CPS2000- Compiler Theory and Practice Course Assignment

Nicholas Vella 0440803L

Task 1 - Table-driven lexer

For the first task, I started off by drawing the UML state diagram of the major EBNF rules which are: <Expr>, <VariableDec>, <Assignment>, <Identifier>, <Type>, <PrintStatement>, <DelayStatement>, <RtrnStatement>, <FunctionDecl>, <IfStatement>, <ForStatement>, <WhileStatement>, <PixelStatement>, <Block>, <Statement>

I drew them in the order mentioned above, as shown in the figure below, the diagrams are not perfectly drawn but they gave me a clear path for when I went to code the transition table. I started with the expression EBNF because it is the most complicated one as it had many EBNFs in other EBNFs so I felt the safe thing to do was to start from it. Besides the complication of the Expression EBNF, I noticed that the Expression EBNF was almost in every other major EBNF. The approach that I took was to draw a piece of the UML diagram, code it and test it. Like this, I was making sure that the UML diagram is drawn correctly and so is the transition table.

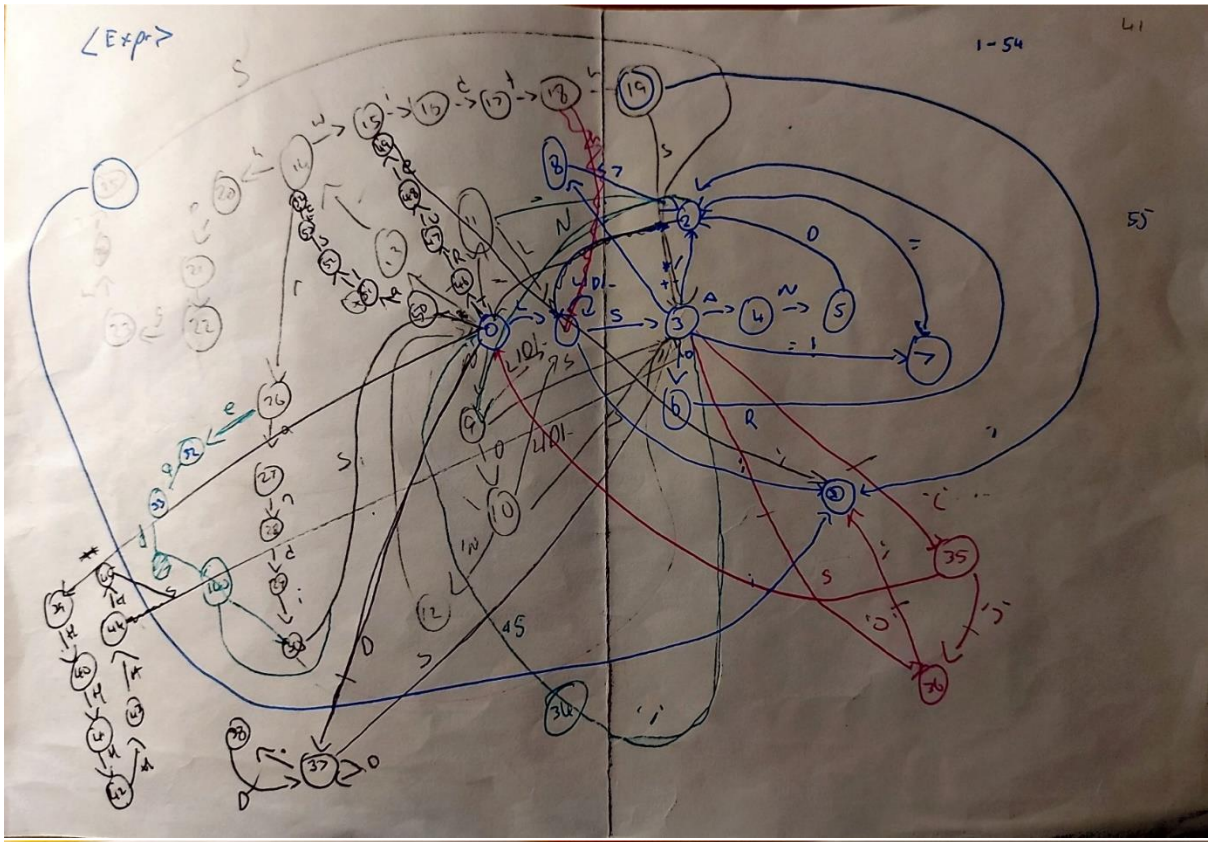
The style of transition table that I work with, is in the form of multiple dictionary entries. The keys in the dictionary represent the current state and the input symbol that triggers the transition. For example, (13, '_') indicates that if the current state is 13 and the input symbol is '_', a transition to state 14 should be taken. The values in the dictionary represent the next state after the transition is taken, i.e. transition_table[(13, '_')] = 14 means that the transition from state 13 with input symbol '_' leads to state 14. I decided to use this approach instead of a 2D array or a matrix because my approach is easier to debug and adjust, considering I had around 203 states. In fact, I started the project working with the 2D array but then switched to a dictionary approach.

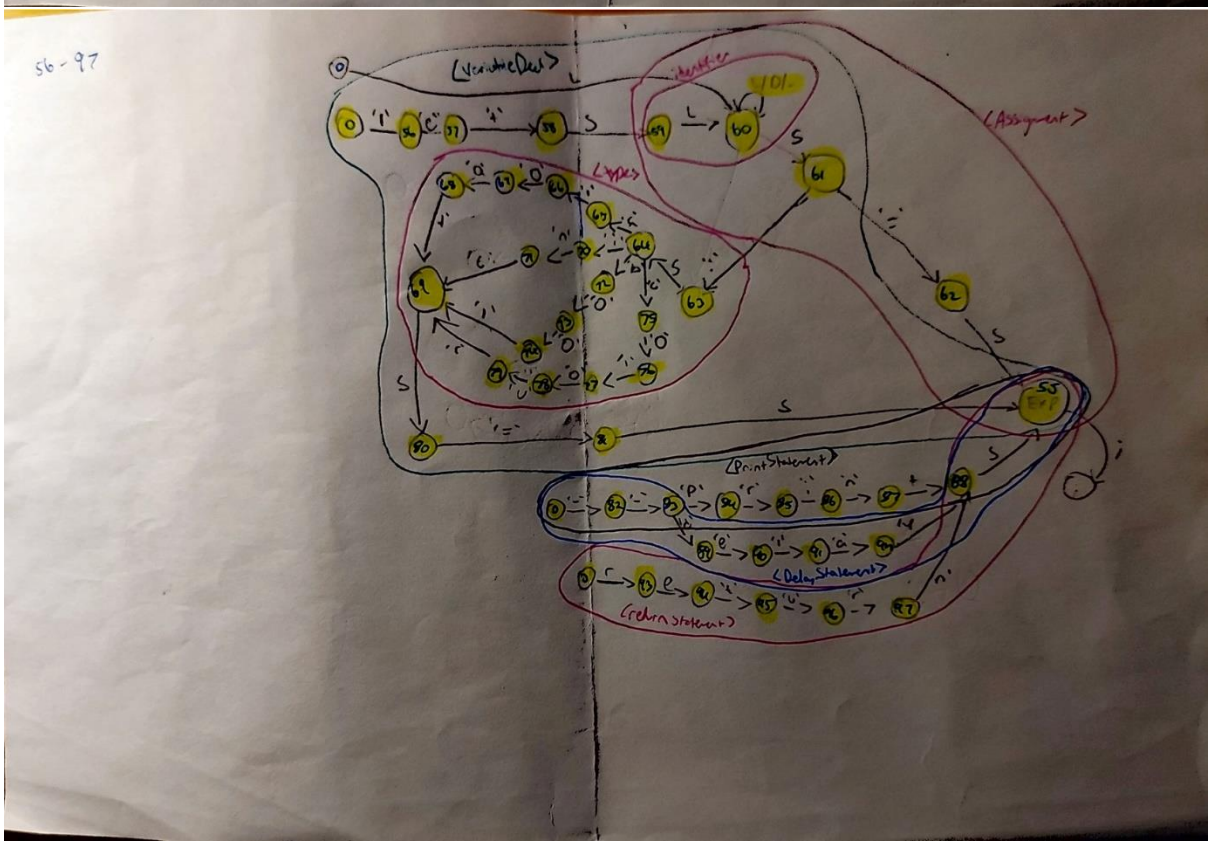
To take care of the transition table, I created a class called Lexer. The Lexer class has several attributes. transition_table stores the transition rules that determine the next state based on the current state and input character. accepting_states holds the states that indicate a valid token has been identified(final states). current_state keeps track of the current state during the lexing process, while current_token accumulates the characters of the current token being processed. The tokenTable attribute is a table that maintains recognized tokens.

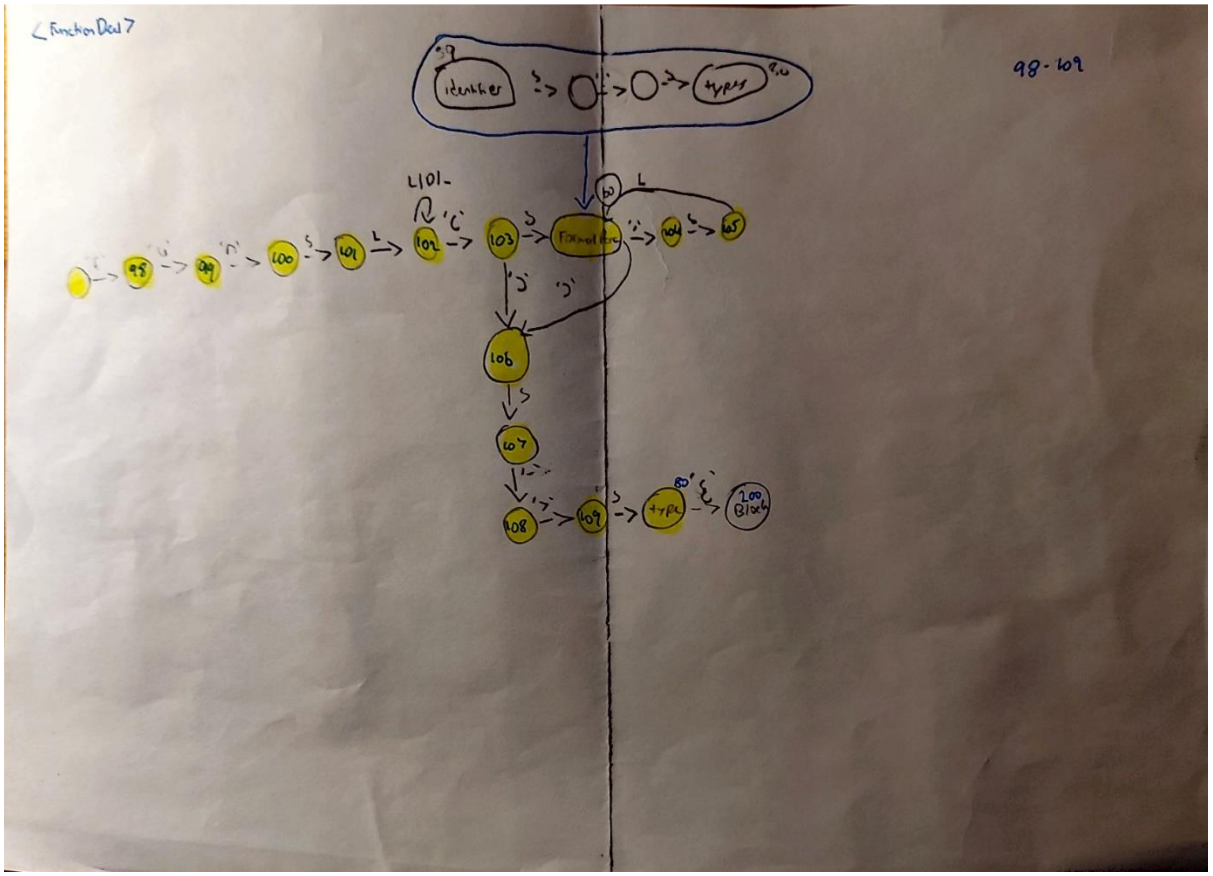
The currentToken method is responsible for handling specific states and tokens encountered during the lexing process. It checks the current state and token and performs the necessary actions. For example, if the state is 58, it adds a token to tokenTable representing a variable declaration with the value 'LET'. This will come in handy for the other tasks.

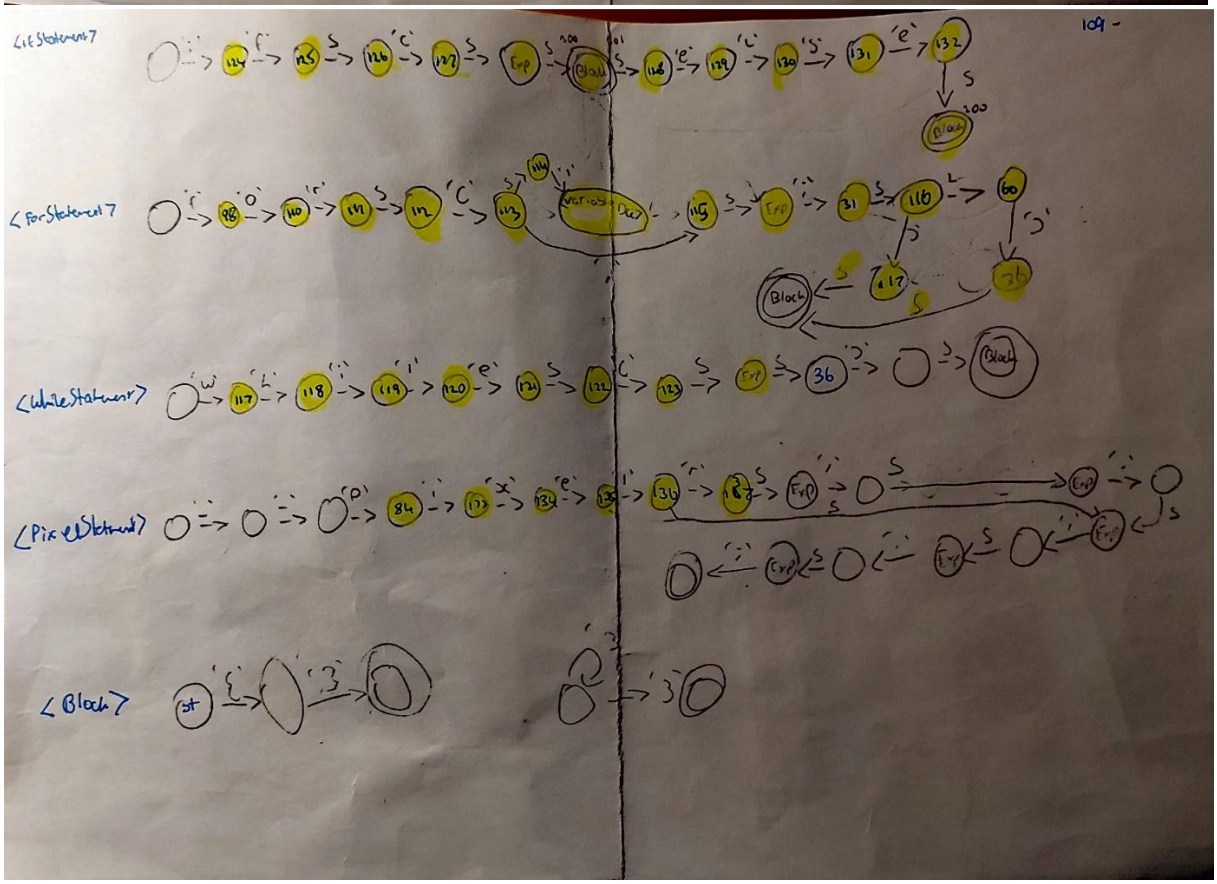
The get_next_token method is the main method for tokenizing an input string. It takes an input string and iterates over each character. For each character, it looks up the next state in the transition_table based on the current state and current character. If a transition is found, the current state and token are updated, and the corresponding action is performed using the currentToken method.

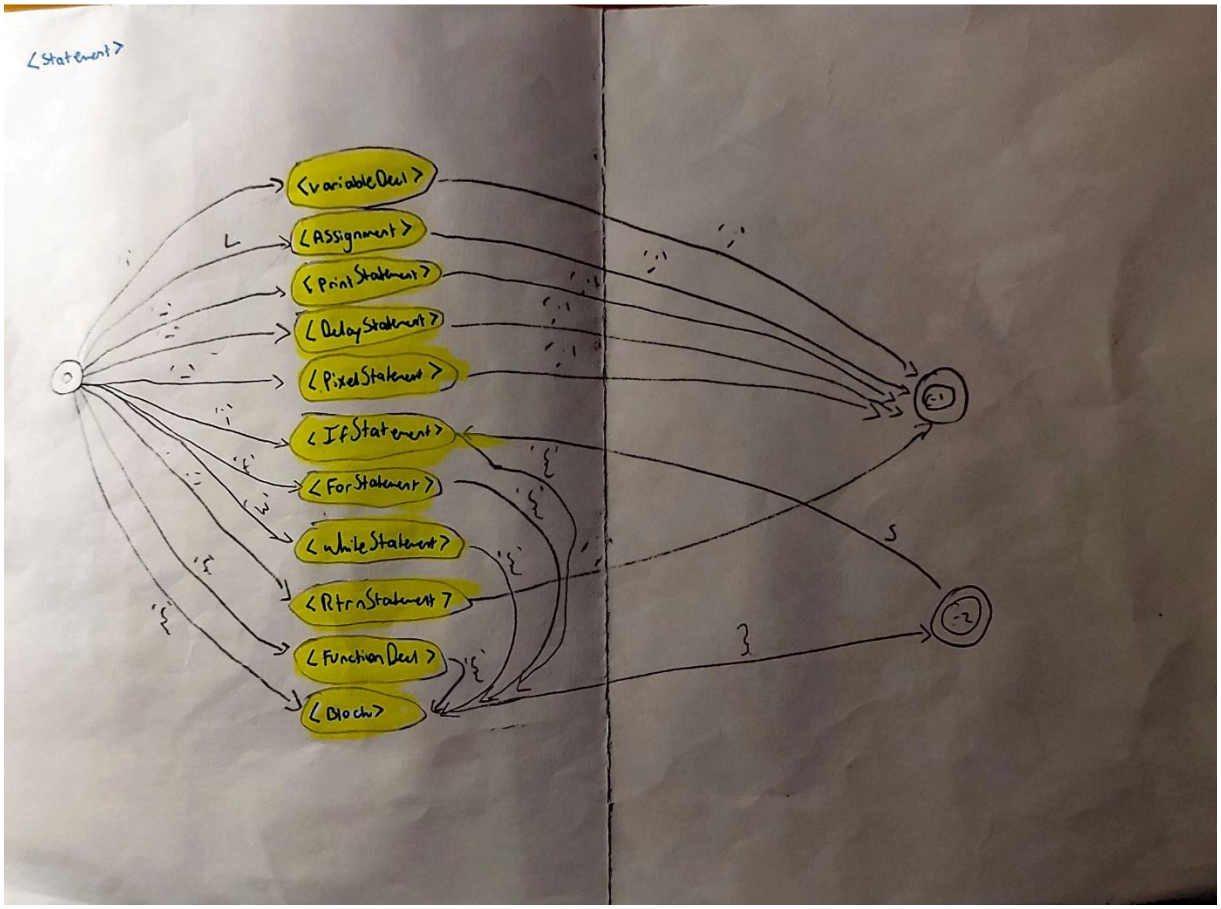
If the lexer reaches an accepting state, indicating a valid token has been recognized, it returns the tokenTable. Otherwise, it returns an indication of invalid syntax.


<Expr>
1-54
41
55


56-97
<VariableDecl>
<Assignment>
<Expr>
<PrintStatement>
<DelayStatement>
<ReturnStatement>


<FunctionDecl>
98-109
identifier
types
FormalArg
Block


109 -
<IfStatement>
<ForStatement>
<WhileStatement>
<PixelStatement>
<Block>


<Statement>
<VariableDecl>
<Assignment>
<PrintStatement>
<DelayStatement>
<PixelStatement>
<IfStatement>
<ForStatement>
<WhileStatement>
<RtrnStatement>
<FunctionDecl>
<Block>
0
1
2

Task 2 - Hand-crafted LL(k) parser

For this task, I create a parser class. The parser class has an initializer method that takes a token table as input. The same table that the lexer class returns if the syntax is correct. The parser also has a current_token_index variable to keep track of the current token being parsed.

The parser contains several methods, each corresponding to a grammar rule of the language, like 'parse_simple_expr(self)', in each of them there is a sequence that if the token table manages to go through it all, means that the syntax is correct. The main method calls def parse_sel(self) method, and this method uses the token table to determine what appropriate method it should call for example if statement, while loop, for loop etc… In short, this method is the one taking care that the parsing sequence is moving as in the token Table. There is also a similar class called def parse_selector(self), the code is similar code-wise, the difference is how I used them. This method is used only by the block, so after, for example, an IF statement does its expression then it calls the block (meaning '{}') and the block method calls the def parse_selector(self) method to take care of the code inside the parenthesis. I know it is repetitive but when I tried using the same one the current_token_index was not working well so I decided to implement the function parse_selector to remove this complication and like that it is working well.

There is an important method called, def error(self, message). This given a message, will stop the execution of the program and prompt the user with the message. The message will differ depending on the function it is called from.

In this task, the part that took me the longest time to get right was the block statement. It was tricky because I had to identify where the block stats start and end in the token table. And also made sure that the self.current_token_index was updating correctly, but luckily after lots of time, I managed to make it work perfectly.

Below is an example of the input:

```
---- PASER ----
[{'Function_Decl': {'Function name': 'Tace', 'params': [('p1_c', ' COLOUR '), ('p2_c', ' COLOUR '), ('score_max', ' INT ')], 'Return type': ' INT ', 'Block': {'Block':
 {'Statement': [{'Variable_Decl': {'name': 'p3_toss', 'type': ' INT ', 'expr': {'PadRandI': '__randi', 'Expression': {'Integer': '1000'}}}}, {'Variable_Decl': {'name':
 'p1_score', 'type': ' INT ', 'expr': {'Integer': '0'}}}, {'Variable_Decl': {'name': 'p1_toss', 'type': ' INT ', 'expr': {'PadRandI': '__randi', 'Expression': {'Integer':
 '1000'}}}}, {'Variable_Decl': {'name': 'p2_score', 'type': ' INT ', 'expr': {'Integer': '0'}}}, {'Variable_Decl': {'name': 'p2_toss', 'type': ' BOOL ', 'expr':
 {'BooleanLiteral': 'True'}}}, {'IF_Stat': {'Expression': {'Expression Op': '>', 'Left Side': {'ExpVariable': 'p1_toss'}, 'Right Side': {'ExpVariable': 'p2_toss'}}, 'Block':
 {'Block': {'Statement': [{'Variable': {'Name': 'p1_score', 'Expression': {'Expression Op': '+', 'Left Side': {'ExpVariable': 'p1_score'}, 'Right Side': {'Integer':
 '1'}}}}]}}, 'ELSE_Stat': {'Block': {'Statement': [{'Variable_Decl': {'name': 'p9_toss', 'type': ' INT ', 'expr': {'PadRandI': '__randi', 'Expression': {'Integer': '1000'}}}},
  {'Variable': {'Name': 'p1_score', 'Expression': {'Expression Op': '+', 'Left Side': {'ExpVariable': 'p1_score'}, 'Right Side': {'PadRandI': '__randi', 'Expression':
 {'Integer': '1000'}}}}}, {'PrintStat': {'Expression': {'ExpVariable': 'p1_score'}}}]}}}}, {'For Stat': {'Variable Decl': {'Variable_Decl': {'name': 'p0_toss', 'type': ' INT
 ', 'expr': {'Integer': '0'}}}, 'Expression': {'Expression Op': '+', 'Left Side': {'ExpVariable': 'p1_score'}, 'Right Side': {'Integer': '1'}}, 'Assignment': {'Variable':
 {'Name': 'p1_score', 'Expression': {'Integer': '20'}}}, 'Block': {'Block': {'Statement': [{'Variable_Decl': {'name': 'pp1_score', 'type': ' INT ', 'expr': {'PadRandI':
 '__randi', 'Expression': {'Integer': '1000'}}}}, {'Variable_Decl': {'name': 'p6_toss', 'type': ' INT ', 'expr': {'PadRandI': '__randi', 'Expression': {'Integer': '1000'}}}},
 {'DelayStat': {'Expression': {'Expression Op': '+', 'Left Side': {'ExpVariable': 'p3_toss'}, 'Right Side': {'Integer': '2'}}}}, {'Pixel_Statment': {'Type:': '__pixelr',
 'Expressions': [{'ExpVariable': 'pp1_score'}, {'ExpVariable': 'p6_toss'}, {'Integer': '1'}, {'Integer': '0'}, {'ColourLiteral': '#00ff00'}]}}, {'Pixel_Statment': {'Type:':
 '__pixel', 'Expressions': [{'Integer': '2'}, {'ExpVariable': 'p2_score'}, {'ExpVariable': 'p2_c'}]}}]}}}}, {'Variable_Decl': {'name': 'Hinner', 'type': ' INT ', 'expr':
 {'Integer': '1'}}}, {'IF_Stat': {'Expression': {'Expression Op': '>', 'Left Side': {'ExpVariable': 'p2_score'}, 'Right Side': {'ExpVariable': 'p1_score'}}, 'Block': {'Block':
  {'Statement': [{'Variable': {'Name': 'Hinner', 'Expression': {'Integer': '2'}}}]}}}}, {'ReturnStat': {'Expression': {'Expression Op': '+', 'Left Side': {'PadWidth':
 '__width'}, 'Right Side': {'ExpVariable': 'p3_toss'}}}}]}}}}]
```

As you can see in the above, the AST is in the form of a list of dictionaries. The AST also include details like variable names and types, and for expressions also operation symbol, left and right.

Task 3 - AST XML Generation Pass

This task was easier than the two previous tasks as it simply was to transform the AST into an XML file. For this task, I created a class called, XmlVisitor class that receives the Abstract Syntax Tree (AST) dictionary. The class has methods for visiting the nodes of the AST and generating the corresponding XML output.

Upon initialization, an XmlVisitor instance is constructed with an AST dictionary as an argument. The __init__ method allocates the AST dictionary to the ast_dict variable and initializes the instance variables, xml, and indentation. The visit method is responsible for the majority of the class's functionality. This method accepts an AST node as an argument and recursively visits the AST nodes to build the XML representation.

If the node is a dictionary, the method iterates through the dictionary's key-value pairs. It indents the XML output based on the current degree of indentation and appends an opening tag to the xml string. It then raises the indentation level and runs the visit function on the relevant value recursively. It inserts a closing tag to the xml string after viewing the value and decreases the indentation level.

If the node is a list, the method iterates through the list, calling the visit function on each item recursively. If the node is neither a dictionary or a list, it simply adds the node's string representation to the xml string. The indent method is a helper method that adds the necessary number of indentation spaces to the xml string based on the current level of indentation. The write_to_file method saves the created XML string to the file location supplied. The generate_xml function is used to generate the XML representation. It invokes the visit method on the AST dictionary's root node and returns the final XML text.

For this task to represent the AST perfectly I had to alter so returns from the previous task like:

info = {'Variable Decl' : varDecl, 'Expression': expr, 'Assignment': assignment, 'Block': block}

return {'For Stat': info}

Since I first inputted the information in the info variable and then return it, I was able to make the XML output to look cleaner.

Below is a screenshot of part of the XML AST. The full XML output is in the video presentation, I could not screenshot of the whole thing as there are many lines. I screenshotted the variable declaration of ‘let p1_score : int = 0 ;’ and ‘let p1_toss : int = __randi 1000 ;’

```
<Function_Decl>
  <Function name>
    Tace
    </Function name>
  <params>
    ('p1_c', ' COLOUR ')
    ('p2_c', ' COLOUR ')
    ('score_max', ' INT ')
    </params>
  <Return type>
     INT
    </Return type>
  <Block>
    <Block>
      <Statement>
        <Variable_Decl>···
        <Variable_Decl>
          <name>
            p1_score
            </name>
          <type>
             INT
            </type>
          <expr>
            <Integer>
              0
              </Integer>
            </expr>
          </Variable_Decl>
        <Variable_Decl>
          <name>
            p1_toss
            </name>
          <type>
             INT
            </type>
          <expr>
            <PadRandI>
              __randi
              </PadRandI>
            <Expression>
              <Integer>
                1000
                </Integer>
              </Expression>
            </expr>
          </Variable_Decl>
        <Variable_Decl>···
        <Variable_Decl>···
        <IF_Stat>···
        <For Stat>···
        <Variable_Decl>···
        <IF_Stat>···
        <ReturnStat>···
        </Statement>
      </Block>
    </Block>
  </Function_Decl>
```

Task 4 - Semantic Analysis Pass

For this task, I created the SemanticAnalysis class. The main objective of this class is to make type-checking. These are the type checking that I did in this task:

1. No duplicate declaration.
2. Variables declared before use.
3. __Pixelr: 4 int and 1 color expressions.
4. __Pixel: 2 int and 1 color expressions.
5. No duplicate functions.
6. Return type match function type.
7. Every function has a return statement.
8. Variable Declaration and Assignment both use the same variable type(e.g. if initialized as integer then only integer values can by stored in the variable).

As for the scope, I created a 2d array to represent the scope called symbolTable. The first element is for the global variables, so if there is a variable being used in a function that was declared as a global variable it will still be considered as correct. The second element contains the local variables, an array containing arrays. Each element is an array that represents a scope (a function). When another function is being processed a new element(array) is created in the second location, this is controlled by the Boolean variable newScope. So when the semantic analyzer needs to know if a variable was correctly declared, it first looks for the global variable(location 0), if it is there then correct else it looks in the local variables(location 1), first it always goes to the last array in the array(the other arrays are no more of use) of local variables and iterates through it one by one.

This was a simplification of how I implemented the scopes for variable declaration and assignment. The local variable elements also include function declaration variables and variables assigned in the for loop expression. In the symbolTable, I added the names and types of each variable so I could use the types for checking if the code is correctly assigning a variable and if the code is returning the correct variable type.

I also have an array functionTable array to make sure that a function is not declared multiple times. Below is an example of the symbolTable:

```
---- Semantic Analysis ----
[[], [[['p1_c', ' COLOUR '], ['p2_c', ' COLOUR '], ['score_max', ' INT '], ['p3_toss', ' INT '], ['p1_score', ' INT '], ['p1_toss', ' INT '], ['p2_score', ' INT '],
 ['p2_toss', ' BOOL '], ['p9_toss', ' INT '], ['p0_toss', ' INT '], ['pp1_score', ' INT '], ['p6_toss', ' INT '], ['Hinner', ' INT ']]]]
```

As you can see there are no global variables and since there is only one array in location 1(the local variable) shows that there was only one function.

As for __Pixelr and __Pixel I did some simple type checking from the symbolTable to make sure that the correct types are being enters and similarly for the return checkings.

The video presentation can be downloaded with this link to my github repository: https://github.com/NicholasVella08/Compiler-Theory-and-Practice/blob/main/Video%20Presentation%20-%20Made%20with%20Clipchamp.mp4

Plagiarism Declaration Form

Plagiarism is defined as "*the unacknowledged use, as one's own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines*" (University Assessment Regulations, 2009, Regulation 39 (b)(i), University of Malta).

I, the undersigned, declare that the report submitted is my work, except where acknowledged and referenced. I understand that the penalties for committing a breach of the regulations include loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

Nicholas Vella	CPS2000	09/06/2023
Student's full name	**Study-unit code**	**Date of submission**

Title of submitted work: Assignment

Student's Signature