L-Università ta' Malta
Faculty of Information & Communication Technology

Department of Computer Information Systems

CPS2000 - Compiler Theory and Practice

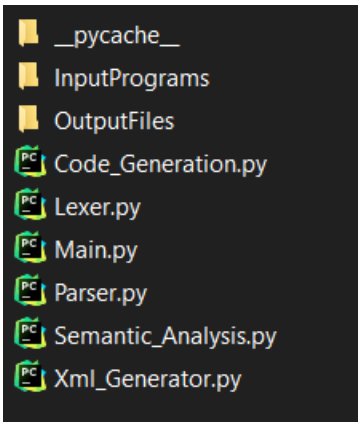
Course Assignment 2022/2023

Table Of Contents

Introduction

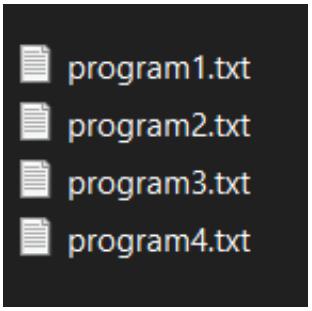
This report details the comprehensive implementation of a compiler for the PixArLang programming language. This project was broken down into 5 tasks, all of which were undertaken.

The code for this project can all be found in the *Code* folder. Each task was written into a separate .py file *(Lexer.py, Parser.py, Xml_Generation.py, Semantic_Analysis.py and Code_Generation.py, for each respective task)*. A *main.py* file calls all the other files and is where the compiler i run from.

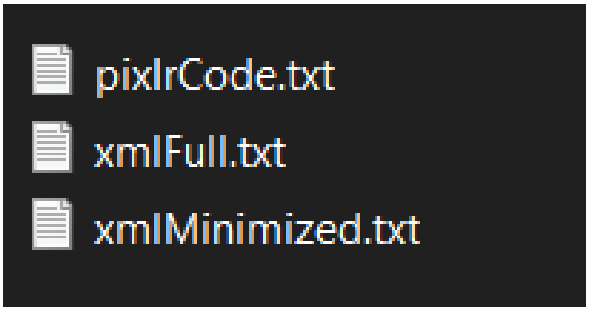


Contents of the folder Code

Two folders *(InputPrograms and OutputFiles)* are used to store the full PixIR programs for the VM. InputPrograms has 4 different programs to try out, changing which requires changing the index in line 10 of main to point at the desired program in the list programs. OutputFiles store the two XML Generations, and the generated PixIR code.



Contents of the folder InputPrograms



Contents of the folder OutputFiles

This report is sectioned in 5 different parts, corresponding to each task outlined in the project brief.

Task 1 - Table Driven Lexer

The aim of the first task was to create a Lexer to break down the inputted code into tokens. In the case of this specific implementation, tokens were stored as Type-Value pairs, for example the token *X* would be saved as *(IDENTIFIER, X)*. Originally the task was to be implemented using a *"Table-Driven"* approach, but due to difficulties with implementation stopping the rest of the assignment to be done, regular expressions were used instead *(through the use of the Python module re)*.

Code Implementation

To implement the lexer, a *Lexer* class was implemented. When creating a new Lexer, the input code is passed as a parameter in the constructor, within which the code is saved to an attribute *code*. The constructor then calls the classes *scanText* function.

The *scanText* function loops through the length of the code string, to begin tokenizing. Whenever it finds a whitespace or newline it skips over it, but in any other case it will attempt to match the the following code to a regular expression *(a list of regular expressions stored in the tokenTable attribute of the Lexer object)*, in which case it will add the token to the *tokenList* attribute. When it reaches the end, the result in *tokenList* should be a fully tokenized version of the input code.

Examples

Below are some examples used to test the lexer. In the first three tests, the tokens should be accepted as valid tokens, but the last test was to see if invalid tokens would throw the expected error.

Example 1 - Valid Tokens in format of EBNF rule

Input Code: `let x:int = 5;`

tokenList: `[('LET','let'), ('IDENTIFIER','x'), (':',':'), ('TYPE','int'), ('=','='), ('INTEGERLITERAL','5'), (';',';')]`

Example 2 - Valid Tokens of same token type

Input Code: `int bool float colour`

tokenList: `[('TYPE','int'), ('TYPE','bool'), ('TYPE','float'), ('TYPE','colour')]`

Example 3 - Valid tokens of different token types

Input Code: `5 12.5 #594939 true X z`

tokenList: `[('INTEGERLITERAL','5'), ('FLOATLITERAL','12.5'), ('COLOURLITERAL','#594939'), ('BOOLEANLITERAL','true'), ('IDENTIFIER','X'), ('IDENTIFIER','z')]`

Example 4 - Invalid input, not recognised as token

Input Code: `_invalidVariable`

Since invalid token, error thrown: `Exception: Error in Syntax`

Task 2 - Hand Crafted LL(k) Parser

The aim of the second task was to create an LL(K) parser to parse the tokens returned by the lexer. For this implementation, since an LL(1) parser is enough for almost all the cases (only one outlier), it follows a recursive descent approach. The end result of the parser is an Abstract Syntax Tree (or AST), made up of "Terminal" and "Non-Terminal" nodes.

The only deviation from the given EBNF to the implementation is in the *<Literal>* rule. The *<PadWidth>*, *<PadHeight>* and *<PadRead>* rules are omitted, since *<Literal>* is only called by *<Factor>*, which itself contains the three omitted rules. As such, including them in *<Literal>* would be redundant, and as such is not implemented.

Code Implementation

To implement the parser, three different classes were used: one class for the parser *(Parser class)* and two classes for different nodes *(TerminalNode and NTerminalNode)*. The node classes are used to describe the different EBNF rules. For this, the rules were divided into 2 categories: rules that were defined by the lexer were Terminal, any other rule would be a Non Terminal. For each respective type of node, a node class was created, the main difference being what the node stores *(terminal nodes store the value of the token, non-terminal nodes store the other nodes that make up the larger rule)*.

The *Parser* class is the class containing all the functions used by the parser. In the constructor, the list of tokens from the lexer is stored, and an empty *tree* attribute is created. The function *parseCode* is then called. This function starts parsing the given tokens, starting with the highest level EBNF rule *(<Program>)*. The rest of the functions were created to deal with every non-terminal rule, and all work in a similar manner.

Example Walkthrough

As an example, below is a walkthrough description of one of these functions: *forStatement*. The *<ForStatement>* rule is described in the EBNF as such:

> *<ForStatement> ::= 'for' '(' [<VariableDecl>] ';' <Expr> ';' [<Assignment>] ')' <Block>*

From this we can infer that there are 4 different cases for a valid *ForStatement*:

1. *'for' '(' ';' <Expr> ';' ')' <Block>*
2. *'for' '(' [<VariableDecl>] ';' <Expr> ';' ')' <Block>*
3. *'for' '(' ';' <Expr> ';' [<Assignment>] ')' <Block>*
4. *'for' '(' [<VariableDecl>] ';' <Expr> ';' [<Assignment>] ')' <Block>*

Since the first two tokens are common for all the cases, both are checked for and accepted. The next step would be checking for the *<Let>* token *(which implies either Case 2 or Case 4)*, and if it's found will proceed with variable declaration. In the case where *<Let>* is not found, it will assume either Case 1 or Case 3. Similarly, after the second *<;>* token, an *<Identifier>* token is checked for, in which case it goes into Case 3 or Case 4 depending on the previous case choices. For each non-terminal rule met *(such as <VariableDecl>, <Assignment>, <Expr>)*, their respective function is called to handle that rule.

Although this only describes the *<ForStatement>* rule, all of the other rules follow a similar methodology.

Examples

For testing, visualisation of the AST tree generated will be covered in the following task, but in this case, the validity of the input could be tested, to make sure that errors in syntax were caught.

Example 1 - Fully valid variable declaration

Input Code: `let x:int = 5;`
Parser Result: `Code Accepted by Parser`

Example 2 - Variable declaration with missing semicolon

Input Code: `let x:int = 5`
Parser Result: `Exception: Error: missing semi-colon`

Example 3 - Variable declaration with missing colon and type

Input Code: `let x = 0;`
Parser Result: `Exception: Error: missing ':'`

Example 4 - Fully valid if statement

Input Code: `if(x < 10){x = 10;}`
Parser Result: `Code Accepted by Parser`

Example 5 - If statement with missing open curly bracket

Input Code: `if(x < 10)x = 10;}`
Parser Result: `Exception: Error: invalid STATEMENT`

Task 3 - AST XML Generation Pass

The aim of the third task was to create a function to traverse the Abstract Syntax Tree *(without changing it)* and print out its contents in an indented XML format. For this task, two kinds of XML were generated: in one every single node is represented accurately *(Full XML)*, in the other non-terminal nodes with only a single child weren't printed, so as to make the XML more easy to digest *(Minimised XML)*.

Code Implementation

To implement this function, the XMLGeneration class was created. The constructor for this class takes the AST from the parser as a parameter. It then creates two XML trees *(minimised and full)* by calling their respective functions. These functions are nearly identical, the only difference being a single if-else statement.

These functions work by recursively going through the AST. Whenever the current node is a non terminal, that node's name is stored on a stack, so that when everything in it has been completely printed, it can pop it off and print the closing line. If the node is a terminal node, it simply prints out the value and returns the line. The only difference in the minimised version, it checks if the node only has 1 child, and if that's the case it simply skips the printing of that node and traverses its one child.

These XML formatted strings are then saved in two files, xmlFull.txt and xmlMinimised.txt respectively *(both of which are located in the OutputFiles folder in the main Code folder)*.

Examples

Below are some visualisations of different AST trees generated by two different *PixArLang* input programs. For each input, both a Full AST and a Minimised AST were generated.

Example 1 - Full AST for "let x:int = 10;"

```
<PROGRAM>
 <STATEMENT>
  <VARIABLEDECL>
   <IDENTIFIER> x </IDENTIFIER>
   <TYPE> int </TYPE>
   <EQUALS> = </EQUALS>
   <EXPR>
    <SIMPLEEXPR>
     <TERM>
```

```
      <FACTOR>
       <LITERAL>
        <INTEGERLITERAL> 10 </INTEGERLITERAL>
       </LITERAL>
      </FACTOR>
     </TERM>
    </SIMPLEEXPR>
   </EXPR>
  </VARIABLEDECL>
 </STATEMENT>
</PROGRAM>
```

Example 2 - Minimised AST for "let x:int = 10;"

```
<PROGRAM>
 <VARIABLEDECL>
  <IDENTIFIER> x </IDENTIFIER>
  <TYPE> int </TYPE>
  <EQUALS> = </EQUALS>
  <INTEGERLITERAL> 10 </INTEGERLITERAL>
 </VARIABLEDECL>
</PROGRAM>
```

Example 3 - Full AST for "let x:int = 10; x = x + 1;"

```
<PROGRAM>
 <STATEMENT>
  <VARIABLEDECL>
   <IDENTIFIER> x </IDENTIFIER>
   <TYPE> int </TYPE>
   <EQUALS> = </EQUALS>
   <EXPR>
    <SIMPLEEXPR>
     <TERM>
      <FACTOR>
       <LITERAL>
        <INTEGERLITERAL> 10 </INTEGERLITERAL>
       </LITERAL>
      </FACTOR>
     </TERM>
    </SIMPLEEXPR>
   </EXPR>
```

```
  </VARIABLEDECL>
 </STATEMENT>
 <STATEMENT>
  <ASSIGNMENT>
   <IDENTIFIER> x </IDENTIFIER>
   <EQUALS> = </EQUALS>
   <EXPR>
    <SIMPLEEXPR>
     <TERM>
      <FACTOR>
       <IDENTIFIER> x </IDENTIFIER>
      </FACTOR>
     </TERM>
     <ADDITIVEOP> + </ADDITIVEOP>
     <TERM>
      <FACTOR>
       <LITERAL>
        <INTEGERLITERAL> 1 </INTEGERLITERAL>
       </LITERAL>
      </FACTOR>
     </TERM>
    </SIMPLEEXPR>
   </EXPR>
  </ASSIGNMENT>
 </STATEMENT>
</PROGRAM>
```

Example 4- Minimised AST for "let x:int = 10; x = x + 1;"

```
<PROGRAM>
 <VARIABLEDECL>
  <IDENTIFIER> x </IDENTIFIER>
  <TYPE> int </TYPE>
  <EQUALS> = </EQUALS>
  <INTEGERLITERAL> 10 </INTEGERLITERAL>
 </VARIABLEDECL>
 <ASSIGNMENT>
  <IDENTIFIER> x </IDENTIFIER>
  <EQUALS> = </EQUALS>
  <SIMPLEEXPR>
   <IDENTIFIER> x </IDENTIFIER>
   <ADDITIVEOP> + </ADDITIVEOP>
```

```
    <INTEGERLITERAL> 1 </INTEGERLITERAL>
   </SIMPLEEXPR>
  </ASSIGNMENT>
</PROGRAM>
```

Task 4 - Semantic Analysis Pass

The aim of the fourth task was to create a function to traverse the Abstract Syntax Tree *(without changing it)* and perform type-checking on it. This component won't return anything, rather it will throw an error if any semantic issues were found in the code.

Code Implementation

To implement this task, two different classes were used, one for the symbol table and another to process the type-checking *(symbolTable and semanticAnalysis respectively)*. The *symbolTable* class stores 2 lists, used to store variables and functions depending on their scope.

The *semanticAnalysis* class' constructor takes the AST from the parser as a parameter, as well as creating an instance of the *symbolTable* object. It then calls the *traverse* function, which recursively traverses through the AST. Depending on the type of node it finds, it will execute a certain number of checks.

Example Walkthrough

As an example, below is a walkthrough description of the checks for an *<Assignment>* node. For an assignment, there are two things to check for:

1. *Check that the variable is already defined*
2. *Check that the variable's value matches its type*

If either of these checks returns a false, the compiler should throw an exception to indicate an error. For the first check, each scope that houses the variable is checked through iteration to find an instance of the variable being referred to. If this is not found, it will throw the error:

```
"Error: variable '" + vName + "' not defined"
```

If this error is not thrown, the next check is done, making use of the *getVarType* function to obtain the typeof the value *(be it a <Literal>, another variable or a <FunctionCall>)*. This result is then compared to the variable being referred to. If they do not match, the following error is shown:

```
"Error: variable '" + vName + "' does not accept type '" +
                    vType + "'"
```

Although this is only a description for *<Assignment>*, all the other non terminal nodes follow a similar pattern and logic.

Rules Checked

The other rules checked by the semantic analysis are *(excluding <Assignment> as they have been mentioned in the above example)* include the following:

Variable Declaration:

1. *Check that the variable is not already declared*
2. *Check that the variable's value matches its type*

Pixel Statement:

1. *__pixel parameters follow the pattern INTEGER, INTEGER, COLOUR/INTEGER*
2. *__pixelr parameters follow the pattern INTEGER, INTEGER, INTEGER, INTEGER, COLOUR, INTEGER*

Return Statement:

1. *Checks that the function has not already been declared*
3. *Checks that the return values match the function type*

Function Declaration:

1. *Checks that the function has already been declared*
2. *Checks if the function does or does not take parameters*
3. *Checks that the function has taken the number of parameters expected*
4. *Checks that the function parameters passed are valid*

Examples

Below are some examples of errors that can be caught by the semantic analyser, testing different rule types and different rules for each.

Example 1 - Type mismatch in VariableDecl

Input Code: `let x:int = 10.1;`

Parser Result: `Error: variable 'x' does not accept type 'float'`

Example 2 - Undeclared variable in Assignment

Input Code: `y = 10`

Parser Result: `Error: variable 'y' not defined`

Example 3 - Invalid format for __pixel in PixelStatment

Input Code: `__pixel 5, 4.1, #123456;`

Parser Result: `Error: invalid format for __PIXEL (int, int, colour)`

Example 4 - Invalid format for __pixelr in PixelStatment

Input Code: `__pixelr 5, 4, 5, 10.4, #123456;`

Parser Result: `Error: invalid format for __PIXELR (int, int, int, int, colour)`

Task 5 - PixIR Code Generation Pass

The aim of the fifth and final task was to create a function to traverse the Abstract Syntax Tree *(without changing it)* and generate PixIr code from the inputted PixArLang code. The result of this component is a series of commands saved in a text file *(pixIrCode.txt)* that when run will generate a specific output in the provided VM.

Through testing, the main limitation found was the declaration of variables within an if statement inside a function. Although attempts to fix this issue were made, it was unable to be resolved and as such limits the kind of code the compiler can take as an input.

Another point to mention, although not exactly a limitation, the *<PadRead>* rule was not implemented as its use was not fully understood during the coding process, so creating a translation for it in *PixIR* was impossible.

It should also be noted that a set of *PixIR* instructions were never used in this implementation. These unused instructions are:

```
[nop, drop, dup, inc, dec, max, min]
```

Apart from these the instruction `cjmp2` was observed in the given code examples, but a formal definition was not given, and as such was not used for good measure *(similarly the* `push[0]` *format for the* `push[0:0]` *instruction was omitted)*.

Code Implementation

To implement this task, a new code generator class *(codeGeneration)* was used, alongside the *symbolTable* class from the previous task. Within the *codeGeneration* class is a list *(self.P)* within which the newly translated code will be stored before being saved into the aforementioned text file.

Within the class' constructor, the translating process is broken down into two parts: the first part traverses the tree *(using the traverse function)* and prints all the commands in the *.main* function, whilst all the *FunctionDecl* nodes are stored in the *functions* attribute of the *symbolTable* and translated one by one after.

In the *traverse* function, the function will check if the node is one of a series of node types, including: *Program*, *Block*, *Statement* and all the rules defined within *Statement*. These are the highest levels of input that a user can enter without causing an error in the semantic analyser. Other than that, all the other *(both terminal and non-terminal)* nodes are referenced in the *getVarValue* and *getVarValueNP* functions.

Example Walkthrough

As an example, below is a walkthrough description of the checks for a <*VariableDecl*> node, specifically, this code:

```
let x:int = 200;
```

The first step is checking if this node is within some scope other than the main one or not *(indicated by if the "list" parameter is None or not)*. For this example, the statement is in the main scope, so it will go through the first branch.

It starts by adding the `push 1` instruction to *self.P* to indicate that 1 memory space is to be saved for the incoming variable, and then checks whether the current scope is empty or not. Since the scope is empty, the functions chooses to add `oframe` to *self.P*.

The next part is not useful in this case, as it is only there as a way to handle nodes that create new frames themselves, even without variables *(such as in If, For and While)*. This is done using a temporary variable with an obscenely long and impractical name that will be instantly removed once found.

Then, the value name of the variable being declared and the value of the variable are stored in respective variables *(to get the value, the function getVarValue is employed)*, and a new variable is created and stored in *self.symbolTable.variable*. After this, the two push instructions for the position are added to *self.P*, and the declaration is finalised with a *st* instruction.

`let x:int = 200;`	Inputted code *(PixIR)*
CODE GENERATION	

`.main`
`push 1`
`oframe`
`push 200`
`push 0`
`push 0`
`st`
`halt`
	Outputted code *(PixArLang)*

Although this is only a description for a single simple program, any program will follow a similar reasoning and logic.

Other Examples

Other examples for code generation that are much longer and more complex can be found either in the **Appendix** as well as in text files located in the InputPrograms folder provided with the source code.

Appendix

Program 1 - Diagonals (Multiple Colours)

This program prints diagonal triangles covering different sectors each time in a looping fashion, where each line making up the triangle is made up of a different colour

PixArLang Code

```
fun getRandomColour() -> int{
    return __randi 16777215;
}

fun printPixels(h:int, w:int, p1:int, p2:int) -> int{
    __delay 10;
    __pixelr h, w, p1, p2, getRandomColour();
    return 0;
}

__pixelr __height, __width, 0, 0, #123456;

let fCall:int = 0;
while (true){

    for (let x:int = __width-1; x >= 0; x = x - 1){
        fCall = printPixels(__height, 1, x, x);
    }

    for (let x:int = __width-1; x >= 0; x = x - 1){
        fCall = printPixels(__height, 1, x, (__width-x)-1);
    }

    for (let x:int = 0; x < __width; x = x + 1){
        fCall = printPixels(1, __width, x, x);
    }

    for (let x:int = 0; x < __width; x = x + 1){
        fCall = printPixels((__height-x), 1, 0, x);
    }
}
```

PixIR Assembly Code

```
.main
push #123456
```

```
height
width
push 0
push 0
pixelr
push 1
oframe
push 0
push 0
push 0
st
oframe
push #PC+5
push 1
cjmp
push #PC+142
jmp
push 1
oframe
push 1
width
sub
push 0
push 0
st
push #PC+7
push 0
push [0:0]
ge
cjmp
push #PC+20
jmp
height
push 1
push [0:0]
push [0:0]
push 4
push .printPixels
call
push 0
push 2
st
push 1
push [0:0]
```

```
sub
push 0
push 0
st
push #PC-23
jmp
cframe
push 1
oframe
push 1
width
sub
push 0
push 0
st
push #PC+7
push 0
push [0:0]
ge
cjmp
push #PC+24
jmp
height
push 1
push [0:0]
push 1
push [0:0]
width
sub
sub
push 4
push .printPixels
call
push 0
push 2
st
push 1
push [0:0]
sub
push 0
push 0
st
push #PC-27
jmp
```

```
cframe
push 1
oframe
push 0
push 0
push 0
st
push #PC+7
width
push [0:0]
lt
cjmp
push #PC+20
jmp
push 1
width
push [0:0]
push [0:0]
push 4
push .printPixels
call
push 0
push 2
st
push 1
push [0:0]
add
push 0
push 0
st
push #PC-23
jmp
cframe
push 1
oframe
push 0
push 0
push 0
st
push #PC+7
width
push [0:0]
lt
cjmp
```

```
push #PC+22
jmp
push [0:0]
height
sub
push 1
push 0
push [0:0]
push 4
push .printPixels
call
push 0
push 2
st
push 1
push [0:0]
add
push 0
push 0
st
push #PC-25
jmp
cframe
push #PC-143
jmp
cframe
halt

.getRandomColour
push 16777215
irnd
ret

.printPixels
push 10
delay
push 0
push .getRandomColour
call
push [3:0]
push [2:0]
push [1:0]
push [0:0]
pixelr
```

```
push 0
ret
```

Program 2 - Diagonals (Solid Colours)

This program prints diagonal triangles covering different sectors each time in a looping fashion, where each line making up the triangle is made up of a single solid colour.

PixArLang Code

```
fun getRandomColour() -> int{
    return __randi 16777215;
}

__pixelr __height, __width, 0, 0, #123456;

let fCall:int = 0;
while (true){

    let z:int = getRandomColour();
    for (let x:int = __width-1; x >= 0; x = x - 1){
        __delay 10;
        __pixelr __height, 1, x, x, z;
    }

    z = __randi 16777215;
    for (let x:int = __width-1; x >= 0; x = x - 1){
        __delay 10;
        __pixelr __height, 1, x, (__width-x)-1, z;
    }

    z = __randi 16777215;
    for (let x:int = 0; x < __width; x = x + 1){
        __delay 10;
        __pixelr 1, __width, x, x, z;
    }

    z = __randi 16777215;
    for (let x:int = 0; x < __width; x = x + 1){
        __delay 10;
        __pixelr __height-x, 1, 0, x, z;
    }
}
```

PixIR Assembly Code

```
.main
push #123456
height
```

```
width
push 0
push 0
pixelr
push 1
oframe
push 0
push 0
push 0
st
oframe
push #PC+5
push 1
cjmp
push #PC+157
jmp
push 1
alloc
push 0
push .getRandomColour
call
push 0
push 0
st
push 1
oframe
push 1
width
sub
push 0
push 0
st
push #PC+7
push 0
push [0:0]
ge
cjmp
push #PC+18
jmp
push 10
delay
push [0:1]
height
push 1
```

```
push [0:0]
push [0:0]
pixelr
push 1
push [0:0]
sub
push 0
push 0
st
push #PC-21
jmp
cframe
push 16777215
irnd
push 0
push 0
st
push 1
oframe
push 1
width
sub
push 0
push 0
st
push #PC+7
push 0
push [0:0]
ge
cjmp
push #PC+22
jmp
push 10
delay
push [0:1]
height
push 1
push [0:0]
push 1
push [0:0]
width
sub
sub
pixelr
```

```
push 1
push [0:0]
sub
push 0
push 0
st
push #PC-25
jmp
cframe
push 16777215
irnd
push 0
push 0
st
push 1
oframe
push 0
push 0
push 0
st
push #PC+7
width
push [0:0]
lt
cjmp
push #PC+18
jmp
push 10
delay
push [0:1]
push 1
width
push [0:0]
push [0:0]
pixelr
push 1
push [0:0]
add
push 0
push 0
st
push #PC-21
jmp
cframe
```

```
push 16777215
irnd
push 0
push 0
st
push 1
oframe
push 0
push 0
push 0
st
push #PC+7
width
push [0:0]
lt
cjmp
push #PC+20
jmp
push 10
delay
push [0:1]
push [0:0]
height
sub
push 1
push 0
push [0:0]
pixelr
push 1
push [0:0]
add
push 0
push 0
st
push #PC-23
jmp
cframe
push #PC-158
jmp
cframe
halt

.getRandomColour
push 16777215
```

```
irnd
ret
```

Program 3 - Enclosing Squares

This program prints a series of enclosing squares, starting from the peripheries and approaching the centre. Upon reaching it, the program will start printing squares for the edges again.

PixArLang Code

```
fun randomColour() -> int{
    return __randi 16777215;
}

fun printSquare(a:int, b:int, c:int, d:int, e:int) -> int{
    let randomColour:int = randomColour();
    __pixelr a, 1, b, b, randomColour;
    __pixelr a, 1, b, c, randomColour;

    __pixelr 1, d, b, b, randomColour;
    __pixelr 1, d, e, b, randomColour;

    return 0;
}

__pixelr __height, __width, 0, 0, #123456;

while(true){
    let u:int = __width-1;
    let v:int = __height-1;

    let h:int = __height;
    let w:int = __width;

    for(let x:int = 0; x < (__width/2); x = x+1){

        let fCall:int = printSquare(h, x, u, w, v);

        u = u-1; v = v-1;
        h = h-2; w = w-2;
        __delay 30;
    }
}
```

PixIR Assembly Code

```
.main
push #123456
```

```
height
width
push 0
push 0
pixelr
oframe
push #PC+5
push 1
cjmp
push #PC+95
jmp
push 1
alloc
push 1
width
sub
push 0
push 0
st
push 1
alloc
push 1
height
sub
push 1
push 0
st
push 1
alloc
height
push 2
push 0
st
push 1
alloc
width
push 3
push 0
st
push 1
oframe
push 0
push 0
push 0
```

```
st
push #PC+9
push 2
width
div
push [0:0]
lt
cjmp
push #PC+49
jmp
push 1
alloc
push [2:1]
push [0:0]
push [0:1]
push [3:1]
push [1:1]
push 5
push .printSquare
call
push 1
push 0
st
push 1
push [0:1]
sub
push 0
push 1
st
push 1
push [1:1]
sub
push 1
push 1
st
push 2
push [2:1]
sub
push 2
push 1
st
push 2
push [3:1]
sub
```

```
push 3
push 1
st
push 30
delay
push 1
push [0:0]
add
push 0
push 0
st
push #PC-54
jmp
cframe
push #PC-96
jmp
cframe
halt

.randomColour
push 16777215
irnd
ret

.printSquare
push 1
alloc
push 0
push .randomColour
call
push 5
push 0
st
push [5:0]
push [4:0]
push 1
push [3:0]
push [3:0]
pixelr
push [5:0]
push [4:0]
push 1
push [3:0]
push [2:0]
```

```
pixelr
push [5:0]
push 1
push [1:0]
push [3:0]
push [3:0]
pixelr
push [5:0]
push 1
push [1:0]
push [0:0]
push [3:0]
pixelr
push 0
ret
```

Program 4 - SMPTE Bars

This program prints an image reminiscent of the old SMPTE bars that old televisions would show. The top coloured bars shift from left to right, and when a colour reaches the end it wraps around back to the front, in a conveyer belt-like pattern. For the best and most accurate illustration of this program, the resolution of the PixArDis VM display should be set to *(105, 90)*.

PixArLang Code

```
__pixelr __height, __width, 0, 0, #000000;

let startPoint:int = 0;
let bHeight:int = 15;
let bWidth:int = 18;

let counter:int = 0;
let x:int = 0;
while(x < __width){

    if (counter == 1){
        __pixelr bHeight, bWidth, startPoint, x, #dfe0eb;
    }else{
        __pixelr bHeight, bWidth, startPoint, x, #1a1818;
        if (counter == 6){
            counter = -1;
        }
    }
    x = x + bWidth;
    counter = counter + 1;
}

startPoint = startPoint + bHeight;
let mHeight:int = 5;
let mWidth:int = 15;

counter = 0;
let y:int = 0;
while(y < __width){

    if (counter == 0){
        __pixelr mHeight, mWidth, startPoint, y, #1020b3;
    }

    if (counter == 1){
```

```
        __pixelr mHeight, mWidth, startPoint, y, #1a1818;
    }

    if (counter == 2){
        __pixelr mHeight, mWidth, startPoint, y, #d140d6;
    }

    if (counter == 3){
        __pixelr mHeight, mWidth, startPoint, y, #1a1818;
    }

    if (counter == 4){
        __pixelr mHeight, mWidth, startPoint, y, #72e6f7;
    }

    if (counter == 5){
        __pixelr mHeight, mWidth, startPoint, y, #1a1818;
    }

    if (counter == 6){
        __pixelr mHeight, mWidth, startPoint, y, #dfe0eb;
        counter = -1;
    }

    y = y + mWidth;
    counter = counter + 1;
}

startPoint = startPoint + mHeight;
let tHeight:int = __height;
let tWidth:int = 15;

counter = 0;
let z:int = 0;
while(z < __width){

    if (counter == 0){
        __pixelr tHeight, tWidth, startPoint, z, #dfe0eb;
    }

    if (counter == 1){
        __pixelr tHeight, tWidth, startPoint, z, #eff21f;
    }
```

```
    if (counter == 2){
        __pixelr tHeight, tWidth, startPoint, z, #8afbff;
    }

    if (counter == 3){
        __pixelr tHeight, tWidth, startPoint, z, #28d431;
    }

    if (counter == 4){
        __pixelr tHeight, tWidth, startPoint, z, #ab17b3;
    }

    if (counter == 5){
        __pixelr tHeight, tWidth, startPoint, z, #e61e24;
    }

    if (counter == 6){
        __pixelr tHeight, tWidth, startPoint, z, #0a178c;
        counter = -1;
    }

    z = z + tWidth;
    counter = counter + 1;
}

counter = 6;
while(true){

    if (counter < 0){
        counter = 6;
    }

    let a:int = 0;
    while(a < __width){

        if (counter == 0){
            __pixelr tHeight, tWidth, startPoint, a, #dfe0eb;
        }

        if (counter == 1){
            __pixelr tHeight, tWidth, startPoint, a, #eff21f;
        }

        if (counter == 2){
```

```
            __pixelr tHeight, tWidth, startPoint, a, #8afbff;
        }

        if (counter == 3){
            __pixelr tHeight, tWidth, startPoint, a, #28d431;
        }

        if (counter == 4){
            __pixelr tHeight, tWidth, startPoint, a, #ab17b3;
        }

        if (counter == 5){
            __pixelr tHeight, tWidth, startPoint, a, #e61e24;
        }

        if (counter == 6){
            __pixelr tHeight, tWidth, startPoint, a, #0a178c;
            counter = -1;
        }

        a = a + tWidth;
        counter = counter + 1;
    }

    counter = counter - 1;
    __delay 250;
}
```

PixIR Assembly Code

```
.main
push #000000
height
width
push 0
push 0
pixelr
push 1
oframe
push 0
push 0
push 0
st
push 1
alloc
```

```
push 15
push 1
push 0
st
push 1
alloc
push 18
push 2
push 0
st
push 1
alloc
push 0
push 3
push 0
st
push 1
alloc
push 0
push 4
push 0
st
oframe
push #PC+7
width
push [4:1]
lt
cjmp
push #PC+50
jmp
oframe
push #PC+26
push 1
push [3:2]
eq
cjmp
push #1a1818
push [1:2]
push [2:2]
push [0:2]
push [4:2]
pixelr
oframe
push #PC+7
```

```
push 6
push [3:3]
eq
cjmp
push #PC+6
jmp
push -1
push 3
push 3
st
cframe
push #PC+8
jmp
push #dfe0eb
push [1:2]
push [2:2]
push [0:2]
push [4:2]
pixelr
cframe
push [2:1]
push [4:1]
add
push 4
push 1
st
push 1
push [3:1]
add
push 3
push 1
st
push #PC-53
jmp
cframe
push [1:0]
push [0:0]
add
push 0
push 0
st
push 1
alloc
push 5
```

```
push 5
push 0
st
push 1
alloc
push 15
push 6
push 0
st
push 0
push 3
push 0
st
push 1
alloc
push 0
push 7
push 0
st
oframe
push #PC+7
width
push [7:1]
lt
cjmp
push #PC+125
jmp
oframe
push #PC+7
push 0
push [3:2]
eq
cjmp
push #PC+8
jmp
push #1020b3
push [5:2]
push [6:2]
push [0:2]
push [7:2]
pixelr
cframe
oframe
push #PC+7
```

```
push 1
push [3:2]
eq
cjmp
push #PC+8
jmp
push #1a1818
push [5:2]
push [6:2]
push [0:2]
push [7:2]
pixelr
cframe
oframe
push #PC+7
push 2
push [3:2]
eq
cjmp
push #PC+8
jmp
push #d140d6
push [5:2]
push [6:2]
push [0:2]
push [7:2]
pixelr
cframe
oframe
push #PC+7
push 3
push [3:2]
eq
cjmp
push #PC+8
jmp
push #1a1818
push [5:2]
push [6:2]
push [0:2]
push [7:2]
pixelr
cframe
oframe
```

```
push #PC+7
push 4
push [3:2]
eq
cjmp
push #PC+8
jmp
push #72e6f7
push [5:2]
push [6:2]
push [0:2]
push [7:2]
pixelr
cframe
oframe
push #PC+7
push 5
push [3:2]
eq
cjmp
push #PC+8
jmp
push #1a1818
push [5:2]
push [6:2]
push [0:2]
push [7:2]
pixelr
cframe
oframe
push #PC+7
push 6
push [3:2]
eq
cjmp
push #PC+12
jmp
push #dfe0eb
push [5:2]
push [6:2]
push [0:2]
push [7:2]
pixelr
push -1
```

```
push 3
push 2
st
cframe
push [6:1]
push [7:1]
add
push 7
push 1
st
push 1
push [3:1]
add
push 3
push 1
st
push #PC-128
jmp
cframe
push [5:0]
push [0:0]
add
push 0
push 0
st
push 1
alloc
height
push 8
push 0
st
push 1
alloc
push 15
push 9
push 0
st
push 0
push 3
push 0
st
push 1
alloc
push 0
```

```
push 10
push 0
st
oframe
push #PC+7
width
push [10:1]
lt
cjmp
push #PC+125
jmp
oframe
push #PC+7
push 0
push [3:2]
eq
cjmp
push #PC+8
jmp
push #dfe0eb
push [8:2]
push [9:2]
push [0:2]
push [10:2]
pixelr
cframe
oframe
push #PC+7
push 1
push [3:2]
eq
cjmp
push #PC+8
jmp
push #eff21f
push [8:2]
push [9:2]
push [0:2]
push [10:2]
pixelr
cframe
oframe
push #PC+7
push 2
```

```
push [3:2]
eq
cjmp
push #PC+8
jmp
push #8afbff
push [8:2]
push [9:2]
push [0:2]
push [10:2]
pixelr
cframe
oframe
push #PC+7
push 3
push [3:2]
eq
cjmp
push #PC+8
jmp
push #28d431
push [8:2]
push [9:2]
push [0:2]
push [10:2]
pixelr
cframe
oframe
push #PC+7
push 4
push [3:2]
eq
cjmp
push #PC+8
jmp
push #ab17b3
push [8:2]
push [9:2]
push [0:2]
push [10:2]
pixelr
cframe
oframe
push #PC+7
```

```
push 5
push [3:2]
eq
cjmp
push #PC+8
jmp
push #e61e24
push [8:2]
push [9:2]
push [0:2]
push [10:2]
pixelr
cframe
oframe
push #PC+7
push 6
push [3:2]
eq
cjmp
push #PC+12
jmp
push #0a178c
push [8:2]
push [9:2]
push [0:2]
push [10:2]
pixelr
push -1
push 3
push 2
st
cframe
push [9:1]
push [10:1]
add
push 10
push 1
st
push 1
push [3:1]
add
push 3
push 1
st
```

```
push #PC-128
jmp
cframe
push 6
push 3
push 0
st
oframe
push #PC+5
push 1
cjmp
push #PC+163
jmp
oframe
push #PC+7
push 0
push [3:2]
lt
cjmp
push #PC+6
jmp
push 6
push 3
push 2
st
cframe
push 1
alloc
push 0
push 0
push 0
st
oframe
push #PC+7
width
push [0:1]
lt
cjmp
push #PC+125
jmp
oframe
push #PC+7
push 0
push [3:3]
```

```
eq
cjmp
push #PC+8
jmp
push #dfe0eb
push [8:3]
push [9:3]
push [0:3]
push [0:2]
pixelr
cframe
oframe
push #PC+7
push 1
push [3:3]
eq
cjmp
push #PC+8
jmp
push #eff21f
push [8:3]
push [9:3]
push [0:3]
push [0:2]
pixelr
cframe
oframe
push #PC+7
push 2
push [3:3]
eq
cjmp
push #PC+8
jmp
push #8afbff
push [8:3]
push [9:3]
push [0:3]
push [0:2]
pixelr
cframe
oframe
push #PC+7
push 3
```

```
push [3:3]
eq
cjmp
push #PC+8
jmp
push #28d431
push [8:3]
push [9:3]
push [0:3]
push [0:2]
pixelr
cframe
oframe
push #PC+7
push 4
push [3:3]
eq
cjmp
push #PC+8
jmp
push #ab17b3
push [8:3]
push [9:3]
push [0:3]
push [0:2]
pixelr
cframe
oframe
push #PC+7
push 5
push [3:3]
eq
cjmp
push #PC+8
jmp
push #e61e24
push [8:3]
push [9:3]
push [0:3]
push [0:2]
pixelr
cframe
oframe
push #PC+7
```

```
push 6
push [3:3]
eq
cjmp
push #PC+12
jmp
push #0a178c
push [8:3]
push [9:3]
push [0:3]
push [0:2]
pixelr
push -1
push 3
push 3
st
cframe
push [9:2]
push [0:1]
add
push 0
push 1
st
push 1
push [3:2]
add
push 3
push 2
st
push #PC-128
jmp
cframe
push 1
push [3:1]
sub
push 3
push 1
st
push 250
delay
push #PC-164
jmp
cframe
halt
```

Plagiarism Form

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

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 / 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 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.

* 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).

David Cachia Enriquez
Student Name


Signature

Student Name

Signature

Student Name

Signature

Student Name

Signature

CPS 2000
Course Code

Course Assignment 2022/2023
Title of work submitted

30/05/2023
Date