CPS2000 – Compiler Theory and Practice - Assignment Report

Task 1 – Design and Development of Lexer

The Lexer is responsible for getting the individual tokens from the PArL code, along with their type, so that they can be passed on to the parser. This Lexer was designed using a table-driven approach, allowing for efficient tokenisation. Hence, it uses a transition table to determine the correct type for each token.

The different token types are initialised and put in a class called TokenType of type Enum, which is short for enumeration. This allows each token type in the PArL language to be more readable, understandable, and maintainable. This is especially true during the parsing process, where the parser is told which token type should be checked in which part of the statement.

In the same program file, there is another class called Lexer, which handles all the steps needed for the lexical analysis process. A list of lexeme types is initialised by default inside the class. This is responsible for categorising each existing token in the program, making it easier to assign the correct type to the tokens.

Upon initialisation, the Lexer also sets up the transition table used to represent the state transition function of the DFA for each token, such as variables, numerical values, operators, and keywords. Once the lexeme type is assigned to a token, the Lexer checks the transition table to assign a state to the token, upon which the correct token type is given.

A new function called ValidateColour() was added for the purpose of making sure that the colour value is in the correct format, which is that its length is equal to 7 and that all its characters, except for the first, only have hexadecimal characters. If the function returns true, then the value is of type colour. Otherwise, if it returns false, then an error is returned by the Lexer.

An important part of the Lexer that was implemented was the ability to group the individual letters and other symbols, not just for detecting variables, but also for detecting other keywords, such as datatypes, Boolean values, the pad keywords and even the return arrow. This was done by creating several dictionaries of each type of keyword, where the keys are the keywords themselves, while the values are the token types they correspond to. These would be initialised by default inside the Lexer class.

Then, inside the GetTokenByFinalState() function, a nested if-else tree is put within the if statements for detecting the variables and operators (for the return arrow only), in order to detect if the current token lexeme is equal to any of the defined keywords. This was done to simplify the process without having to create an excessive amount of states just to detect these keywords.

Task 2 – Design and Development of Parser

The goal of the Parser is to ensure that there are no syntax errors in the code and to generate an AST which describes the structure of the program. In this program, a hand-crafted LL(k) parser is used, where k represents the number of symbols required to decide on which production rule to use. Throughout this process, an EBNF of the PArL language, representing the grammar rules, was given, which guided how each type of statement should be parsed. No deviations from the original EBNF were made throughout the development of this assignment.

For the process, separate Abstract Syntax Tree (AST) nodes are created, which determine the AST type for each token or statement. The small AST nodes, particularly those for literal values or variables, only require the token lexeme itself as their parameter, while the large AST nodes, particularly those used for statements, can have smaller nodes as parameters, hence ensuring that the code is maintainable.

The parsing process begins by retrieving the list of individual tokens from the source program given by the Lexer. The parser then goes through each statement checking the type of the first token, along with any other tokens that are deemed necessary for checking. This is done to determine the type of statement needed to be parsed. Once the statement type is determined, the appropriate function is called, which checks for the appropriate type and value for each of the individual tokens in the statement to be retrieved. The AST representation of the token is retrieved. Smaller representations, such as those for variables, integer, float, Boolean and colour literals may be grouped together to form a bigger AST representation of the entire statement.

Along the way, the parser can display syntax errors if it cannot detect a specific token type which is required for a particular statement to work, or if it detects a token type which is not defined in the Lexer. This can also happen after parsing a statement, particularly when there is the lack of a semicolon at the end of the parsed statement.

Once the parsing process is complete, the generated AST nodes for the source program are printed out, along with their values. Whilst the statement AST nodes are printed out, indentation is updated accordingly to print out the value AST nodes contained within the statement nodes. This process continues until the end of the program is reached, where at this point the end token gets detected. Finally, the statement “END of AST” gets printed out to signal that the whole AST is printed out.

This is done using the Visitor design pattern, which is used to describe the operation to be performed on the elements of the AST, which in this case is printing out the AST Nodes. The name of the parent class which manages all necessary operation is called ASTVisitor. The name of the class which is responsible for the printing of the AST nodes is called PrintNodesVisitor. This can be useful, especially for the programmer, to ensure that all the statements in the program are parsed correctly.

A challenge that was encountered in the latter part of the assignment was that the colour values might not work properly after testing the generated PArIR program,

especially when generating a random colour value. This was fixed by converting the colour value to its integer equivalent when parsing the original value. Both the string version and the integer version of the value are printed while traversing the entire program AST. This ensures that colour values are compatible with all types of statements in PArIR.

Another challenge that was encountered took place when parsing for and while loops. Whenever there was a variable declaration statement within the loop, the parser mistakes it for another type of statement, resulting in errors. This is due to the fact that there is variable called self.stmt_list that splits the source program into multiple elements when encountering a semicolon. This especially is a conflict in for loops, as there are the variable initialisation, condition and incrementation parts, which have to be separated using semicolons. This was fixed by incrementing the self.stmt_index variable whenever there is a semicolon while defining the for loop. This change was not needed for while loops since these do not need any semicolons.

One major issue with the parser is that any PArL programs need to be in one single line for it to work. Any attempt to put each of the statements in a new line in a hardcode way results in a NoneType error. The other one is that functions in PArL are not fully supported, even though the functionality is present in the parser. This is because the code generation for functions, which is tackled later, does not work well.

Task 3 – Design and Development of Semantic Analysis

The role of the Semantic Analyser is to go through the source program and ensure that there are no semantic errors in the code. This can range from checking that variables have assigned values according to the proper datatype to checking that a variable or value of a specific datatype is being compared to another variable or value of the same datatype.

This was developed using an implementation of the Visitor design pattern. The class used for semantic analysis is called SemanticAnalyser, which is a child of the ASTVisitor class, meaning that it uses the same functions of the parent class along with its own functions.

In small AST nodes, such as those for integers, float, Boolean and colour values, the Node class type is checked to ensure that the value is of the correct datatype. Otherwise, a message will be printed out stating that the value is not of the assigned datatype, which is defined as a semantic error. For variable checking, the analyser repeats the same process, and then checks to make sure that the variable is declared. If it not declared, a message is printed out stating that the variable has not been declared in the scope, which is also considered a semantic error.

In larger AST nodes, such as variable declarations, assignment statements, and function declarations, the implemented functions for the small nodes are called when required. This enhances readability and accessibility of the program, avoiding unnecessary code repetition. For variable declarations, the datatype is also checked to ensure that the value that will be assigned to the new variable is in the scope of the datatype. If it is not the case, then a message is displayed showing that the value being assigned to the variable is not of the datatype given. These are also checked to ensure that the variable is not already declared within the scope. If it is, then a message stating that this is the case is displayed.

Task 4 – Design and Development of Code Generation

The role of the PArIR code generator is to take the PArL code and generate its PArIR code syntax. This was by far the hardest part of the whole task, as at first it was hard to clearly understand the representation of the ParIR instructions in PArL form. However, slowly but surely, creating the code generator became easier once most things were figured out.

This was also developed using an implementation of the Visitor design pattern to traverse the AST nodes present in the source program. The class used for this process is called PArIRCodeGenerator, which is a child of the ASTVisitor class. This means that, just like the semantic analysis class, it uses the same functions of the parent class along with its own functions. Upon initialisation of the code generator, the instruction ".main" is automatically inserted into the code variable inside the class. This ensures that any ParIR code generated always begins with the ".main" instruction, so as to allow the program to execute properly. A similar thing was done for when the end token is detected. For this, the ASTEndNode is visited and the instruction "halt" gets added at the end of the already generated code. This prevents the hassle of having to input them manually.

A small challenge that was encountered was that, after declaring a Boolean variable, or assigning a new value to an existing Boolean variable, the code gets generated perfectly fine, but when put into the program, it gets stuck trying to store the value in memory. This was when it was noted that the program does not allow Boolean values to be used directly. To fix this, the Boolean value is being parsed, it is retrieved and changed such that it becomes 1 if the value is true or 0 if it is false. This is even done when comparing Boolean values using an if-else statement. This ensures that Boolean values work perfectly when they are generated in ParIR form.

Another challenge that became apparent was that after parsing a program containing a loop or an if-else statement, the code of the sub-blocks got generated before the code of the loop or if-else statement. This was fixed by iterating through the self.stmt_type list and indicating which indexes contain AST nodes which are within the sub-blocks. These are added to a list called index_to_rm which is then sorted in descending order. This was done using nested loops to be precise about which element indexes to remove from the list. The list is sorted in descending order as if it is sorted the other way, then any indexes which might need to be removed might be higher than the size of the list, which would have a huge impact on how the visitor classes operate.

Another noticeable challenge that involves the same above statements was that the program could not keep track the program indexes to jump to on its own. This was solved by creating a new dictionary called self.labels which stores a list of labels and the indexes they store. A new label, along with the current program index at the time of code generation, are created and stored in the dictionary each time before visiting an AST Block node. For example, in if statements, two labels are created, the true label and the end label, while in if-else statements, three labels are created, the true label, the

else label and the end label. Three labels are also created in for and while loops, the start label, the body label, and the end label. These labels are created as placeholders, which are later replaced by the “#PC+offset” instruction, which is discussed below. This functionality works just like in a normal assembly program.

Once the ASTEndNode is visited inside the code generator, the program then uses a separate function called replace_placeholders() which replaces any label name in the code with “#PC”, along with the offset. This is done by iterating through the program indexes, and retrieving the index stored under the label name within the dictionary whenever the label name is encountered. The offset is calculated by subtracting the current program index from the retrieved index value. This ensures that the offset is generated dynamically, making if statements and loops work better.

One small problem that exists in the code generator is how scopes in for loops, while loops and if-else statements are handled. It does not push the number of variables declared within the block into to the stack and calling the “oframe” instruction when entering the block and calling the “cframe” instruction when exiting the block. This is because it was deemed unnecessary by the code generator since the first part is done at the start of the program. Despite this, any generated programs which have a loop or an if-else statement work perfectly fine.

Another noticeable problem is that the stack level is not being used at all. Especially when declaring new variables, the code generator only considers the index in frame, as it deems the stack level to be unnecessary to modify, even if it enters a sub-block. In fact, the stack level always stays at 0.

Task 5 – Design and Development of Arrays in PArL

The general role of arrays is to provide a way of storing a collection of values of the same data type under a single variable name. This is an efficient way of organising and managing data, without having to create new variables to store each of the elements in memory.

In the Lexer, two states are present, one for the start square brackets and another for the end square brackets. This was done in a seamless way. There are also two separate token types for each of these characters. The same thing is true in the lexeme list defined by the Lexer itself. The tokens are generated successfully after thorough testing.

In the Parser, two new function instances for variable declaration and assignment statements respectively are created, this time adding support for arrays. Another function called ParseArrayElement() is also present with the purpose of creating a node which stores the value of an element in an existing array.

This is reflected in the AST program, as several new nodes are present for array support. These are called ASTArrayNode, ASTAssignmentArrayNode, ASTVarDeclArrayNode and ASTArrayElementNode. The first node is solely used for storing arrays, reflected by their assigned variable and list of values, while the last one is used for storing a single element in an array, as explained above. This is useful for many operations, such as printing and assigning the value to a new variable. The second and third nodes allow support for variable declaration and assignments respectively.

Before executing all the functionality for the Visitor design classes, the length of all arrays is added to a counter which is used at the beginning of the program to allocate the necessary space for variables. This ensures that array values do not go undefined by the PArIR program.

One small challenge that existed when creating the code generation for arrays was that it was difficult to keep track of the array values and the array length. This was fixed by creating two separate dictionaries, one called self.array_vals_table and another called self.array_len_table under the PArIRCodeGenerator class, which are then populated accordingly as the program gets traversed. This change was necessary to allow the code generator to generate the code for storing arrays in memory and even when doing operations such as printing. This is especially crucial for accessing individual elements in an array, for the very same operations.

Sample ParL Programs and Testing

Test Cases

1.
    a. Functionality being Tested: Testing that variable declaration statements work as intended.
    b. Expected Output: The values of the variables printed out in order of declaration.
2.
    a. Functionality being Tested: Testing that the write instruction works properly.
    b. Expected Output: A pixel at a random position in the PArIR simulator program changes to a random colour.
3.
    a. Functionality being Tested: Same functionality as 2 but now testing same action being repeated multiple times using for loop.
    b. Expected Output: Multiple pixels at random positions in the PArIR simulator program change to random colours.
4.
    a. Functionality being Tested: Testing that the clear instruction works properly.
    b. Expected Output: All pixels in the PArIR simulator change to a random colour if the Boolean variable cond is true and to black if it is false
5.
    a. Functionality being Tested: Testing that the writebox instruction works properly.
    b. Expected Output: A red box with a width of 12 and a height of 15 is drawn on the PArIR simulator window.
6.
    a. Functionality being Tested: Testing that arrays work properly.
    b. Expected Output: The values of the array elements are printed out in order.
7.
    a. Functionality being Tested: Testing that individual array elements are read properly.
    b. Expected Output: Same output as above but instead of printing out all the array elements, only the array element at the index specified is printed.
8.
    a. Functionality being Tested: Testing that individual array element values can be assigned to other variables.
    b. Expected Output: Same output as above. The difference is that in the console log, it is shown that the array element is stored in another location in memory.

AST and PArIR Output

1. Testing that variable declaration statements work as intended.
    a. PArL Code:

```
let  x:int=  23;
let y : int=  100;
let z:float = 2.3;
let c1:colour = #00ff00;
let cond:bool= true;
__print x;
__print y;
__print z;
__print c1;
__print cond;
```

    b. AST Output:

```
New Block =>
        VARIABLE DECLARATION Node =>
                Variable =>  x
                Data Type:: int
                Integer Value:: 23
        VARIABLE DECLARATION Node =>
                Variable =>  y
                Data Type:: int
                Integer Value:: 100
        VARIABLE DECLARATION Node =>
                Variable =>  z
                Data Type:: float
                Float Value:: 2.3
        VARIABLE DECLARATION Node =>
                Variable =>  c1
                Data Type:: colour
                Colour Value:: 65280 (#ff00)
        VARIABLE DECLARATION Node =>
                Variable =>  cond
                Data Type:: bool
                Boolean Value:: 1
        Print =>  x
        Print =>  y
        Print =>  z
        Print =>  c1
        Print =>  cond
END of AST
```

    c. Semantic Analysis Output:

```
Semantic Analysis Check:
Number of semantic errors: 0
No Semantic Errors detected!
```

d. Generated PArIR Code:

```
.main
push 5
oframe
push 23
push 0
push 0
st
push 100
push 1
push 0
st
push 2.3
push 2
push 0
st
push 65280
push 3
push 0
st
push 1
push 4
push 0
st
push [0:0]
print
push [1:0]
print
push [2:0]
print
push [3:0]
print
push [4:0]
print
halt
```

e. Program Execution:

i. Partial Log:

```
=>>> logs start here
-- 23
-- 100
-- 2.3
-- 65280
-- 1
```

ii. Console Log:

```
=>>> logs start here
[1/34] push 5 |=> [ 5 ]
[2/34] oframe |=> [ ]
[3/34] push 23 |=> [ 23 ]
[4/34] push 0 |=> [ 0 23 ]
[5/34] push 0 |=> [ 0 0 23 ]
[6/34] st |=> [ ]
[7/34] push 100 |=> [ 100 ]
[8/34] push 1 |=> [ 1 100 ]
[9/34] push 0 |=> [ 0 1 100 ]
[10/34] st |=> [ ]
[11/34] push 2.3 |=> [ 2.3 ]
[12/34] push 2 |=> [ 2 2.3 ]
[13/34] push 0 |=> [ 0 2 2.3 ]
[14/34] st |=> [ ]
[15/34] push 65280 |=> [ 65280 ]
[16/34] push 3 |=> [ 3 65280 ]
[17/34] push 0 |=> [ 0 3 65280 ]
[18/34] st |=> [ ]
[19/34] push 1 |=> [ 1 ]
[20/34] push 4 |=> [ 4 1 ]
[21/34] push 0 |=> [ 0 4 1 ]
[22/34] st |=> [ ]
[23/34] push [0:0] |=> [ 23 ]
-- 23
[24/34] print |=> [ ]
[25/34] push [1:0] |=> [ 100 ]
-- 100
[26/34] print |=> [ ]
[27/34] push [2:0] |=> [ 2.3 ]
-- 2.3
[28/34] print |=> [ ]
[29/34] push [3:0] |=> [ 65280 ]
-- 65280
[30/34] print |=> [ ]
[31/34] push [4:0] |=> [ 1 ]
-- 1
[32/34] print |=> [ ]
[33/34] halt |=> [ ]
[Program Execution] HALT
```

2. Testing that the write instruction works properly.
    a. PArL Code:
    let c:colour = (__random_int 16777216);
    let x_1:int = (__random_int 35);
    __print 36;
    let x_2:int = (__random_int __width);
    let y: int = (__random_int __height);
    __write x_2, y, c;
    b. AST Output:

```
New Block =>
        VARIABLE DECLARATION Node =>
                Variable =>  c
                Data Type:: colour
                Random =>  16777216
                        From:  0
                        To:  16777216
        VARIABLE DECLARATION Node =>
                Variable =>  x_1
                Data Type:: int
                Random =>  35
                        From:  0
                        To:  35
        Print =>  36
        VARIABLE DECLARATION Node =>
                Variable =>  x_2
                Data Type:: int
                Random =>  __width
                        From:  0
                        To:  __width
        VARIABLE DECLARATION Node =>
                Variable =>  y
                Data Type:: int
                Random =>  __height
                        From:  0
                        To:  __height
        Write =>
                Variable =>  c
                Variable =>  y
                Variable =>  x_2
END of AST
```

    c. Semantic Analysis Output:

```
Semantic Analysis Check:
Number of semantic errors: 0
No Semantic Errors detected!
```

    d. Generated PArIR Code:
    .main

```
push 4
oframe
push 16777216
irnd
push 0
push 0
st
push 35
irnd
push 1
push 0
st
push 36
print
width
irnd
push 2
push 0
st
height
irnd
push 3
push 0
st
push [0:0]
push [3:0]
push [2:0]
write
halt
```

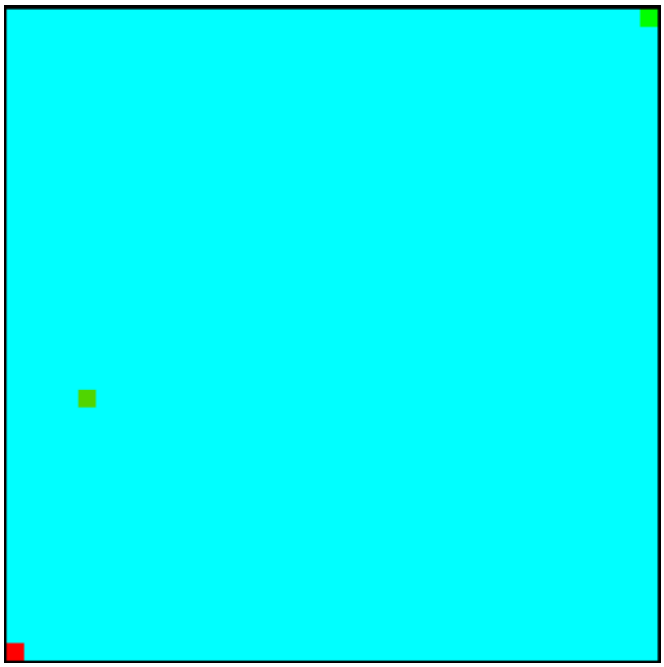
e. Program Execution:
    i. Partial Log:

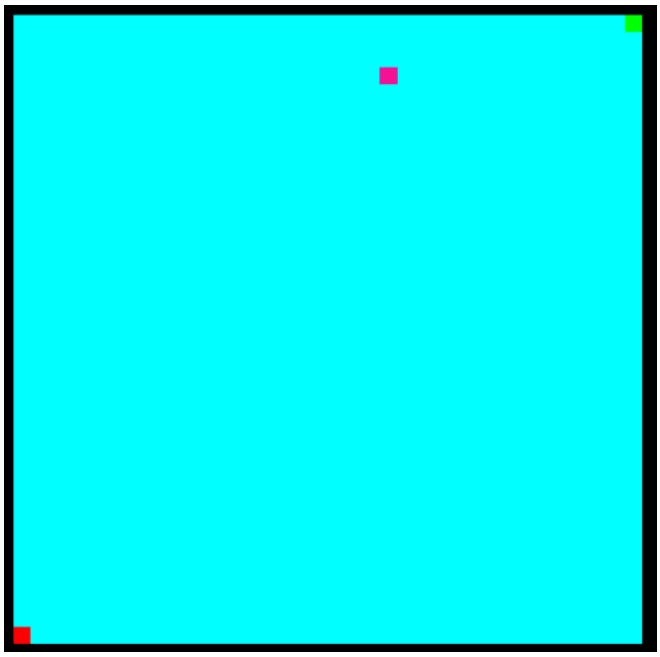
```
=>>> logs start here
-- 36
```

ii. Console Log:

```
=>>> logs start here
[1/30] push 4 |=> [ 4 ]
[2/30] oframe |=> [ ]
[3/30] push 16777216 |=> [ 16777216 ]
[4/30] irnd |=> [ 12989691 ]
[5/30] push 0 |=> [ 0 12989691 ]
[6/30] push 0 |=> [ 0 0 12989691 ]
[7/30] st |=> [ ]
[8/30] push 35 |=> [ 35 ]
[9/30] irnd |=> [ 33 ]
[10/30] push 1 |=> [ 1 33 ]
[11/30] push 0 |=> [ 0 1 33 ]
[12/30] st |=> [ ]
[13/30] push 36 |=> [ 36 ]
-- 36
[14/30] print |=> [ ]
[15/30] width |=> [ 36 ]
[16/30] irnd |=> [ 1 ]
[17/30] push 2 |=> [ 2 1 ]
[18/30] push 0 |=> [ 0 2 1 ]
[19/30] st |=> [ ]
[20/30] height |=> [ 36 ]
[21/30] irnd |=> [ 35 ]
[22/30] push 3 |=> [ 3 35 ]
[23/30] push 0 |=> [ 0 3 35 ]
[24/30] st |=> [ ]
[25/30] push [0:0] |=> [ 12989691 ]
[26/30] push [3:0] |=> [ 35 12989691 ]
[27/30] push [2:0] |=> [ 1 35 12989691 ]
[28/30] write |=> [ ]
[29/30] halt |=> [ ]
[Program Execution] HALT
```

iii. Simulator Window Display:





3. Same functionality as 2 but now testing same action being repeated multiple times using for loop.
    a. PArL Code:

```
for (let counter:int = 0; counter < 10; counter = counter + 1){
    let c:colour = (__random_int 16777216);
    let x_1:int = (__random_int 35);
    __print 36;
    let x_2:int = (__random_int __width);
    let y: int = (__random_int __height);
```

```
    __write x_2, y, c;
}
```

b. AST Output:

```
New Block =>
        FOR Statement =>
                VARIABLE DECLARATION Node =>
                        Variable =>  counter
                        Data Type:: int
                        Integer Value:: 0
                Condition =>
                        Variable =>  counter
                        Relational Operator =>  <
                        Integer Value:: 10
                ASSIGNMENT Node =>
                        Variable =>  counter
                        MODIFIED EXPRESSION Node =>
                                Variable =>  counter
                                Operator =>  +
                                Integer Value:: 1
                        Body:
                        New Block =>
                                VARIABLE DECLARATION Node =>
                                        Variable =>  c
                                        Data Type:: colour
                                        Random =>  16777216
                                                From:  0
                                                To:  16777216
                                VARIABLE DECLARATION Node =>
                                        Variable =>  x_1
                                        Data Type:: int
                                        Random =>  35
                                                From:  0
                                                To:  35
                                Print =>  36
                                VARIABLE DECLARATION Node =>
                                        Variable =>  x_2
                                        Data Type:: int
                                        Random =>  __width
                                                From:  0
                                                To:  __width
                                VARIABLE DECLARATION Node =>
                                        Variable =>  y
                                        Data Type:: int
                                        Random =>  __height
                                                From:  0
                                                To:  __height
                                Write =>
                                        Variable =>  c
                                        Variable =>  y
                                        Variable =>  x_2
END of AST
```

c. Semantic Analysis Output:

```
Semantic Analysis Check:
Number of semantic errors: 0
No Semantic Errors detected!
```

d. Generated PArIR Code:

```
.main
push 5
oframe
push 0
push 0
push 0
st
push 10
push [0:0]
```

```
lt
push #PC+4
cjmp
push #PC+36
jmp
push 16777216
irnd
push 1
push 0
st
push 35
irnd
push 2
push 0
st
push 36
print
width
irnd
push 3
push 0
st
height
irnd
push 4
push 0
st
push [1:0]
push [4:0]
push [3:0]
write
push [0:0]
push 1
add
push 0
push 0
st
push #PC-39
jmp
halt
```

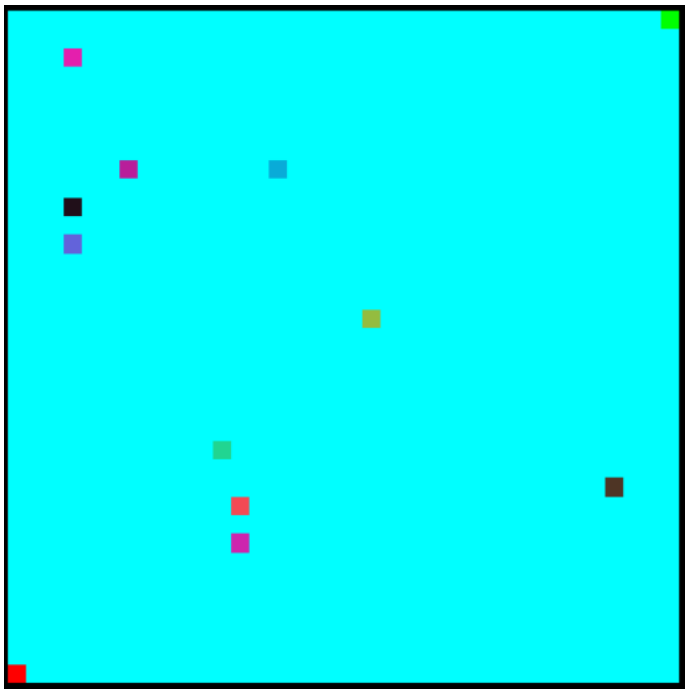
e. Program Execution:

i. Partial Log:

```
=>>> logs start here
-- 36
-- 36
-- 36
-- 36
-- 36
-- 36
-- 36
-- 36
-- 36
-- 36
```

ii. Console Log:

iii. Simulator Window Display:



4. Testing that the clear instruction works properly.
   a. PArL Code:

```
let c:colour = (__random_int 16777216);
let cond:bool = true;
if (cond == true){
    __clear c;
}
else{
    __clear 0;
}
```

b. AST Output:

```
New Block =>
        VARIABLE DECLARATION Node =>
                Variable =>  c
                Data Type:: colour
                Random =>  16777216
                        From:  0
                        To:  16777216
        VARIABLE DECLARATION Node =>
                Variable =>  cond
                Data Type:: bool
                Boolean Value:: 1
        IF-ELSE Statement =>
                Condition =>
                        Variable =>  cond
                        Relational Operator =>  ==
                        Boolean Value:: 1
                IF Body:
                New Block =>
                        Clear =>
                                Variable =>  c
                ELSE Body:
                New Block =>
                        Clear =>
                                Integer Value:: 0
END of AST
```

c. Semantic Analysis Output:

```
Semantic Analysis Check:
Number of semantic errors: 0
No Semantic Errors detected!
```

d. Generated PArIR Code:

```
.main
push 2
oframe
push 16777216
irnd
push 0
push 0
st
push 1
push 1
push 0
st
push 1
push [1:0]
eq
push #PC+4
cjmp
```

```
push #PC+6
jmp
push [0:0]
clear
push #PC+4
jmp
push 0
clear
halt
```

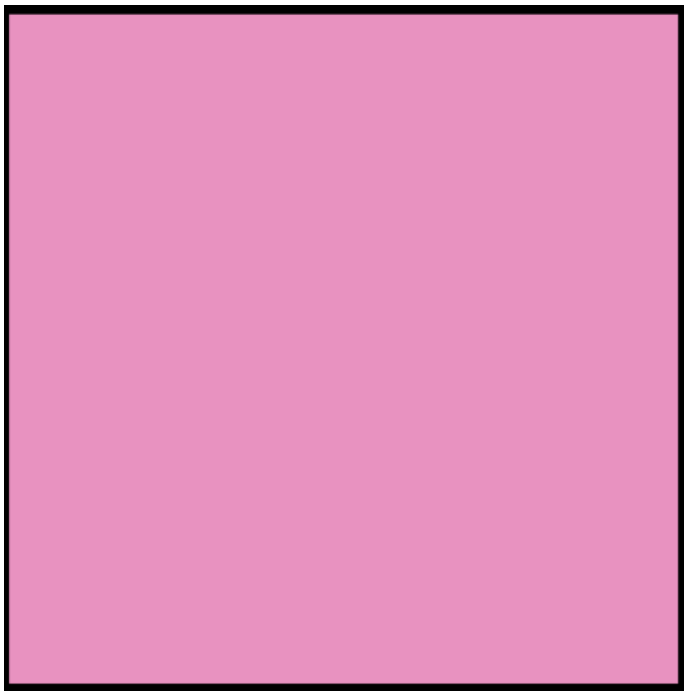
e. Program Execution:
   i. Partial Log:

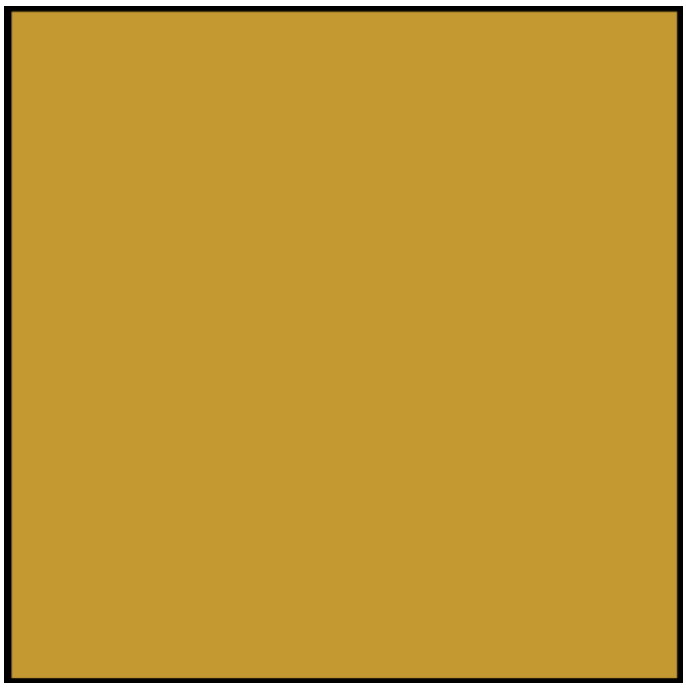
```
=>>> logs start here
[Program Execution] HALT
```

   ii. Console Log:

```
=>>> logs start here
[1/26] push 2 |=> [ 2 ]
[2/26] oframe |=> [ ]
[3/26] push 16777216 |=> [ 16777216 ]
[4/26] irnd |=> [ 15241920 ]
[5/26] push 0 |=> [ 0 15241920 ]
[6/26] push 0 |=> [ 0 0 15241920 ]
[7/26] st |=> [ ]
[8/26] push 1 |=> [ 1 ]
[9/26] push 1 |=> [ 1 1 ]
[10/26] push 0 |=> [ 0 1 1 ]
[11/26] st |=> [ ]
[12/26] push 1 |=> [ 1 ]
[13/26] push [1:0] |=> [ 1 1 ]
[14/26] eq |=> [ 1 ]
[15/26] push #PC+4 |=> [ 19 1 ]
[16/26] cjmp |=> [ ]
[19/26] push [0:0] |=> [ 15241920 ]
[20/26] clear |=> [ ]
[21/26] push #PC+4 |=> [ 25 ]
[22/26] jmp |=> [ ]
[25/26] halt |=> [ ]
[Program Execution] HALT
```

iii. Simulator Window Output:





5. Testing that the writebox instruction works properly.
    a. PArL Code:

```
let w:int = 12;
let h:int = 15;
let y:int = 5;
let x:int = 3;
__write_box x, y, w, h, #ff0000;
```

b. AST Output:

```
 New Block =>
        VARIABLE DECLARATION Node =>
                Variable =>  w
                Data Type:: int
                Integer Value:: 12
        VARIABLE DECLARATION Node =>
                Variable =>  h
                Data Type:: int
                Integer Value:: 15
        VARIABLE DECLARATION Node =>
                Variable =>  y
                Data Type:: int
                Integer Value:: 5
        VARIABLE DECLARATION Node =>
                Variable =>  x
                Data Type:: int
                Integer Value:: 3
        Write Box =>
                Colour Value:: 16711680 (#ff0000)
                Variable =>  h
                Variable =>  w
                Variable =>  y
                Variable =>  x
END of AST
```

c. Semantic Analysis Output:

```
Semantic Analysis Check:
Number of semantic errors: 0
No Semantic Errors detected!
```

d. Generated PArIR Code:

```
.main
push 4
oframe
push 12
push 0
push 0
st
push 15
push 1
push 0
st
push 5
push 2
push 0
st
push 3
push 3
push 0
st
```

```
push 16711680
push [1:0]
push [0:0]
push [2:0]
push [3:0]
writebox
halt
```

e. Program Execution:

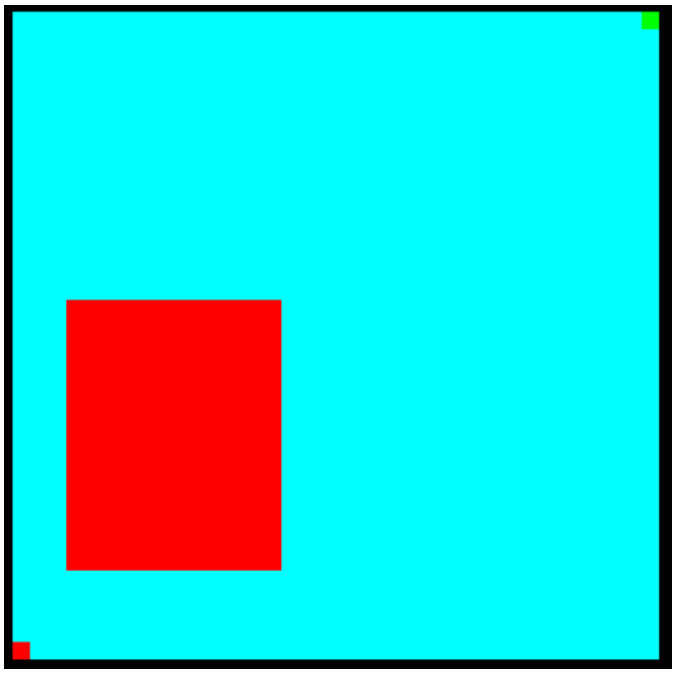
i. Partial Log:

```
=>>> logs start here
```

ii. Console Log:

```
=>>> logs start here
[1/26] push 4 |=> [ 4 ]
[2/26] oframe |=> [ ]
[3/26] push 12 |=> [ 12 ]
[4/26] push 0 |=> [ 0 12 ]
[5/26] push 0 |=> [ 0 0 12 ]
[6/26] st |=> [ ]
[7/26] push 15 |=> [ 15 ]
[8/26] push 1 |=> [ 1 15 ]
[9/26] push 0 |=> [ 0 1 15 ]
[10/26] st |=> [ ]
[11/26] push 5 |=> [ 5 ]
[12/26] push 2 |=> [ 2 5 ]
[13/26] push 0 |=> [ 0 2 5 ]
[14/26] st |=> [ ]
[15/26] push 3 |=> [ 3 ]
[16/26] push 3 |=> [ 3 3 ]
[17/26] push 0 |=> [ 0 3 3 ]
[18/26] st |=> [ ]
[19/26] push 16711680 |=> [ 16711680 ]
[20/26] push [1:0] |=> [ 15 16711680 ]
[21/26] push [0:0] |=> [ 12 15 16711680 ]
[22/26] push [2:0] |=> [ 5 12 15 16711680 ]
[23/26] push [3:0] |=> [ 3 5 12 15 16711680 ]
[24/26] writebox |=> [ ]
[25/26] halt |=> [ ]
[Program Execution] HALT
```

iii. Simulator Window Output:



6. Testing that arrays work properly.
   a. PArL Code:
      let a:int [] = [10, 20, 30, 40, 50, 60, 70, 80, 90];
      __print a;
   b. AST Output:

```
New Block =>
        VARIABLE DECLARATION ARRAY Node =>
                Variable =>  a
                Data Type:: int
                Integer Value:: 10
                Integer Value:: 20
                Integer Value:: 30
                Integer Value:: 40
                Integer Value:: 50
                Integer Value:: 60
                Integer Value:: 70
                Integer Value:: 80
                Integer Value:: 90
                Array Length =>  9
        Print =>  a
END of AST
```

   c. Semantic Analysis Output:

```
Semantic Analysis Check:
Number of semantic errors: 0
No Semantic Errors detected!
```

   d. Generated PArIR Code:
      .main
      push 10

```
oframe
push 10
push 20
push 30
push 40
push 50
push 60
push 70
push 80
push 90
push 9
push 0
push 0
sta
push 9
pusha [0:0]
push 9
printa
halt
```

e. Program Execution:
    i. Partial Log:

```
=>>> logs start here
-- 10
-- 20
-- 30
-- 40
-- 50
-- 60
-- 70
-- 80
-- 90
```

ii. Console Log:

```
=>>> logs start here
[1/21] push 10 |=> [ 10 ]
[2/21] oframe |=> [ ]
[3/21] push 10 |=> [ 10 ]
[4/21] push 20 |=> [ 20 10 ]
[5/21] push 30 |=> [ 30 20 10 ]
[6/21] push 40 |=> [ 40 30 20 10 ]
[7/21] push 50 |=> [ 50 40 30 20 10 ]
[8/21] push 60 |=> [ 60 50 40 30 20 10 ]
[9/21] push 70 |=> [ 70 60 50 40 30 20 10 ]
[10/21] push 80 |=> [ 80 70 60 50 40 30 20 10 ]
[11/21] push 90 |=> [ 90 80 70 60 50 40 30 20 10
]
[12/21] push 9 |=> [ 9 90 80 70 60 50 40 30 20
10 ]
[13/21] push 0 |=> [ 0 9 90 80 70 60 50 40 30 20
10 ]
[14/21] push 0 |=> [ 0 0 9 90 80 70 60 50 40 30
20 10 ]
[15/21] sta |=> [ ]
[16/21] push 9 |=> [ 9 ]
[17/21] pusha [0:0] |=> [ 10 20 30 40 50 60 70 80
90 ]
[18/21] push 9 |=> [ 9 10 20 30 40 50 60 70 80
90 ]
-- 10
-- 20
-- 30
-- 40
-- 50
-- 60
-- 70
-- 80
-- 90
[19/21] printa |=> [ ]
[20/21] halt |=> [ ]
[Program Execution] HALT
```

7. Testing that individual array elements are read properly.
   a. PArL Code:

```
let a:int [] = [10, 20, 30, 40, 50, 60, 70, 80, 90];
__print a[3];
```

b. AST Output:

```
 New Block =>
         VARIABLE DECLARATION ARRAY Node =>
                 Variable =>  a
                 Data Type:: int
                 Integer Value:: 10
                 Integer Value:: 20
                 Integer Value:: 30
                 Integer Value:: 40
                 Integer Value:: 50
                 Integer Value:: 60
                 Integer Value:: 70
                 Integer Value:: 80
                 Integer Value:: 90
                 Array Length =>  9
         Print =>  40
END of AST
```

c. Semantic Analysis Output:

```
Semantic Analysis Check:
Number of semantic errors: 0
No Semantic Errors detected!
```

d. Generated PArIR Code:

```
.main
push 10
oframe
push 10
push 20
push 30
push 40
push 50
push 60
push 70
push 80
push 90
push 9
push 0
push 0
sta
push 3
push 1
add
push 9
sub
push +[0:0]
```

print
halt

e. Program Execution:
   i. Partial Log:

```
=>>> logs start here
-- 40
```

   ii. Console Log:

```
=>>> logs start here
[1/24] push 10 |=> [ 10 ]
[2/24] oframe |=> [ ]
[3/24] push 10 |=> [ 10 ]
[4/24] push 20 |=> [ 20 10 ]
[5/24] push 30 |=> [ 30 20 10 ]
[6/24] push 40 |=> [ 40 30 20 10 ]
[7/24] push 50 |=> [ 50 40 30 20 10 ]
[8/24] push 60 |=> [ 60 50 40 30 20 10 ]
[9/24] push 70 |=> [ 70 60 50 40 30 20 10 ]
[10/24] push 80 |=> [ 80 70 60 50 40 30 20 10 ]
[11/24] push 90 |=> [ 90 80 70 60 50 40 30 20 10
]
[12/24] push 9 |=> [ 9 90 80 70 60 50 40 30 20
10 ]
[13/24] push 0 |=> [ 0 9 90 80 70 60 50 40 30 20
10 ]
[14/24] push 0 |=> [ 0 0 9 90 80 70 60 50 40 30
20 10 ]
[15/24] sta |=> [ ]
[16/24] push 3 |=> [ 3 ]
[17/24] push 1 |=> [ 1 3 ]
[18/24] add |=> [ 4 ]
[19/24] push 9 |=> [ 9 4 ]
[20/24] sub |=> [ 5 ]
[21/24] push +[0:0] |=> [ 40 ]
-- 40
[22/24] print |=> [ ]
[23/24] halt |=> [ ]
[Program Execution] HALT
```

8. Testing that individual array element values can be assigned to other variables.
    a. PArL Code:
    let a:int [] = [10, 20, 30, 40, 50, 60, 70, 80, 90];
    let m:int = a[3];
    __print m;
    b. AST Output:

```
New Block =>
        VARIABLE DECLARATION ARRAY Node =>
                Variable =>  a
                Data Type:: int
                Integer Value:: 10
                Integer Value:: 20
                Integer Value:: 30
                Integer Value:: 40
                Integer Value:: 50
                Integer Value:: 60
                Integer Value:: 70
                Integer Value:: 80
                Integer Value:: 90
                Array Length =>  9
        VARIABLE DECLARATION Node =>
                Variable =>  m
                Data Type:: int
                Array Element Value =>
                        Array =>
                                Integer Value:: 10
                                Integer Value:: 20
                                Integer Value:: 30
                                Integer Value:: 40
                                Integer Value:: 50
                                Integer Value:: 60
                                Integer Value:: 70
                                Integer Value:: 80
                                Integer Value:: 90
                                Array Length =>  9
                        Index =>  3
                        Value =>
                                Integer Value:: 40
        Print =>  m
END of AST
```

    c. Semantic Analysis Output:

```
Semantic Analysis Check:
Number of semantic errors: 0
No Semantic Errors detected!
```

    d. Generated PArIR Code:
    .main
    push 11
    oframe
    push 10
    push 20
    push 30
    push 40
    push 50
    push 60
    push 70

```
push 80
push 90
push 9
push 0
push 0
sta
push 3
push 1
add
push 9
sub
push +[0:0]
push 1
push 0
st
push [1:0]
print
halt
```

e. Program Execution:
    i. Partial Log:

```
=>>> logs start here
-- 40
```

ii. Console Log:

```
=>>> logs start here
[1/28] push 11 |=> [ 11 ]
[2/28] oframe |=> [ ]
[3/28] push 10 |=> [ 10 ]
[4/28] push 20 |=> [ 20 10 ]
[5/28] push 30 |=> [ 30 20 10 ]
[6/28] push 40 |=> [ 40 30 20 10 ]
[7/28] push 50 |=> [ 50 40 30 20 10 ]
[8/28] push 60 |=> [ 60 50 40 30 20 10 ]
[9/28] push 70 |=> [ 70 60 50 40 30 20 10 ]
[10/28] push 80 |=> [ 80 70 60 50 40 30 20 10 ]
[11/28] push 90 |=> [ 90 80 70 60 50 40 30 20 10
]
[12/28] push 9 |=> [ 9 90 80 70 60 50 40 30 20
10 ]
[13/28] push 0 |=> [ 0 9 90 80 70 60 50 40 30 20
10 ]
[14/28] push 0 |=> [ 0 0 9 90 80 70 60 50 40 30
20 10 ]
[15/28] sta |=> [ ]
[16/28] push 3 |=> [ 3 ]
[17/28] push 1 |=> [ 1 3 ]
[18/28] add |=> [ 4 ]
[19/28] push 9 |=> [ 9 4 ]
[20/28] sub |=> [ 5 ]
[21/28] push +[0:0] |=> [ 40 ]
[22/28] push 1 |=> [ 1 40 ]
[23/28] push 0 |=> [ 0 1 40 ]
[24/28] st |=> [ ]
[25/28] push [1:0] |=> [ 40 ]
-- 40
[26/28] print |=> [ ]
[27/28] halt |=> [ ]
[Program Execution] HALT
```

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as "the unacknowledged use, as one's own work, of work of another person, whether or not such work has been published" (Regulations Governing Conduct at Examinations, 1997, Regulation 1 (viii), University of Malta).

I / ~~We~~*, the undersigned, declare that the [assignment / ~~Assigned Practical Task report / Final Year Project report~~] submitted is my /~~our~~* work, except where acknowledged and referenced.

I /~~We~~* understand that the penalties for making a false declaration may include, but are not limited to, 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.

* Delete as appropriate.

(N.B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

Matthias Vassallo Pulis	
Student Name	Signature
Student Name	Signature
Student Name	Signature
Student Name	Signature

CPS2000	Compiler Theory & Practice Assignment
Course Code	Title of work submitted

30/05/2024

Date