CPS2000 - Compiler Theory and Practice
Course Assignment 2022/2023

Department of Computer Science, University of Malta
Sandro Spina

April 4, 2023

Instructions

- This is **an individual** assignment and carries **100%** of the final **CPS2000** grade.
- The submission deadline is **31st May 2023**. A soft-copy of the report and all related files (including code) must be uploaded to the VLE by midnight of the indicated deadline. A link to the presentation video should also be included in the uploaded report. Hard copies **are not** required to be handed in. Source and executable files must be archived into a single .zip file before uploading to the VLE. It is the student's responsibility to ensure that the uploaded .zip file is valid. The PDF report must be submitted separately through the Turnitin submission system on the VLE.
- A report describing how you designed and implemented the different tasks of the assignment **is required**. Tasks (1-5) for which no such information is provided in the report will **not be** assessed and therefore, no marks assigned.
- In addition to the report, a video presentation showcasing your implementation using a number of example programs **is required**. You are encouraged to include a voice over to describe the examples being demonstrated. The maximum length of your video is 5 minutes and the maximum resolution 720p. If you are not able to upload the video onto the VLE (due to size) please upload to your Google cloud storage and include a hyperlink in the (.pdf) report.
- You are welcome to share ideas and suggestions. However, under no circumstances should code be shared among students. Please remember that plagiarism will not be tolerated; the final submission must be entirely your work.
- Your implementation can be carried out in Java, C++, C# or Python.
- You are to allocate approximately **40** hours for this assignment.

Description

You have just been employed with the startup company *PixArDis* that specialises in the development of programmable pixel art displays. The company has just released their first display (model PAD2000) to the market and has so far garnered several mixed reviews. Positive reviews highlighted the excellent hardware capabilities of the device which uses the latest energy-saving LED technology. The negative reviews, whilst highlighting the feature-rich programmable controller, argued that the programming interface of the PAD2000 is far from ideal. *PixArDis* has been receiving several reports that the programs (written in the assembly-like language *PixIR*) for the device do not seem to be functioning as expected. In order to address these concerns, you have been assigned the task of developing a higher-level, more intuitive programming interface which clients can use to program their PAD2000. In the meantime, the R&D team at *PixArDis* has developed an online simulator (see Figure 3) mimicking the behaviour of the PAD2000 hardware which you can use whilst developing the new imperative-style programming language *PixArLang*.

In this assignment you are to develop a lexer, parser, semantic analyser and code generator targeting *PixIR* for the programming language - *PixArLang*. The assignment is composed of four major components. These are:

1. the design and implementation of a FSA-based table-driven lexer (scanner) and hand-crafted top-down LL(k) parser,
2. the implementation of visitor classes to perform XML generation, semantic analysis and code generation of the abstract syntax tree (AST) produced by the parser
3. a report detailing how you designed and implemented the different tasks using a range of examples to highlight the merits and limitations of your work, and
4. a video presentation demonstrating the compilation pipeline and execution of *PixIR* code on the simulator provided.

PixArLang is an expression-based strongly-typed programming language. *PixArLang* distinguishes between expressions and statements. The language has C-style comments, that is, `//...` for line comments and `/*...*/` for block comments. The language is case-sensitive and each function is expected to return a value. *PixArLang* has 4 primitive types: 'bool', 'colour', 'int' and 'float'. For the 'colour' type, the colours supported range between hexadecimal values 0x000000 (black) and 0xffffff (white). Binary operators, such as '+', require that the operands have matching types; the language does not perform any implicit/automatic typecast.

A *PixArLang* program is simply a sequence of 0 or more statements. A function declaration is itself a statement. It is declared using the 'fun' keyword and its arguments are type annotated. All functions return a value and the return type must be specified after an arrow ->. Note that functions do not necessarily have to be declared before they are used since semantic analysis and code generation are carried out on the AST created following a correct parse of the entire program input. An important aspect of a programming language are the different ways to modify flow control. *PixArLang* supports both branching (if-else) and looping (while, for). Neither *break* nor *continue* are supported to exit loops.

The following lists a syntactically and semantically correct *PixArLang* program:

```
/* Multi-line comment
 * This function takes two integers and return true if
 * the first argument is greater than the second.
 * Otherwise if returns false.
 */
fun XGreaterY(x:int, y:int) -> bool {
 let ans:bool = true;
 if (y > x) { ans = false; }
 return ans;
}


//Single-line comment
//Same functionality as function above but using less code
fun XGreaterY_2(x:int, y:int) -> bool {
 return x > y;
}


//Allocates memory space for 4 variables.
fun AverageOfTwo(x:int, y:int) -> float {
  let t0 = x + y;
  let t1 = t0 / 2;
  return t1;


/*
 * Same functionality as function above but using less code.
 * Note the use of the brackets in the expression following
 * the return statement. Allocates space for 2 variables.
 */
fun AverageOfTwo_2(x:int, y:int) -> float {
  return (x + y) / 2;
}


//Takes two integers and returns the max of the two.
fun Max(x:int, y:int) -> int {
  let m:int = x;
  if (y > m) { m = y; }
  return m;
}
```

```
/*
 * This function takes two colours (players) and a max score.
 * A while loop is used to iteratively draw random numbers for the two
 * players and advance (along the y-axis) the player that gets the
 * highest score. Returns the winner (either 1 or 2) when max score is
 * reached by any of the players.
 */

fun Race(p1_c:colour, p2_c:colour, score_max:int) -> int {
  let p1_score:int = 0;
  let p2_score:int = 0;
  //while (Max(p1_score, p2_score) < score_max) //Alternative loop
  while ((p1_score < score_max) AND (p2_score < score_max))
  {
    let p1_toss:int = __randi 1000;
    let p2_toss:int = __randi 1000;
    if (p1_toss > p2_toss) {
        p1_score = p1_score + 1;
    } else {
        p2_score = p2_score + 1;
    }
    __pixel 1,p1_score,p1_c;
    __pixel 2,p2_score,p2_c;
  }
  let winner:int = 1;
  if (p2_score > p1_score) winner = 2;
  return winner;
}

//Execution (program entry point) starts at the first statement
//that is not a function declaration. This should go in the .main
//function of PixIR.

let c1:colour = #00ff00;        //green
let c2:colour = #0000ff;        //blue
let m:int = __height;           //the height (y-values) of the pad
let w:int = Race(c1, c2, m);    //call function Race
__print w;                      //prints value of expression to VM logs
```

The *PixArLang* programming language

The following rules describe the syntax of *TinyLang* in EBNF. Each rule has three parts: a left hand side (LHS), a right-hand side (RHS) and the ‘::=’ symbol separating these two sides. The LHS names the EBNF rule whereas the RHS provides a description of this name. Note that the RHS uses four control forms namely sequence, choice, option and repetition. In a sequence order is important and items appear left-to-right. The stroke symbol (. . . | . . .) is used to denote choice between alternatives. One item is chosen from this list; order is not important. Optional items are enclosed in square brackets ([. . .]) indicating that the item can either be included or discarded. Repeatable items are enclosed in curly brackets ({ . . . }); the items within can be repeated **zero** or more times. For example, a *Block* consists of zero or more *Statement* enclosed in curly brackets.

⟨*Letter*⟩	::=	[A-Za-z]
⟨*Digit*⟩	::=	[0-9]
⟨*Hex*⟩	::=	[A-Fa-f] | ⟨*Digit*⟩
⟨*Type*⟩	::=	‘`float`’ | ‘`int`’ | ‘`bool`’ | ‘`colour`’
⟨*BooleanLiteral*⟩	::=	‘`true`’ | ‘`false`’
⟨*IntegerLiteral*⟩	::=	⟨*Digit*⟩ { ⟨*Digit*⟩ }
⟨*FloatLiteral*⟩	::=	⟨*Digit*⟩ { ⟨*Digit*⟩ } ‘`.`’ ⟨*Digit*⟩ { ⟨*Digit*⟩ }
⟨*ColourLiteral*⟩	::=	‘`#`’ ⟨*Hex*⟩ ⟨*Hex*⟩ ⟨*Hex*⟩ ⟨*Hex*⟩ ⟨*Hex*⟩ ⟨*Hex*⟩
⟨*PadWidth*⟩	:: =	‘`__width`’
⟨*PadHeight*⟩	:: =	‘`__height`’
⟨*PadRead*⟩	:: =	‘`__read`’ ⟨*Expr*⟩‘`,`’⟨*Expr*⟩
⟨*PadRandI*⟩	:: =	‘`__randi`’ ⟨*Expr*⟩
⟨*Literal*⟩	::=	⟨*BooleanLiteral*⟩
	|	⟨*IntegerLiteral*⟩
	|	⟨*FloatLiteral*⟩
	|	⟨*ColourLiteral*⟩
	|	⟨*PadWidth*⟩
	|	⟨*PadHeight*⟩
	|	⟨*PadRead*⟩
⟨*Identifier*⟩	::=	(⟨*Letter*⟩) { ‘`_`’ | ⟨*Letter*⟩ | ⟨*Digit*⟩ }
⟨*MultiplicativeOp*⟩	::=	‘`*`’ | ‘`/`’ | ‘`and`’
⟨*AdditiveOp*⟩	::=	‘`+`’ | ‘`-`’ | ‘`or`’
⟨*RelationalOp*⟩	::=	‘`<`’ | ‘`>`’ | ‘`==`’ | ‘`!=`’ | ‘`<=`’ | ‘`>=`’
⟨*ActualParams*⟩	::=	⟨*Expr*⟩ { ‘`,`’ ⟨*Expr*⟩ }

```
⟨FunctionCall⟩     ::= ⟨Identifier⟩ ‘(’ [ ⟨ActualParams⟩ ] ‘)’

⟨SubExpr⟩          ::= ‘(’ ⟨Expr⟩ ‘)’

⟨Unary⟩            ::= ( ‘-’ | ‘not’ ) ⟨Expr⟩

⟨Factor⟩           ::= ⟨Literal⟩
                     |  ⟨Identifier⟩
                     |  ⟨FunctionCall⟩
                     |  ⟨SubExpr⟩
                     |  ⟨Unary⟩
                     |  ⟨PadRandI⟩
                     |  ⟨PadWidth⟩
                     |  ⟨PadHeight⟩
                     |  ⟨PadRead⟩

⟨Term⟩             ::= ⟨Factor⟩ { ⟨MultiplicativeOp⟩ ⟨Factor⟩ }

⟨SimpleExpr⟩       ::= ⟨Term⟩ { ⟨AdditiveOp⟩ ⟨Term⟩ }

⟨Expr⟩             ::= ⟨SimpleExpr⟩ { ⟨RelationalOp⟩ ⟨SimpleExpr⟩ }

⟨Assignment⟩       ::= ⟨Identifier⟩ ‘=’ ⟨Expr⟩

⟨VariableDecl⟩     ::= ‘let’ ⟨Identifier⟩ ‘:’ ⟨Type⟩ ‘=’ ⟨Expr⟩

⟨PrintStatement⟩   ::= ‘__print’ ⟨Expr⟩

⟨DelayStatement⟩   ::= ‘__delay’ ⟨Expr⟩

⟨PixelStatement⟩   ::= ‘__pixelr’ ⟨Expr⟩‘,’⟨Expr⟩‘,’⟨Expr⟩‘,’⟨Expr⟩‘,’⟨Expr⟩
                     |  ‘__pixel’ ⟨Expr⟩‘,’⟨Expr⟩‘,’⟨Expr⟩

⟨RtrnStatement⟩    ::= ‘return’ ⟨Expr⟩

⟨IfStatement⟩      ::= ‘if’ ‘(’ ⟨Expr⟩ ‘)’ ⟨Block⟩ [ ‘else’ ⟨Block⟩ ]

⟨ForStatement⟩     ::= ‘for’ ‘(’ [ ⟨VariableDecl⟩ ] ’;’ ⟨Expr⟩ ’;’ [ ⟨Assignment⟩ ] ‘)’ ⟨Block⟩

⟨WhileStatement⟩   ::= ‘while’ ‘(’ ⟨Expr⟩ ‘)’ ⟨Block⟩

⟨FormalParam⟩      ::= ⟨Identifier⟩ ‘:’ ⟨Type⟩

⟨FormalParams⟩     ::= ⟨FormalParam⟩ { ‘,’ ⟨FormalParam⟩ }

⟨FunctionDecl⟩     ::= ‘fun’ ⟨Identifier⟩ ‘(’ [ ⟨FormalParams⟩ ] ‘)’ ‘->’ ⟨Type⟩ ⟨Block⟩

⟨Statement⟩        ::= ⟨VariableDecl⟩ ‘;’
                     |  ⟨Assignment⟩ ‘;’
                     |  ⟨PrintStatement⟩ ‘;’
                     |  ⟨DelayStatement⟩ ‘;’
                     |  ⟨PixelStatement⟩ ‘;’
                     |  ⟨IfStatement⟩
                     |  ⟨ForStatement⟩
                     |  ⟨WhileStatement⟩
```

| $\langle$*RtrnStatement*$\rangle$ ';'
| $\langle$*FunctionDecl*$\rangle$
| $\langle$*Block*$\rangle$

$\langle$*Block*$\rangle$ ::= '{' { $\langle$*Statement*$\rangle$ } '}'

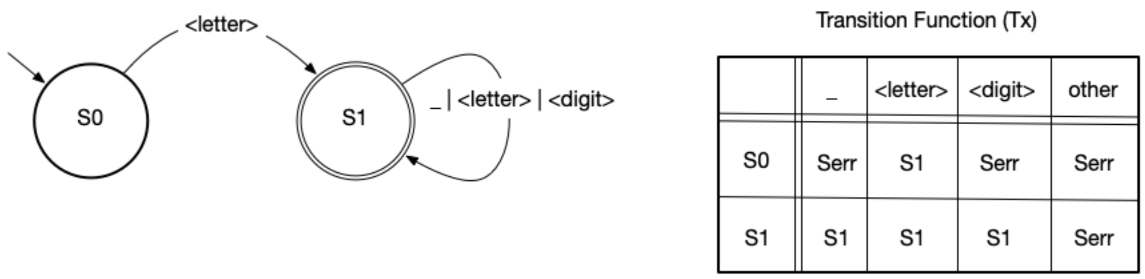
$\langle$*Program*$\rangle$ ::= { $\langle$*Statement*$\rangle$ }

Task Breakdown

Task 1 - Table-driven lexer

In this first task you are to develop the lexer for the *PixArLang* language. The lexer is to be implemented using the table-driven approach which simulates the DFA transition function of the *PixArLang* micro-syntax. The lexer should be able to report any lexical errors in the input program. Note that you should first determine the micro-syntax for the language, i.e. identify the tokens which will be required by the parser. The EBNF for the language should give you clear indications of the selection of tokens. The transition table (DFA encoding) can be hard-coded as a 2D array into the scanner source directly. Alternatively it can be written to a file and loaded by the lexer into an appropriate data structure upon initialisation. Figure 1 illustrates the DFA and transition function for the identifier token.

[Marks: 15%]



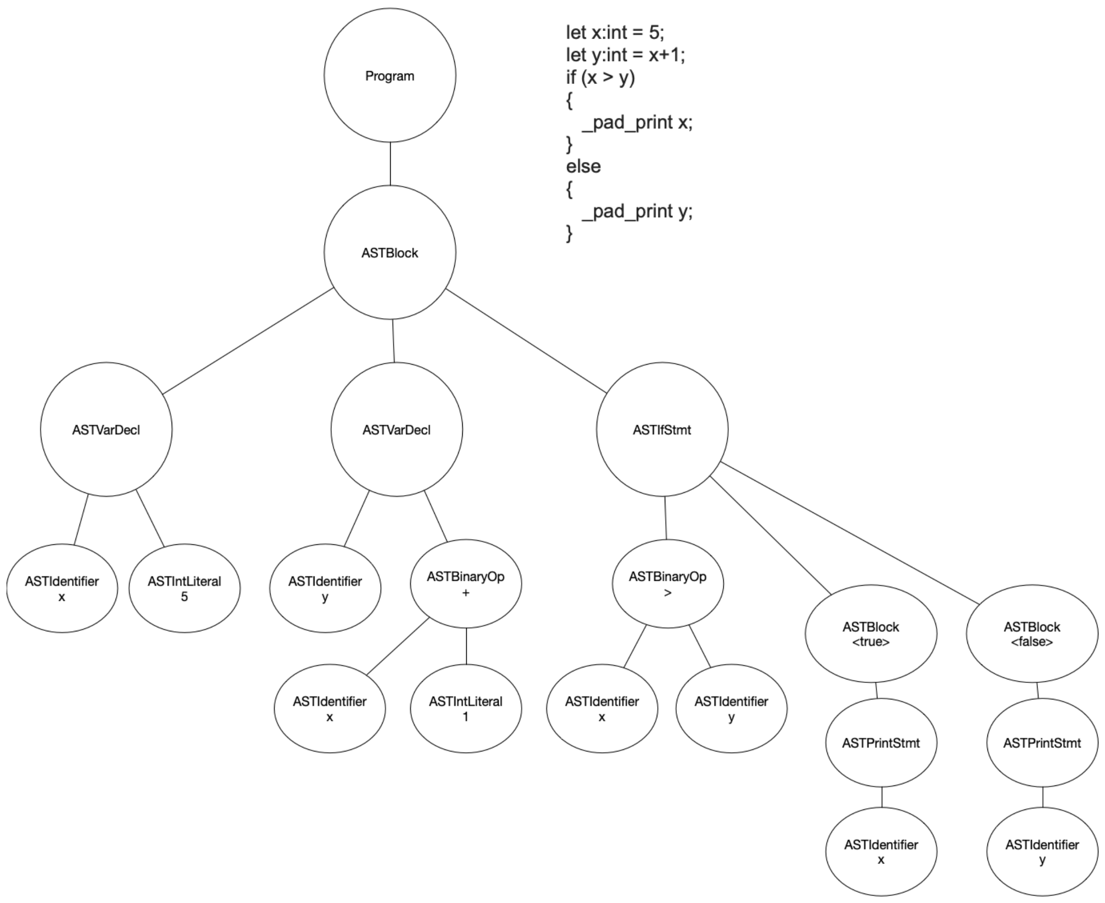
	_	<letter>	<digit>	other
S0	Serr	S1	Serr	Serr
S1	S1	S1	S1	Serr

Figure 1: An example DFA for the *PixArLang* identifier token.

Task 2 - Hand-crafted LL(k) parser

In this task you are to develop a *hand-crafted top-down LL(k) parser* for the *PixArLang* language. The Lexer and Parser classes interact through the function *GetNextToken*() which the parser uses to get the next valid token from the lexer. Note that for the vast majority of cases, the parser only needs to read one symbol of lookahead (k=1) in order to determine which production rule to use. The parser should be able to report any syntax errors in the input program. A successful parse of the input should produce an abstract syntax tree (AST) describing the structure of the program. You are free to decide on the form of the AST. Figure 2 illustrates an example AST generated for a small *PixArLang* program. Note that the *ASTBlock* node should, strictly speaking, not be included in the AST since no { and } are included in the code fragment. Should the code be placed in the curly brackets then the *ASTBlock* node has to be present to indicate that its child nodes are evaluated in the same scope.

[Marks: 30%]



```
let x:int = 5;
let y:int = x+1;
if (x > y)
{
   _pad_print x;
}
else
{
   _pad_print y;
}
```

Figure 2: An example AST for a small *PixArLang* program.

Task 3 - AST XML Generation Pass

In OOP programming, the Visitor design pattern is used to describe an operation to be performed on the elements of an object structure without changing the classes on which it operates. In our case this object structure is the AST (**not** the parse tree) produced by the parser in Task 2. For this task you are to implement a visitor class to output a properly indented XML representation of the generated AST. Please note that additional notes - with examples - on the Visitor design pattern are available on the VLE. Check those out before starting this task. The following code listing illustrates an example XML output for the declaration of variable ‘x’ of type ‘float’. In this case, the root node of the AST is a $< Decl >$ node with children representing the variable being declared (x) and the expression (3.142 * 20.0754 + 5) it is being assigned to. Note that the XML tags can be whatever you decide upon; for instance instead of $< Decl >$, $< ASTDeclNode >$ could be used.

Also note that the expression on the right hand side is composed of two binary expression nodes. Recall that the AST should reflect operator precedence as specified in the grammar. Since this is an XML representation of the AST there's no need to include non-terminals such as $< Factor >$ and $< Term >$.

[Marks: 5%]

```
let x : float = 3.142 * 20.0765 + 5;

<Decl>
  <Var Type="float">x</Id>
  <BinExprNode Op="+">
     <BinExprNode Op="*">
        <FloatConst>3.142</FloatConst>
        <FloatConst>20.0765</FloatConst>
      </BinExprNode>
      <FloatConst>5</FloatConst>
  </BinExprNode>
</Decl>
```

Task 4 - Semantic Analysis Pass

For this task, you are to implement another visitor class to traverse the AST and perform type-checking (e.g. checking that variables are assigned to appropriately typed expressions, variables are not declared multiple times in the same scope, variables are declared before being used, etc.). Scopes are created whenever a block is entered and destroyed when control leaves the block. The program's execution entry point (let x: int = 45; in example below) creates the first scope. Note that blocks may be nested and that to carry out this task, it is essential to have a proper implementation of a symbol table. Your compiler should be able to report any semantic analysis errors resulting from the traversal of the AST. Note that whereas *PixArLang* syntax allows for function definitions within local scope, for this task, function definitions need only be declared within the first scope. An important check carried out by the semantic analysis visitor is that of ensuring that a function always returns a value and that this value is compatible with the function signature. This can be slightly more challenging when multiple return statements are present in the body of a function.

This pass also needs to ensure that the arguments passed on to the specialised instructions of PAD2000 are compatible with what the hardware is expecting. For instance in the case of *__pixel* the first four expressions are of type int and the last one of type colour. Similarly for the rest of the instructions which will be described in Task 5.

[Marks: 20%]

```
fun MoreThan50(x: int) -> bool {
   let x:int = 23; //syntax ok, but this should not be allowed!!
   if (x <= 50) {
     return false;
   }
   return true;
}

let x: int = 45; //this is fine
while (x < 50) {
  print MoreThan50(x);  //"false" x6 since bool operator is <
  x = x + 1;
}

let x: int = 45; //re-declaration in the same scope ... not allowed!!
while (MoreThan50(x)) {
  print MoreThan50(x);  //"false" x5 since bool operator is <=
  x = x + 1;
}

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

for (let u:int = 0; u<w; u = u+1)
{
  for (let v:int = 0; v<h; v = v+1)
   {
     //set the pixel at u,v to the colour green
     __pixelr u,v,1,1,#00ff00;
     //or ... assume one pixel 1x1
     //__pixel u,v,#00ff00;
   }
}
```

Task 5 - PixIR Code Generation Pass

For this task, you are to implement another visitor class to traverse the AST and generate PixIR code for the PAD2000. The hardware operates using two stacks to store operands used by the instructions referred to as the operand stack (**OpS**) and another stack to store address locations used by the *call* instruction and referred to as the address stack (**AdS**). Memory (**M**) is modelled as a stack of frames (**SoF**) where the top of the stack represents the current scope. A stack frame (**F**) is pushed onto M whenever a new scope is entered and popped when exited. Local variables within the scope are stored in an array and accessed using the instruction *st* to store a value into memory and *push*$[i : l]$ to read a value from memory. A program (**P**) consists of a sequences of instructions (strictly one per line) organised in function blocks each starting with the line *.name* where *name* indicates the name of the function. The program counter points to the current instruction (index in P) being executed. The execution entry point is the first instruction in the *.main* function block. If this is not present the program cannot be executed and the VM reports an error. The following describes the PixIR instruction set.

Instruction	Effect	Description	Pre-Cond.
push x	a → ax	Pushes the value x onto OpS. If x is a colour (e.g. #00ff00) it is first encoded as an integer.	x is a value
push .n	a → ax	Pushes the instruction index/address (x) of function n onto OpS. Does nothing if the name n does not exist.	n is a valid function name
push #PC±y	a → xa	Pushes the current instruction index plus or minus offset y onto OpS.	0 < x < #P
push [i:l]	a → ax	Pushes the value x, stored in SoF at index i of the frame at stack level l. If memory location does not exist, an undefined value may be pushed to OpS.	l < #SoF
st	dcba → d	Pops values a, b and c from OpS and stores value d at frame index c, level b of the SoF.	NA
nop	a → a	No operation. No changes to VM state.	NA
drop	ba → b	Removes the value at the top of OpS.	#OpS ≥ 1
dup	ba → bxx	Pops a from OpS and pushes the value x = a twice to OpS.	#OpS ≥ 1
add	ba → x	Pops a and b from OpS and pushes back the value x = a+b.	#OpS ≥ 2
sub	ba → x	Pops a and b from OpS and pushes back the value x = a-b.	#OpS ≥ 2
mul	ba → x	Pops a and b from OpS and pushes back the value x = a*b.	#OpS ≥ 2
div	ba → x	Pops a and b from OpS and pushes back the value x = a/b.	#OpS ≥ 2
inc	a → x	Pops a from OpS and pushes back the value x = a+1.	#OpS ≥ 1
dec	a → x	Pops a from OpS and pushes back the value x = a-1.	#OpS ≥ 1
max	ba → x	Pops a and b from OpS and pushes back the value x = max(a,b).	#OpS ≥ 2
min	ba → x	Pops a and b from OpS and pushes back the value x = min(a,b).	#OpS ≥ 2
irnd	a → x	Pops a from OpS and pushes back the integer value x = random(0,a).	#OpS ≥ 1

lt	ba $\rightarrow$ x	Pops a and b from OpS and pushes back the value x = a<b ? 1 : 0.	#OpS $\geq$ 2
le	ba $\rightarrow$ x	Pops a and b from OpS and pushes back the value x = a$\leq$b ? 1 : 0.	#OpS $\geq$ 2
gt	ba $\rightarrow$ x	Pops a and b from OpS and pushes back the value x = a>b ? 1 : 0.	#OpS $\geq$ 2
ge	ba $\rightarrow$ x	Pops a and b from OpS and pushes back the value x = a$\geq$b ? 1 : 0.	#OpS $\geq$ 2
eq	ba $\rightarrow$ x	Pops a and b from OpS and pushes back the value x = a$\geq$b ? 1 : 0.	#OpS $\geq$ 2
jmp	ba $\rightarrow$ b	Pops a from OpS and jumps to instruction at location a (representing program line idx).	#OpS $\geq$ 1, 0 < a < #P
cjmp	cba $\rightarrow$ c	Pops a and b from OpS and jumps to instruction at location a (representing program line idx) if b$\neq$0.	#OpS $\geq$ 2, 0 < a < #P
call	..ba $\rightarrow$ .., u $\rightarrow$ ux	Pushes x = #PC to AdS. Creates and pushes frame F to M storing b values popped from OpS. Jumps to instruction at location a (representing instruction idx of called function).	#OpS $\geq$ b+2
ret	vu $\rightarrow$ v	Pops u from AdS and jumps to instruction at location u (representing program line idx).	#AdS $\geq$ 1
halt	a $\rightarrow$ a	Halts the execution of the program. Should be present in the .main function.	NA
oframe	ba $\rightarrow$ a	Pops a from OpS and allocates a new frame with space for a variables. Pushes the new frame onto the memory stack.	#OpS $\geq$ 1
cframe	a $\rightarrow$ a	Pops the frame at the top of the memory stack SoF.	#SoF $\geq$ 1
alloc	ba $\rightarrow$ b	Pops a from OpS and allocates a additional variable locations in the current scope.	#SoF $\geq$ 1

delay	ba → b	Pops a from the OpS and pauses the execution of the program for a milliseconds.	#OpS ≥ 1
pixel	dcba → d	Pops a,b,c values from OpS and changes the colour of pixel at location a,b to colour c.	#OpS ≥ 5
pixelr	fedcba → f	Pops a,b,c,d,e values from OpS and changes the colour of pixel region at location a,b, and width c and height d to colour e. If c and d are 1, only one pixel is changed like pixel above.	#OpS ≥ 5
clear	ba → b	Pops a from the OpS and clears all pixels to colour a.	#OpS ≥ 1
width	a → ax	Pushes onto OpS the value x = width of the PAD2000 display.	NA
height	a → ax	Pushes onto OpS the value x = height of the PAD2000 display.	NA
print	ba → b	Pops a from OpS and prints a to the logs section of the simulator. Value is always printed irrespective of the log level set.	#OpS ≥ 1

The symbol table now needs to also store additional details about variable memory locations (index in frame and level in SoF) in addition to type as was done with Task 4. This information is required when using the *push*[*i* : *l*] and *st* instructions. Recall that before st is scheduled, the value to the stored, the frame index in SoF and the index within that frame have to be pushed to OpS. st consumes the three values. Implementation-wise, note that you should start this task by copy-pasting the semantic analysis visitor implemented for Task 4 and build on it. The following code listing shows the translation of the *PixArLang* code shown at the beginning of this document to *PixIR*.

[Marks: 30%]

```
.main               //entry point function
push 4              //4 local variables
oframe              //allocate space for variables (new frame)
push #ffffff        //clear colour is white
clear               //clear pixels to white
push #0000ff        //player 1 colour is blue
push 0              //index 0 in current scope
push 0              //level 0 in SoF
st                  //store colour (var c1)
push #00ff00        //player 2 colour is green
push 1              //index 1 in current scope
push 0              //level 0 in SoF
st                  //store colour (var c2)
height              //push display height
push 2              //index 2 in current scope
push 0              //level 0 in SoF
```

```
st                //store pad height (var m)
push [2]          //push value of m
push [1]          //push value of c2
push [0]          //push value of c1
push 3            //push number of parameters to race function
push .race        //push address of race function
call              //call function race
print             //print value returned by call to race (can use w)
halt              //halt program execution

.race             //entry point to function race
push 3            //three local variables (p1_score, p2_score, winner)
alloc             //add three locations to current scope
push 0            //initial score is 0
push 3            //frame index of p1_score
push 0            //current scope (level 0)
st                //store 0 to p1_score
push 0            //initial score is 0
push 4            //frame index of p2_score
push 0            //current scope (level 0)
st                //store 0 to p2_score
push 2            //two new locals in while loop
oframe            //create new frame - while scope
push [2:1]        //push scoreMax - start of while
push [3:1]        //push p1_score value
push [4:1]        //push p2_score value
push 2            //two parameters
push .max         //push inst. address to .max
call              //call max
lt                //is max < scoreMax
push 0            //compare to false (0)
eq                //if false jump
push #PC+43       //offset relative to current instruction idx
cjmp2             //exit while loop - inverts operands in stack
push 1000         //1000 is used as operand to irnd
irnd              //pop 1000 and push value between 0 and 1000
push 0            //frame index of p1_toss
push 0            //current scope (level 0)
st                //store p1_toss
push 1000         //1000 is used as operand to irnd
irnd              //pop 1000 and push value between 0 and 1000
push 1            //frame index of p2_toss
push 0            //current scope (level 0)
st                //store p2_toss
push [1:0]        //load p2_toss value
push [0:0]        //load p1_toss value
gt                //is p1_toss > p2_toss?
```

```
push 0          //if false jump to else
eq              //compare with false
push #PC+10.    //offset to else. Jmp here if false
cjmp2           //jump to else
push [3:1]      //load p1_score value
push 1          //operand to add
add             //p1_score + 1
push 3          //frame index for p1_score
push 1          //scope level 1 (not top)
st              //store updated p1_score
push #PC+8      //address offset to after else
jmp             //jump to after else
push [4:1]      //load p2_score value
push 1          //operand to add
add             //p2_score + 1
push 4          //frame index for p2_score
push 1          //scope level 1 (not top)
st              //store updated p2_score
push [0:1]      //colour of p1
push [3:1]      //score (y value on panel) of p1
push 0          //x value on panel
pixel           //draw p1 pixel
push [1:1]      //colour of p2
push [4:1]      //score (y value on panel) of p2
push 1          //x value on panel
pixel           //draw p2 pixel
push #PC-50     //address of instruction to eval. while expression
jmp             //go back to start of loop
cframe          //end of loop - pop frame
push 1          //push value of 1
push 5          //frame index for winner
push 0          //scope level 0 (top)
st              //store 1 to winner
push [3:0]      //push p1_score (note level 0 now)
push [4:0]      //push p2_score (note level 0 now)
gt              //p2score > p1score
push 0          //compare to 0 (false)
eq              //is expression false?
push #PC+6      //if yes, jump 6 instructions
cjmp2           //jump to else if eq=0 is 1
push 2          //push value of 2
push 5          //frame index for winner
push 0          //scope level 0 (top)
st              //store 2 to winner
push [5:0]      //push value of winner variable
ret             //return from function
```

```
.max                //entry point to function max
push #PC+10         //using cjmp not cjmp2; address to else part goes here
push [1]            //push second parameter
push [0]            //push first parameter
gt                  //first greater than second ?
push 0              //push 0 (false)
eq                  //is expression (if) equal to false
cjmp                //if false jump tp else part
push [0]            //0>1 therefore push parameter 0
push #PC+3          //address after else
jmp                 //jmp to after else
push [1]            //1>0 therefore push parameter 1
ret                 //return, max value is on operand stack
```

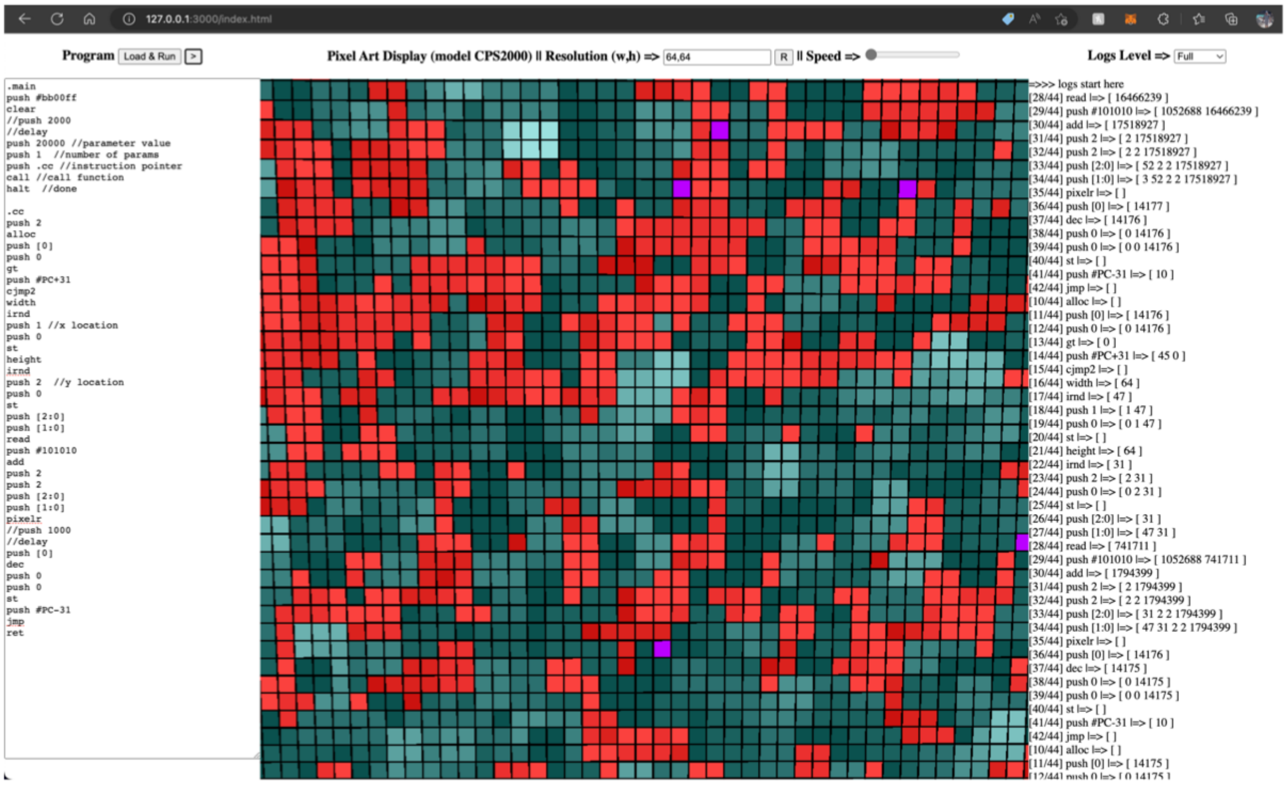
```
.main
push #bb00ff
clear
push 20000          //parameter value
push 1              //number of params
push .cc            //instruction pointer
call                //call function
halt                //done

.cc
push 2
alloc
push [0]
push 0
gt
push #PC+31
cjmp2
width
irnd
push 1              //x location
push 0
st
height
irnd
push 2              //y location
push 0
st
push [2:0]
push [1:0]
read                //pushes to OpS colour at location [1:0], [2:0]
push #101010
add                 //update current colour at that location
push 2
push 2
push [2:0]
push [1:0]
pixelr           //draw new colour
push [0]
dec
push 0
push 0
st
push #PC-31
jmp
ret
```

Online Simulator - PixArDis

The PAD2000 VM simulator is shown in Figure 3 and consists of three main components. On the left hand side, the user can write PixIR code (paste from generator) for execution. Clicking on the button 'Load & Run', loads the program into the VM and starts executing the program. Note that single line comments (following //) can be used following instructions. Instructions can themselves be commented in which case they will not be counted to determine instruction addresses. The 'play/pause' button can be used to play and pause the execution of the program when necessary. The central part of the VM simulator renders the pixel display hardware. The initial resolution is set to 36x36, however this can easily be changed by writing the desired w,h in the resolution textfield and clicking the button 'R'. Program execution is restarted. The right hand side (third column) is used to display debug logs which should help you understand how the instructions are changing the state of the VM. Note that disabling logs (level -none-) greatly improves the performance of the VM. Note that the stack based VM will be discussed in detail during lectures when intermediate representation (IR), assembly languages and runtime environments are discussed. The simulator is accessible at http://10.60.10.133:3000

Report

In addition to the source and class files, you are to write and submit a report and video presentation. Remember that tasks 1 to 5 for which no information is provided in the report will not be assessed. In your report include any deviations from the original EBNF, the salient points on how you developed the lexer / parser / code generator (and reasons behind any decisions you took) including semantic rules and code execution, and any sample *PixArLang* programs you developed for testing the outcome of your compiler. In your report, state what you are testing for, insert the program AST and the outcome of your test. Any PixIR programs can be included as separate text files in an examples folder which you'll be uploading to the VLE with the rest of the source code.


127.0.0.1:3000/index.html
Program
Load & Run
>
Pixel Art Display (model CPS2000) || Resolution (w,h) =>
R
|| Speed =>

Resolution (w,h) => 64,64

Logs Level => Full

```
.main
push #bb00ff
clear
//push 2000
//delay
push 20000 //parameter value
push 1  //number of params
push .cc //instruction pointer
call //call function
halt  //done

.cc
push 2
alloc
push [0]
push 0
gt
push #PC+31
cjmp2
width
irnd
push 1 //x location
push 0
st
height
irnd
push 2  //y location
push 0
st
push [2:0]
push [1:0]
read
push #101010
add
push 2
push 2
push [2:0]
push [1:0]
pixelr
//push 1000
//delay
push [0]
dec
push 0
push 0
st
push #PC-31
jmp
ret
```

```
=>>> logs start here
[28/44] read |=> [ 16466239 ]
[29/44] push #101010 |=> [ 1052688 16466239 ]
[30/44] add |=> [ 17518927 ]
[31/44] push 2 |=> [ 2 17518927 ]
[32/44] push 2 |=> [ 2 2 17518927 ]
[33/44] push [2:0] |=> [ 52 2 2 17518927 ]
[34/44] push [1:0] |=> [ 3 52 2 2 17518927 ]
[35/44] pixelr |=> [ ]
[36/44] push [0] |=> [ 14177 ]
[37/44] dec |=> [ 14176 ]
[38/44] push 0 |=> [ 0 14176 ]
[39/44] push 0 |=> [ 0 0 14176 ]
[40/44] st |=> [ ]
[41/44] push #PC-31 |=> [ 10 ]
[42/44] jmp |=> [ ]
[10/44] alloc |=> [ ]
[11/44] push [0] |=> [ 14176 ]
[12/44] push 0 |=> [ 0 14176 ]
[13/44] gt |=> [ 0 ]
[14/44] push #PC+31 |=> [ 45 0 ]
[15/44] cjmp2 |=> [ ]
[16/44] width |=> [ 64 ]
[17/44] irnd |=> [ 47 ]
[18/44] push 1 |=> [ 1 47 ]
[19/44] push 0 |=> [ 0 1 47 ]
[20/44] st |=> [ ]
[21/44] height |=> [ 64 ]
[22/44] irnd |=> [ 31 ]
[23/44] push 2 |=> [ 2 31 ]
[24/44] push 0 |=> [ 0 2 31 ]
[25/44] st |=> [ ]
[26/44] push [2:0] |=> [ 31 ]
[27/44] push [1:0] |=> [ 47 31 ]
[28/44] read |=> [ 741711 ]
[29/44] push #101010 |=> [ 1052688 741711 ]
[30/44] add |=> [ 1794399 ]
[31/44] push 2 |=> [ 2 1794399 ]
[32/44] push 2 |=> [ 2 2 1794399 ]
[33/44] push [2:0] |=> [ 31 2 2 1794399 ]
[34/44] push [1:0] |=> [ 47 31 2 2 1794399 ]
[35/44] pixelr |=> [ ]
[36/44] push [0] |=> [ 14176 ]
[37/44] dec |=> [ 14175 ]
[38/44] push 0 |=> [ 0 14175 ]
[39/44] push 0 |=> [ 0 0 14175 ]
[40/44] st |=> [ ]
[41/44] push #PC-31 |=> [ 10 ]
[42/44] jmp |=> [ ]
[10/44] alloc |=> [ ]
[11/44] push [0] |=> [ 14175 ]
[12/44] push 0 |=> [ 0 14175 ]
```

Figure 3: PixArDis PAD2000 online simulator