CPS2000 - Compiler Theory and Practice
Course Assignment 2023/2024

Department of Computer Science, University of Malta
Sandro Spina

April 29, 2024

Instructions

- This is **an individual** assignment and carries **100%** of the final **CPS2000** grade.
- The submission deadline is **31st May 2024**. 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 short 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 Rust, Java, C++, C# or Python.
- You are to allocate approximately **80** hours for this assignment.

Description

You have just been employed with the startup company PArDis that specialises in the development of programmable pixel art displays. The company has just released their third display (model PAD2000c) to the market and has so far garnered mixed reviews. Positive reviews highlighted the excellent hardware capabilities of the device which uses the latest energy-saving OLED technology. The negative reviews, whilst highlighting the feature-rich programmable controller, argued that the programming interface of the PAD2000c is far from ideal. PArDis customer care has been receiving several reports that the programs (written in the assembly-like language PArIR) 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 compiler for a high-level, more intuitive programming language (compiled to PArIR instructions) which clients can use to program their PAD2000c. In the meantime, the R&D team at PArDis has developed an online simulator (see Figure 3) mimicking closely the behaviour of the PAD2000c hardware which you can use whilst developing the compiler for the new imperative-style programming language PArL .

In this assignment you are to develop a lexer, parser, semantic analyser and code generator targeting PArIR for the programming language PArL . 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 PArL semantic analysis and PArIR code generation using 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 short video presentation demonstrating the compilation pipeline (PArL to PArIR) and the execution of code on the simulator provided.

PArL is an expression-based strongly-typed programming language. PArL 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. PArL 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. The ‘as’ operator can be used to cast between types. To make life easier for programmers, PArL also supports fixed-size arrays which can be declared and initialised in a number of ways.

A PArL 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 passes 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.

PArL 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 PArL program:

```
/* 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;
}

// 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 (x,y,t0,t1).
fun AverageOfTwo(x:int, y:int) -> float {
  let t0:int = x + y;
  let t1:float = t0 / 2 as float;   //casting expression to a float
  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 as float;
}

//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;
}

__write 10, 14, #00ff00;
__delay 100;
__write_box 10, 14, 2, 2, #0000ff;

for (let i:int = 0; i<10; i=i+1) {
  __print i;
  __delay 1000;
}
```

```
/* 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. Winner printed on console.
 */
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;
            __write 1, p1_score, p1_c;
        } else {
            p2_score = p2_score + 1;
            __write 2, p2_score, p2_c;
        }

        __delay 100;
    }

    if (p2_score > p1_score) {
        return 2;
    }

    return 1;
}
//Execution (program entry point) starts at the first statement
//that is not a function declaration. This should go in the .main
//function of ParIR.

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 PArL programming language

The following rules describe the syntax of PArL 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*⟩ :: = '`__random_int`' ⟨*Expr*⟩

⟨*Literal*⟩ ::= ⟨*BooleanLiteral*⟩
| ⟨*IntegerLiteral*⟩
| ⟨*FloatLiteral*⟩
| ⟨*ColourLiteral*⟩
| ⟨*PadWidth*⟩
| ⟨*PadHeight*⟩
| ⟨*PadRead*⟩

⟨*Identifier*⟩ ::= ⟨*Letter*⟩ { '`_`' | ⟨*Letter*⟩ | ⟨*Digit*⟩ } ['`[`' ⟨*Expr*⟩ '`]`']

⟨*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⟩ } [ ‘as’ ⟨Type⟩ ]

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

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

⟨VariableDeclSuffix⟩ ::= ‘=’ ⟨Expr⟩ | ‘[’ ⟨VarialeDeclArray⟩

⟨VariableDeclArray⟩  ::= ⟨IntegerLiteral⟩ ‘]’ ‘=’ ‘[’ ⟨Literal⟩ ‘]’
                        |  ‘]’ ‘=’ ‘[’ ⟨Literal⟩ { ‘,’ ⟨Literal⟩ } ‘]’

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

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

⟨WriteStatement⟩      ::= ‘__write_box’ ⟨Expr⟩‘,’⟨Expr⟩‘,’⟨Expr⟩‘,’⟨Expr⟩‘,’⟨Expr⟩
                        |  ‘__write’ ⟨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⟩ [ ‘[’ ⟨IntegerLiteral⟩ ‘]’ ]

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

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

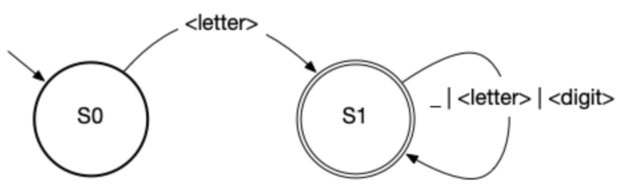
⟨*Statement*⟩	::=	⟨*VariableDecl*⟩ ‘;’
	|	⟨*Assignment*⟩ ‘;’
	|	⟨*PrintStatement*⟩ ‘;’
	|	⟨*DelayStatement*⟩ ‘;’
	|	⟨*WriteStatement*⟩ ‘;’
	|	⟨*IfStatement*⟩
	|	⟨*ForStatement*⟩
	|	⟨*WhileStatement*⟩
	|	⟨*RtrnStatement*⟩ ‘;’
	|	⟨*FunctionDecl*⟩
	|	⟨*Block*⟩
⟨*Block*⟩	::=	‘{’ { ⟨*Statement*⟩ } ‘}’
⟨*Program*⟩	::=	{ ⟨*Statement*⟩ }

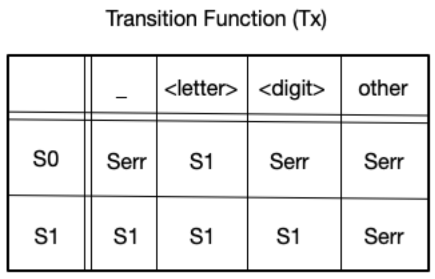
Task Breakdown

Task 1 - Table-driven lexer - No Arrays

In this first task you are to develop the lexer for the PArL language. The lexer is to be implemented using the table-driven approach which simulates the DFA transition function of the PArL 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: 20%]




Transition Function (Tx)

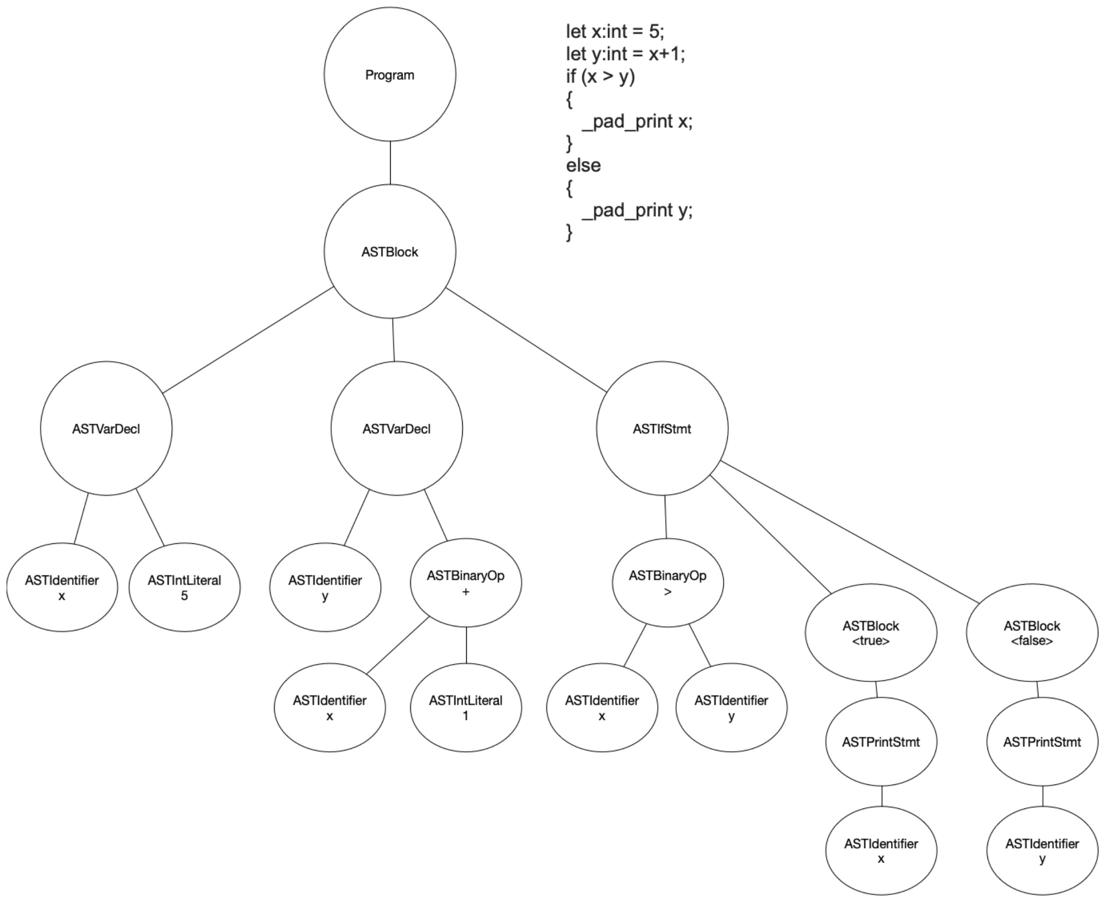
	_	<letter>	<digit>	other
S0	Serr	S1	Serr	Serr
S1	S1	S1	S1	Serr

Figure 1: An example DFA for the PArL identifier token. Note that in this example we are assuming the language does not support arrays (e.g. arr[20] or arr[i]). When implemented (for Task 5) the scanning of the array characters would follow after state S1.

Task 2 - Hand-crafted LL(k) parser - No Arrays

In this task you are to develop a *hand-crafted top-down LL(k) parser* for the PArL language. The Lexer and Parser classes interact through the function *GetNextToken*() which the parser uses to get the next valid token from the lexer. Alternatively you might want to generate all tokens and pass them directly (as a list) to the parser. 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 PArL program. Note that the *ASTBlock* node may, strictly speaking, not be included in the AST since no { and } are included in the code fragment. If you want you can assume that the top level statements are included in an implicit block. 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 - Semantic Analysis Pass - No Arrays

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 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 PArL 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 the *PAD2000c* are compatible with what the hardware is expecting. For instance in the case of __*write* the first two expressions following the keyword are of type int and the last one of type colour. Similarly, for the rest of the instructions which will be described in Task 4.

[Marks: 15%]

```
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
     __write_box u,v,1,1,#00ff00;
     //or ... assume one pixel 1x1
     //__write u,v,#00ff00;
   }
}
```

Task 4 - PArIR Code Generation Pass - No Arrays

For this task, you are to implement another visitor class to traverse the AST and generate PArIR code for the PAD2000c. 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 PArIR instruction set. Note that a number of additional PArIR instructions will be included in Task 5 which are associated with the use of arrays in PArL programs. You do not need to use them in this task.

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 the 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 a small PArL program into its equivalent PArIR instructions. More example translations are available on the VLE.

[Marks: 25%]

Instruction	Effect	Description
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.
push .n	a → ax	Pushes the instruction index/address (x) of function n onto OpS. Does nothing if the name n does not exist. Note that the VM keeps track of all functions in your program.
push #PC±y	a → ax	Pushes the current instruction index plus or minus offset y onto OpS.
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.
st	cba →	Pops values a, b and c from OpS and stores value c at frame index b, level a of the SoF.
nop	a → a	No operation. No changes to VM state.
drop	ba → b	Removes the value at the top of OpS.
dup	ba → baa	Pops a from OpS and pushes the value a twice to OpS.
add	ba → x	Pops a and b from OpS and pushes back the value x = a+b.
sub	ba → x	Pops a and b from OpS and pushes back the value x = a-b.
mul	ba → x	Pops a and b from OpS and pushes back the value x = a*b.
div	ba → x	Pops a and b from OpS and pushes back the value x = a/b.
mod	ba → x	Pops a and b from OpS and pushes back the value x = a%b.
inc	a → x	Pops a from OpS and pushes back the value x = a+1.
dec	a → x	Pops a from OpS and pushes back the value x = a-1.
max	ba → x	Pops a and b from OpS and pushes back the value x = max(a,b).
min	ba → x	Pops a and b from OpS and pushes back the value x = min(a,b).

irnd	a → x	Pops a from OpS and pushes back the integer value x = random(0,a).
and	ba → x	Pops a and b from OpS and pushes back the value x = (a==1 && b==1) ? 1 : 0.
or	ba → x	Pops a and b from OpS and pushes back the value x = (a==1 || b==1) ? 1 : 0.
not	ba → bx	Pops a from OpS and pushes back the value x = (a==1) ? 0 : 1.
lt	ba → x	Pops a and b from OpS and pushes back the value x = a<b ? 1 : 0.
le	ba → x	Pops a and b from OpS and pushes back the value x = a≤b ? 1 : 0.
gt	ba → x	Pops a and b from OpS and pushes back the value x = a>b ? 1 : 0.
ge	ba → x	Pops a and b from OpS and pushes back the value x = a≥b ? 1 : 0.
eq	ba → x	Pops a and b from OpS and pushes back the value x = a≥b ? 1 : 0.
jmp	ba → b	Pops a from OpS and jumps to instruction at location a (representing program line idx).
cjmp	cba → c	Pops a and b from OpS and jumps to instruction at location a (representing program line idx) if b≠0.
call	...dcba → c	Pushes #PC+1 to AdS. Creates and pushes stack frame F to M storing b values popped from OpS (the parameters). Jumps to instruction at location a (representing instruction idx of called function).
ret	vu → v	Pops u from AdS and jumps to instruction at location u (representing program line idx).
halt	a → a	Halts the execution of the program. Should be present in the .main function.

oframe	ba → a	Pops a from OpS and allocates a new frame with space for a variables. Pushes the new frame onto the memory stack.
cframe	a → a	Pops the frame at the top of the memory stack SoF.
alloc	ba → b	Pops a from OpS and allocates a additional variable locations in the current scope.
delay	ba → b	Pops a from the OpS and pauses the execution of the program for a milliseconds.
write	dcba → d	Pops a,b,c values from OpS and changes the colour of pixel at location a,b to colour c.
writebox	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.
clear	ba → b	Pops a from the OpS and clears all pixels to colour a.
width	a → ax	Pushes onto OpS the value x = width of the PAD2000c display.
height	a → ax	Pushes onto OpS the value x = height of the PAD2000c display.
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.

```
let c:colour = 0 as colour;

for (let i:int = 0; i < 64; i = i + 1) {
    c = (__random_int 1677216) as colour;
    __clear c;

    __delay 16;
}
```

```
.main           //this is the entry point into the program
push 4          //push 4 on the operand stack
jmp             //jmp to instruction 4 (consume 4 from operand stack)
halt            //every program has a halt ... to quit
push 1          //the program needs to allocate space for 1 variable
oframe          //open frame - allocate space for variab
push 0          //value to store in c
push 0          //location index in the frame
push 0          //location level in the memory stack
st              //store 0 at index 0 of the frame located at level 0
push 1          //for loop - requires allocation for one variable.
oframe          //open a new frame - allocate space for variable i
push 0          //counter starts at 0
push 0          //location index in frame
push 0          //local level in memory stack
st              //store the value 0 to i.
push 64         //evaluate conditional expression. push 64
push [0:0]      //push value of i - from index 0, level 0
lt              //push 1 or 0 depending on result of lt
push #PC+4      //push current program counter+4 to stay in the loop
cjmp            //update program counter if 1 to #PC+4 (stay in loop)
push #PC+22     //push current program counter+22 to exit loop
jmp             //update program counter to exit loop
push 0          //0 new variables on the { } scope
oframe          //open frame for { } block, no space allocations required
push 1677216    //push max value for random operator
irnd            //pop 1677216 from stack and push random int value
push 0          //location index 0 in frame of variable c
push 2          //location level 2 in memory stack for variable c.
st              //store random int value to c
push [0:2]      //impl of __clear inst. first push value of var c
clear           //then clear the display with that value
push 16         //impl. of __delay inst. first push 16
delay           //call delay for 16 millisecond
cframe          //close block { } frame
push 1          //push 1
push [0:0]      //push value of i
add             //add i+1
push 0          //location index in frame of i
push 0          //location level of the frame
st              //store the new value (i+1) to i
push #PC-25     //push instruction location at expr. eval. of loop
jmp             //update program counter
cframe          //close frame for loop
cframe          //close frame main program
halt            //exit the program
```

Task 5 - Arrays in PArL

Arrays are an important data structure in all programming languages. PArDis management has always felt arrays should be included and have now requested it's developers to do so. PArIR instructions have been augmented to facilitate this task to the compiler developer. All instructions are now implemented in the CPS200c hardware and the VM simulator. In this task you are to augment your compiler with arrays functionality. An example program is listed below together with its equivalent PArIR instructions.

[Marks: 10%]

dupa	cv → v...v	Pops v (value) and c (count) from the OpS and pushes back to OpS the value v for c times.
sta	fedcba → f (if c==2)	Pops a (memory level), b (frame index), c (count) values from OpS and for i=0 to c-1 times, pops value from OpS and writes it to b+i
pusha [i:l]	ac → axy (if c==2)	Pushes the values in an array size c (popped from OpS), and pushes to OpS values starting at index i of the frame at stack level l. If memory location does not exist, an undefined value may be pushed to OpS.
push +[i:l]	ba → bx	Pops offset o from OpS and pushes back to OpS the value x located at index i+o of the frame at stack level l. Used to read array values at specific indices. If memory location does not exist, an undefined value may be pushed to OpS.
printa	dbac → d (if (c==2)	Pops c (count) from OpS and prints to the logs section c popped values from the OpS. Mainly used to print arrays.
reta	dbac → dab (if (c==2)	Pops c (count) and c values from OpS and pushes them back in reverse order to OpS. Used to return an array.

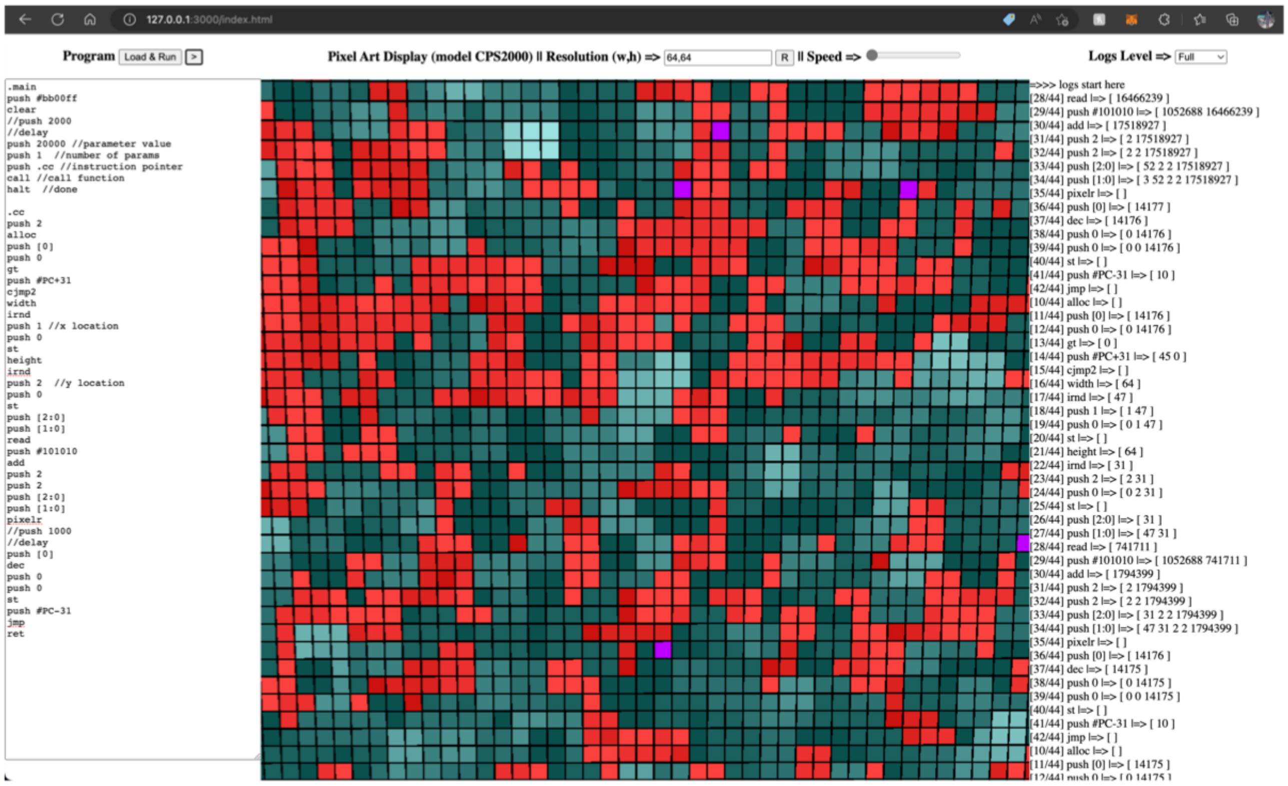
```
//x is an array of 16 +ve integers
fun MaxInArray(x:int[8]) -> int {
  let m:int = 0;
  for (let i:int = 0; i < 8; i = i+1) {
   if (x[i] > m) { m = x[i]; }
  }
  return m;
}

let list_of_integers:int[] = [23, 54, 3, 65, 99, 120, 34, 21];
let max:int = MaxInArray(list_of_integers);
__print max;
```

```
.main
push 4
jmp
halt
push 10             //
oframe
push #PC+52
jmp
.MaxInArray         //start of MaxInArray function instructions
push 9              //how many local variables. 9 is over-allocating.
alloc               //create the space for variable m
push 0              //initial value of variable m
push 8              //location index in frame - following array
push 0              //frame level in memory stack
st                  //store 0 to m
push 1              //for loop variables between ( ). Just i here.
oframe              //create a new scope and allocate space for i
push 0              //initial value of i
push 0              //index of i in the frame
push 0              //location of frame in memory stack
st                  //store value 0 to i
push 8              //start of < evaluation. push RHS operand
push [0:0]          //push LHS operand
lt                  //carry out operation
push #PC+4          //start of instructions implementing for block
cjmp                //if true jump there
push #PC+29         //instruction location is condition is false
jmp                 //go there. exit for loop.
push 0              //no new local variables in the for block
oframe              //create a new frame for the for loop block
push [8:2]          //push value of m
push [0:1]          //push value of i
push +[0:2]         //push value of x offset by index i.
gt                  //compare x[i] with m
push #PC+4          //push instruction line number if true
cjmp                //go there is true
push #PC+10         //push instruction line number if false
jmp                 //just go there
push 0              //if true scope has no new local variables
oframe              //create the scope for the if true block
push [0:2]          //push value of i (note frame level is now 2)
push +[0:3]         //push value of x at idx i (note frame level is 3)
push 8              //push location of m in frame
push 3              //push frame level were m is stored
st                  //update value of m
cframe              //close frame (if true block)
cframe              //close frame (for loop)
push 1              //start of instructions for i=i+1; push 1
push [0:0]          //push value of i
add                 //add i and 1
push 0              //location index in frame for variable i
push 0              //frame level in the memory stack
st                  //assign the new value of i
push #PC-32         //push line number of bool expression of the for loop
jmp                 //go there
cframe              //close function frame
push [8:0]          //push value of m
ret                 //return from function
push 21             //push last array value
push 34             //push array value
push 120            //push array value
push 99             //push array value
push 65             //push array value
push 3              //push array value
push 54             //push array value
push 23             //push array value
push 8              //push size of array
push 1              //push index in frame where array starts
push 0              //push frame level on memory stack
sta                 //pop all 8 array values and store them
push 8              //prepare for call - push size of array
pusha [1:0]         //push array values starting at index 1
push 8              //8 values are passed to MaxInArray function
push .MaxInArray    //push line number of instruction starting function
call                //call function .. update program counter
push 9              //location index in frame for variable max
push 0              //location level in memory stack
st                  //store returned value to max
push [9:0]          //push value of max
print               //print to the console
cframe              //close program frame
halt                //exit program
```

Online Simulator - PArDis CPS2000

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 will be available online shortly.


127.0.0.1:3000/index.html
Program
Load & Run
>
Pixel Art Display (model CPS2000) || Resolution (w,h) =>
64,64
R
|| Speed =>
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

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 PArL 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.