of the main function. However, conventionally as in C, we may use return 0 to indicate successful termination of our program.

Language Reference Manual:

1. Introduction

The C+ programming language is a high level and procedural type of programming language which supports Object Oriented programming, Lexical scoping, Function Overloading and Nested Comments. When considered either C or C++, some features are exclusive to only one language. So in order to cover these exclusive features and also to support additional functionalities we came up with our own language C+ which is a union of C and C++. This C+ can be considered as a prototype as it is not much help to work with hardcore programs. The main purpose of designing this language is to cover up the drawbacks of C and C++ by covering the features of both these languages in a single language and thereby producing a new language C+.

2. Operators

C+ language supports a wide variety of operations. Most of the operators are similar to C. The different types of operators that are allowed by C+ are given here.

2.1. Arithmetic Operators

All the arithmetic operators that are supported by C such as addition(+), subtraction(-), multiplication(*), division(/), remainder(%) are also supported by C+.

2.2. Logical Operators

C+ supports the logical operators such as logical OR(||), logical AND(&&) and logical NOT(!). In addition to these, the keyword 'and' can also be used for logical AND, similarly 'or' for logical OR and 'not' for logical NOT.

2.3. Bitwise Operators

All the Bitwise Operators - Bitwise OR (|), Bitwise AND(&), Bitwise XOR(^), Bitwise NOT or one's complement(~), Bitwise Left Shift(<<), Bitwise Right Shift(>>) are supported by C+.

2.4. Relational Operators

All the Relational Operators - Less than(<), Greater than(>), Less than or equal to(<=), greater than or equal to(>=), equality(==), Not equality(! =) are supported by C+.

2.5. Assignment Operators

A wide variety of assignment operators like =, + =, − =, * =, / =, % =, |=, & =,^=, <<=, >>= are supported by C+.

2.6. Predefined Constants

As some of the universal constants are used many times, we included them as global constants and can be used anytime.

There are two predefined constants we introduced in this language

1. _pi = 3.14
2. _e = 2.78

3. Data Types

The Data Types included in C+ are int, char, float, bool.

3.1. Int

A variable of type 'int' occupies 4 bytes of memory which in turn requires 32bits. The highest bit(32nd bit) is used to store the sign of the integer. This bit is 1 if the number is negative, and 0 if the number is positive. So the range that can be stored under int-data type ranges from −2^31 to 2^31 − 1.

3.2. Char

A variable of type 'char' is assigned according to ASCII rules and since ASCII tabulates a total of 128 characters, C+ also supports 128 characters which can be stored using a variable of data type 'char'. A variable of 'char' data type is assigned 1byte of memory.

3.3. Float

A 'float' data type can be used to store any decimal valued constants. It occupies 4 byte of memory and follows IEEE standard representation. It ranges from −1.2e^38 to + 3.4e^38 .

3.4. Bool

C+ supports variable of data type 'bool'. There are 2 boolean constants - 'true' or 'false'.

4. Comments

C+ supports two types of comments.

4.1. Single Line Comments

A single-line comment begins with two backslashes(//) and ends with a newline character and comments are ignored by the compiler.

4.2. Multi line Comments 0r Block Comments

A comment extending over multiple lines can be inserted in between /* and */. Unlike C, C+ also supports Nested comments.

Example:

```
/*
* This is a comment
* Containing Multiple lines
* /* This is a Nested Comment*/
*/-->
int main(){
      int a = 5; // This is a single Line comment
}
```

5. Decision Control Statements

5.1. if Statement

For a single statement, it can be simply written as

```
if(expression)
      statement;
```

When the expression inside the if condition evaluates to the Boolean true value the statement present below the if condition will be evaluated. For multiple statements, the block of code to be included under the if condition is wrapped with a pair of braces.

5.2. if-else statement

An else statement can be added to the if statement and will be evaluated when the expression inside the if condition evaluates to Boolean false value. We can also use else if clauses and Nested if-else(an entire if-else construct within either the body of the if statement or the body of an else statement) for multiple evaluation as required.

Example:

```
if (condition1)              // Simple if
   statement1;
if (condition2){             // if-else
   if (condition3)          // nested if
       statement2;
   else if(condition4)      // else if
       statement3;
   else
       statement4;
}
else{
   statement5;
}
```

6. Jump statement

These statements are used to jump from one line of execution to another line of execution.

6.1. break statement

'***break***' keyword is used to exit from the execution of the code below it which is wrapped under a loop and the line of execution becomes the immediate line after the loop constituting this keyword in the code.

6.2. continue statement

'***continue***' keyword can be used to skip the code execution below it which is wrapped under a loop and line execution becomes the first line in the loop constituting this keyword.

6.3. return statement

'***return***' keyword can be used to exit from a function and while doing so can also return a constant (can be int or char or float or bool depending on the return type of the function).

7. Loop Statements

Loops are used to execute the same sequence of statements for desired no.of times.

7.1. for Loop

A structure which is used to execute a sequence of statements multiple times. We provide two types of formats:

1. *for (initialisation ; condition ; increment){ }*

This is much similar to C ’for’ loop but here we provide an optional initialisation, conditional, increment statements.

```
for(initialise counter ; test counter ; increment
counter){
   statement1;
   statement2;
   statement3;
}
```

2. *for(int i : arr){ }*

   This structure is known as ranged for loop which is used when we are iterating over a range or over an array elements and it can be used to avoid array out of bounds error. This is similar to that of ranged for loop in python.

   Example:

```
int main(){
   int arr[10] = {0};
   for(int i : arr){
       arr[i] = i+1;
   }
}
```

7.2. while Loop

Repeats a sequence of statements until the decision making condition becomes false.

```
while (condition) {
   do this;
   increment if required;
}
```

8. Array

Array is a data structure which contains a group of elements of the same data type (int or char or float or bool) to be represented as a single variable. The indexing of the array begins with 0. The data is allocated in contiguous memory locations.

8.1. Initialisation

Initialisation of an array can be done in two ways:-

1. *To initialize all the elements of the array to a single value*: All the array elements can be initialised to a single value using the following initialisation code.

```
int arr[10] = {10};
```

2. *To initialize the different elements of the array with different values:*

   Each element in the array can be initialised with a specific value by following the structure as described below.

```
int arr[10] = {1,2,3,4,5,6,7,8,9,10};
```

   The above code initialises arr[0] = 1, arr[1] = 2, arr[2] = 3 .... soon.

8.2. String

C+ doesn't support any exclusive datatype to store string constants. But enables strings to be stored as an array of characters but the number of characters in the string should be predefined. A string can be initialized by the following ways:-

```
char arr1[11] = {'H','e','l','l','o',' ','w','o','r','l','d'};
char arr2[11] = "Hello World";
```

8.3. Multidimensional Array

C+ supports multidimensional array. This feature enables the users to store multiple strings. A multidimensional array can be initialised by the following ways:-

```
int arr1[3][3] = {{1,2,3},{4,5,6},{7,8,9}};
int arr2[3][3] = {{0}};
```

9. Functions

The declaration of function is similar to C and the arguments of the function are separated by comma and the return type should be declared before the function name.

9.1. Lexical Scoping

Unlike C and C++, C+ support Lexical Scoping or Nested Functions.

Example:

```
int func1(int a, int b) {
   int func2(int a,int b) {
       return a*b;
   }
   return func2(a,b);
}
```

9.2. Function overloading

Unlike C, C+ supports Function Overloading.

Example:

```c
void func(char s[20]) {
   printf("Function with string %s",s);
}
void func(int a) {
   printf("Function with integer %d",a);
}
void func(int a, char b) {
   printf("Function with integer character %d, %c",a,b);
}
int main() {
   func("cs18btech");
   func(25);
   func(16,'c');
   return 0;
}
```

10. Objective Oriented Programming

C language doesn't support Object-oriented programming but C++ does, So we would like to introduce it in this language. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function. In C+ we like to support Classes & Objects and facilitate Encapsulation, Polymorphism and also Inheritance.

10.1. Classes and Objects

Class is a user-defined Data type, which holds its own data member functions. They can be accessed by creating an instance of that class. An Object is an instance of the class. When a class is defined no memory is allocated, but when it is instantiated memory is allocated.

Example:

```
class student{                              //student is class
   char name[50];                          //Data member
   char rollno[15];
   int marks;
   public:
       void get_details(){                 //Member function
           printf("Name is %s", name);
           printf("Roll No is %s", rollno);
           printf("Marks in the Exam: %d", marks);
       }
};
int main() {
   student A;                    //A is an object
   student B;                    //B is an object
}
```

10.2. Encapsulation

Generally, encapsulation is defined as wrapping up of data information under a single unit, here it is defined as binding together the data and the functions that manipulate them. By this, we can hide some data from certain parts of code and make it available for some parts of code.

Example:

```
class example {
     //Private variable only accessible for class members
private:
      char pvt[20];
     //Public functions, variables accessible for all members
of code.
public:
      int pbl;
```

```
        void write(char s[20]) {
            pvt = s;
        }
        void read() {
            printf("Private variable is %s", pvt);
            printf("Public variable is %d", public);
        }
};
int main() {
      example ex1;            //ex1 is an Object
      ex1.public = 25;
   // ex1.pvt = "string";    //This is inaccessible from main()
   // write can access the private variable since it is member
of class
      ex1.write("string");
      ex1.read();         //Prints the public & private variable
   return 0;
}
```

10.3. Polymorphism

Generally, polymorphism means having many forms. C+ supports compile time polymorphism and which can be achieved by function overloading.

Function Overloading:

If there are multiple functions with different parameters and the same name.

Example:

```
class poly {
public:
      void func (char s[20]){                //Function with
string as argument
            printf("Function with string %s", s);
```

```
        }
        void func (int a){                    //Function with
integer as argument
            printf("Function with integer %d", a);
        }
        void func (int a, char b){            //Function with
integer & character as argument
            printf ("Function with integer & character %d, %c",
a, b);
        }
};
int main() {
   poly obj;
   obj.func("cs18btech");           //1st func is called
   obj.func(25);                    //2nd func is called
   obj.func(16, 'c');               //3rd func is called
   return 0;
}
```

10.4. Inheritance

The capability of a class to inherit properties and characteristics from another class is called Inheritance.

Sub Class: The class that inherits properties from another class is called Subclass or Derived class.

Super Class: The class whose properties are inherited by a subclass is called a base class or Super Class.

Private members of any class cannot be inherited by any other class, but public, protected members can be inherited. C+ supports only single inheritance i.e a class is allowed to inherit from only one class.

Example:

```
class A {
public:
   int x;
   void A(){
       printf("This is class A");
   }
protected:
   int y;
private:
   int z;
};
class B inherits public A {
   // x is public
   // y is protected
   // z is not accessible from B
};
class C inherits protected A{
   // x is protected
   // y is protected
   // z is not accessible from C
};
class D inherits private A {
   // x is private
   // y is private
   // z is not accessible from D
};
int main(){
   B obj;
   return 0;
}
```

11. ECOOP Compiler Design

ECOOP(Exceptional C with object-oriented programming) compiler has been designed for the C+ language and the design is explained here.

11.1. Lexer & Parser

Lexer and parser are written combined in ANTLR grammar.

11.2. Error Handling

11.2.1. Array out of bound error

Trying to access the elements of an array which are beyond the size of the array would result in an array out of bound error.

```
#include <stdio.h>
void main()
{
      int array[5]={0,1,2,3,4};
      array[5]=array[5]+array[2];
}
```

The above program would result in an Array out of bounds error because the index of array start from 0 that is in the above the indices of the 'array' runs from 0-4 (since size of the array is 5), So therefore accessing array[5] is not acceptable.

11.2.2. Floating point error

An operation that leads to unterminating floating point will be reported as Floating point error by the compiler.

Example: division with 0.

```
#include <stdio.h>
void main()
{
      int x;
      int y=314;
      x=y/0;
}
```

The above would result in Floating point error because division of an integer with 0 is not legit in this language as it would result in an unterminated floating point number.

```
#include <stdio.h>
void main()
{
      int x=22/7;
}
```

The above program wouldn't result in any error even though 22/7 is non terminating because the compiler would just initialize x with the integral part of 22/7 as the x is declared under data type 'int' .

11.2.3. Variable out of scope error

Every variable in this language has some scope that is the area of the program only through which this variable can be accessed. Accessing the variable from a function or piece of code in the program which is not in the scope of the variable then the compiler would report this as a Variable out of scope error.

Example:

```
#include <stdio.h>
bool find(int arr[],int size, int num)  {
 for(int i=0;i<num;i++) {
       if(arr[i]==num)
       break;
   }
   if(i==num)
       return(false);
   else
       return(true);
}
void main() {
   //program to find whether the given number is present
in the array
   int x=5; //given number
   int arr[]={32, 67, 123, 987};
   int n=sizeof(arr)/sizeof(arr[0]);
   find(arr,n,x);
}
```

In the above program, consider the function ’find’ in which focusing on the ’for’ loop where the variable ’i’ is initialized and declared that is the scope of this variable is only within this ’for’ loop and therefore by accessing the variable out of this ’for’ loop would result in Variable out of scope error.

11.2.4. Unterminated string constant

If there is an unescaped new line in a string then the compiler considers this as a lexical error and reports it as an unterminated string constant error.

Example:

```
#include <stdio.h>
void main()
{
   char str[]="The language C+ is cool
   but it is just a prototype"
}
```

The above program would result in EOF in the comment error because the program reached the end of the file even before the multi-line comment in the 'main' function is closed.

11.2.5. EOF in comment

If there is EOF inside a comment then the compiler considers this as a lexical error and reports as EOF in the comment error.

```
#include <stdio.h>
int length_of_str(char str[]){
   int i=0;
   while(str[i]!='\0')
       i++;
   return(i);
   /*in this language the end of every string is
indicated with a special character named null
character('\0')*/
}
void main() {
   char str[]="Mahesh is whitehat Jr's chintu";
   //program to find the length of the given string;
   length_of_str(str);
   //final value of i is the length of the given string
```

```
   /*The function 'length_of_str' will return the length
of given string
}
```

The above program would result in EOF in the comment error because the program reached the end of the file even before the multi-line comment in the 'main' function is closed.

11.2.6. EOF in string constant

If there is EOF inside a string then the compiler considers this as a lexical error and reports as EOF in the string constant error.

```
#include <stdio.h>
void main()
{
      char str[]="There are not many good anime to
      watch during this COVID-19 pandemic
}
```

The above program would result in EOF in string constant error because the program reached the end of the file even before the string which is to be initialised to 'str' is closed or completed.

11.2.7. Unbalanced Parentheses

Parentheses have special meaning in regular expressions supported by the lexer of C+. If the parentheses are not

balanced considering the rules of lexer then the compiler would report this as Unbalanced parentheses error.

```
#include <stdio.h>
void main()
{
      for(int i=0;i<10;i++)
{
      //some code
}
```

In the above program, the no.of opening parenthesis and no.of closing parenthesis are not equal that is not balanced which results in lexical error named Unbalanced parentheses.

12. Standard I/O

C+ supports 2 pre-defined functions 'printf'-to write output to the console and 'scanf' to read input from from the console. These functions (printf and scanf which are similar to as that of C language) enables the user to control the input and output stream according to their desire.

Project Plan:

1) For deciding on the project we have done a various search for each feature, finally on 30 September 2020, we have concluded to implement our language C+(as it has features intermediate between C, C++) and compiler ECOOP(Exceptional C with Object-Oriented Programming).
2) After deciding on our language, we started working on its design, and our Language Guru completed it by 10 October 2020.
3) Meanwhile, the System integrator was looking into various compiler tools(like LLVM, FLEX and BISON, ANTLR or to simple CPP) for finding the best suitable tool for our project. Finally, on 8 October 2020, we have finalised our working tool as ANTLR after taking into consideration the pros and cons of each tool.
4) Meetings were conducted very frequently, and the project manager has maintained a log of minutes of each meeting.
5) We have planned to complete the lexer by Oct 16 and we have successfully completed.
6) We have decided to use GitHub for sharing code among team members.
7) We have decided to divide the work uniformly and discuss the progress and further plans in weekly meetings

COMPILER PROJECT PLAN

How it All Went Down

(Sept 30, 2020)

Birth of C+ language, ECOOP Compiler

(Oct 8, 2020)

Flnalised ANTLR as working tool

(Oct 7, 2020)

Language Design Completed

(Oct 3, 2020)

Comparing Different Compiler Tools

(Oct 12, 2020)

Comparision of LLVM vs GCC

(Oct 14, 2020)

Completed Kaleidoscope LLVM tutorial

(Oct 16, 2020)

Lexer Completed in ANTLR

(Nov 1, 2020)

Started Error Handling

(Oct 28, 2020)

Parse Tree Generation completed

(Oct 23, 2020)

Finished White paper for the language

(Nov 10, 2020)

Written Test files

(Nov 18, 2020)

Started Working on Semantic Analyser

(Nov 28, 2020)

Started Reading on LLVM IR

(Dec 15, 2020)

Demo vedio on C+ and ECOOP

(Dec 18, 2020)

Github Classroom Submit, Final Report for ECCOP

https://github.com/IITH-COMPILERS2/compilers2-project-team-10

Language Evolution:

We were exploring through various projects regarding compilers and features of the language to make it efficient and straightforward to use.

We first started programming in C, so we were more comfortable with it and decided to develop the compiler for a similar language with some additional features to make it more interesting.

We felt that the main drawback of C is that it doesn't support Object-Oriented programming. So we decided to design a new language retaining the features of C with the addition of support to the Object-Oriented Programming.

Along with the Object-oriented part, we decided to add some more simple features making the language more comfortable. We added bool data type, keywords for logical operators, predefined constants for pi and e.

We have observed that C language does not support nested comments, function overloading but C++ supports whereas C++ does not support Lexical Scoping but GCC version of C supports. After seeing the facility of ranged for loop in python, we ought to add this feature in our language. We thought Redefining the C language wouldn't be much negotiable. So we have gone into our own Language and have developed a compiler for the same.

Compiler Architecture

1. Lexical Analyzer :

The compiler uses Antlr support for the lexical analyzer. It takes the input source code and breaks it into a sequence of tokens to be passed on to later stages. The Lexer handles the case sensitivity of pre-defined keywords. It checks for invalid characters in case of strings. It also checks if string flows out of memory. Once the Lexer finishes converting code into a sequence of tokens, the compiler uses a script to move the generated tokens to the parser.

2. Parse Tree Generator :

The parser also is written in Antlr. The parser takes the sequence of tokens from lexer as input, checks for syntax issues in the code and then finally resolves the ambiguity in the input string tokens order (if any) and develops an internal representation of the derivation tree i.e., parse tree.

3. Semantic Analyzer :

The semantic analyzer is responsible for identifying semantic errors in the generated parse tree. We have tried to complete the semantic analysis using an ANTLR generated listener, but we couldn’t really get it to work. Our lack of proper resources meant that we couldn’t move beyond this step.

4. Code Generation :

When we had decided on our language specifications and started to work on the compiler, we thought of using LLVM IR as our Intermediate representation output for the code generator. Because it was a new language, an entirely new compiler would have many compatibility issues but this step makes it easier because once converted to LLVM IR, we can convert the code to be compatible on most compilers.

But because we failed to move beyond the semantic analyzer, we never got to start working on the code generator.

Development and Environment

This programming language was developed in the Linux Environment. The tools used were ANTLR “Another Tool for Language Recognition” (for generating lexer and parser), JDK “Java Development Kit” (for compiling and running java files), Visual Studio Code as text editor.

To run the lexer and parser files, we prepared two bash scripts **compile_grammar.sh** and **compile_java.sh** and were executed in order.

compile_grammar.sh

1. We enter into the grammar directory
2. We add ***antlr-4.5-complete.jar*** to our environment variable classpath.
3. We compile the grammar files EcoopLexer.g4 and EcoopParser.g4 using the above jar file and the java files obtained after compiling are exported to the ecoop folder present in java directory.
4. Once the script executes successfully, a message appears saying execution of script is successful.

```
#!/bin/bash

cd ./grammar

export CLASSPATH=".:/usr/local/lib/antlr-4.5-complete.jar:$CLASSPATH"

java -jar /usr/local/lib/antlr-4.5-complete.jar -o ../java/ecoop/
EcoopLexer.g4 EcoopParser.g4

echo "Execution Completed Successfully"
```

compile_java.sh

1. The above exported java files are now compiled using the ***javac*** command.
2. The user is prompted to enter the name of an input test file. The relative path is maintained with respect to the **testcases** folder.
3. We present the parse tree generated using a GUI feature of ANTLR by the command in the last line of the script.

4. We can also present the parse tree in terminal by using the same command but replacing ***-gui*** with -***cli*** option

```
cd ./java

export CLASSPATH=".:/usr/local/lib/antlr-4.5-complete.jar:$CLASSPATH"

cd ./ecoop

javac *.java

read -p "Enter the name of testcase file: " name

java org.antlr.v4.runtime.misc.TestRig Ecoop cplusgrammar -gui <
../../testcases/$name
```

Test Plan and Test Suites:

We have written multiple test files to test every feature supported by our language and we have ensured that files which are syntactically correct give corresponding output and files which are syntactically wrong give appropriate errors.

Code in the picture below is a sample C+ program which demonstrates the use of operators and predefined constants. This code has no errors.

```
int main()
{
    int a=5;
    char c='t';
    float f=_pi; // _pi is predefined constant
    bool d=true;

    float sum= 4.5 + _e; //_e is predefined constant
          sum+=sum;
    float product = 4.2 * a;
          product*=product;
    float div = 4.2/4;
          div/=2;
    float diff = f - a;
          diff-=3.2;
    int rem= 102%7;
        rem%=2;


  return 0;
}
```

Code in the picture below is a sample C+ program, which contains many errors such as array out of bound,missing semicolon,floating exception and variable out of scope.

```
int main()
{

    int a[5];  // array out of bound
    a[5]=10;

    int temp=0 // missing semicolon

    float err = 3.0/0;  // floating point exception
    int i=0;
    if(i==0)
    {
        int num=0;
    }
    else
    {
        int num=1;
    }
    num=2;  // variable not in scope


}
```

Conclusions:

What we planned :

- To build a working compiler for our language C+.
- We planned to add extra features like bool data type, predefined constants,function-overloading,range-for loop
- To organize weekly meets as much as possible to compensate for the online semester where we couldn’t meet in one place to discuss ideas

What went good :

- Weekly and regular meetings
- Lexer and Parser successfully done
- Coordination among team members
- Version control helped us to keep track of work

What could we have done better :

- Mentor interaction

What lessons we learned :

- It is difficult to manage projects in large groups in the online semester.
- We got great experience in team management etc,.
- Designing a language and building a compiler for the features we plan is not an easy task
- Communicating concepts with team members becomes an important task - especially when you are working on a project together.

Appendix:

- **ECOOP Folder :**
  - **Grammar**
    - EcoopLexer.g4
    - EcoopParser.g4
    - compile_grammar.sh
  - **Java**
    - **ecoop**
      - EcoopLexer.java
      - EcoopParser.java
      - EcoopLexer.tokens
      - EcoopParser.tokens
    - compile_java.sh
  - **testcases**
    - **Language_testcases**
      - nested_comments_test.cp
      - bool_test.cp
      - function_overloading_test.cp
      - lexical_scope_dynamic.cp
      - lexical_scope_static.cp
      - logical_operator_test.cp
      - predefined_constants_test.cp
      - Ranged_loop_test.cp
    - test1.cp
    - test2.cp
    - test3.cp
    - test4.cp
    - test5.cp
  - Makefile
- CS3423_whitepaper_CS18BTECH11001.pdf
- Compilers Presentation.pdf
- Final Project Report.pdf
- README.md

THE END