

BattleShip in Jack & Parser

A project report submitted by

Sarvesh Shashikumar [CB.EN.U4AIE21163]

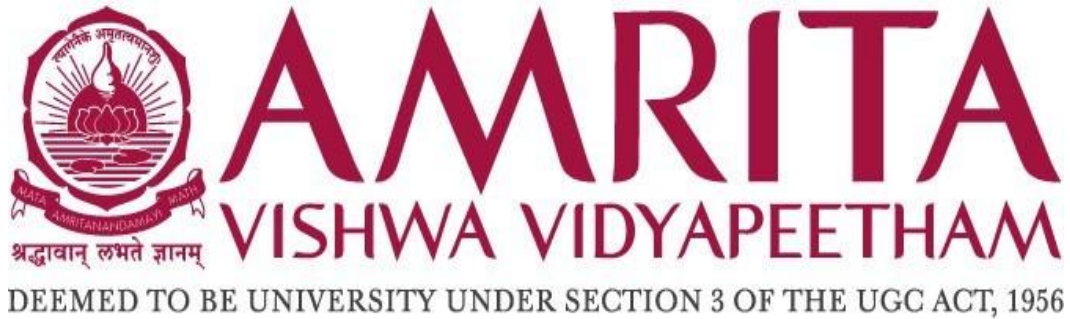
Kalyana Sundaram [CB.EN.U4AIE21120]

Kaushik Jonnada[CB.EN.U4AIE21122]

Subikksha [CB.EN.U4AIE21167]

As part of the subject:

Elements of computing - II



**Center for Computational Engineering and Networking
AMRITA SCHOOL OF ENGINEERING
AMRITA VISHWA VIDYAPEETHAM
COIMBATORE - 641 112 (INDIA)
July - 2022**

AMRITA SCHOOL OF ENGINEERING
AMRITA VISHWA VIDYAPEETHAM
COIMBATORE - 641 112

DECLARATION

We solemnly declare that the project report is based on our own work carried out during the course of our study under the supervision of Ms. Sreelakshmi K. We assert the statements made and conclusions drawn are an outcome of our research work. We further certify that:

I. The work contained in the report is original and has been

done by us under the general supervision of our supervisor.

II. The work has not been submitted to any other Institution

for any other degree/diploma/certificate in this university or any other University of India or abroad.

III. We have followed the guidelines provided by the university in writing the report.

IV. Whenever we have used materials (data, theoretical analysis, and text) from other sources, we have given due credit to them in the text of the report and giving their details in the references.

Place: Ettimadai

Date: 16-7-2022

Contents

Acknowledgement

About the project

Abstract

Chapter -1 : Parser

- Code & Explanation
- Standard libraries in jack and their functions

Chapter -2 : BattleShip

- What exactly does it mean and how to play it
- How to use the interface and give game inputs
- What we used to make the code.
- How it works

Acknowledgement

The completion of this project has been made possible by the efforts of many. We would like to thank Dr. Sasangan Ramanathan, Dean - Engineering, and our faculty for the course Elements of Computing - II, Ms. Sreelakshmi K, for giving us this end semester project and allowing us to work together to finish the same and for their unstinting support. Our team has also learned a lot of valuable information while doing so and we would like to acknowledge our thanks to you. We would also like to extend our gratitude to our friends and family who have aided us throughout the project.

About the Project

Part - 1: Making the two-player battleship game & a parser

Part - 2: Making an assembler in python.

Part - 3: Use assembler to get machine code generated in RAM.hdl and check the same in CPU Emulator.

Abstract

For this term project, we made a game known as battleship and a parser, both of which used a programming language known as jack.

The battleship game is two-player and has 3 two-size ships and 3 three-size ships.

This report explains about the game and the parser and the basics of the programming language jack.

CHAPTER-1: PARSER

What is Jack?

Jack language is a high level language that is designed to aid programmers to write high level programs. It is an object-oriented language which is simple minded and contains the basic features and the essence of modern languages. The Jack language poses just like languages like Java and C# with a less complicated syntax and no support for inheritance. Jack is a general purpose language which is used to create a wide range of applications. It has proven to be highly helpful in simple interactive games like Snake, Tetris and BattleShip.

Standard Libraries in Jack

The Jack language comes with a standard library which can also be looked at as an interface to an underlying operating system. The collection of classes in Jack must be provided in every implementation done using the Jack language and is called the library. The standard library includes the following classes:

- Math: The math class provides basic mathematical operations
- String: The string class implements the type 'String' and basic operations related to the type.
- Array: The array class defines the type 'Array' and allows its construction and disposal.
- Output: The output class deals with the text based output.
- Screen: The screen class deals with the screen output which is graphic in nature.
- Keyboard: The keyboard class handles the input from the user from the keyboard.
- Memory: The memory class takes care of the memory operations.
- Sys: The sys class provides services related to execution.

Math:

- ★ Function `int abs(int x)`: It returns the absolute value of x.
- ★ Function `int multiply(int x, int y)`: It returns the product of x and y
- ★ Function `int divide(int x, int y)`: It returns the integer part of the division x/y .
- ★ Function `int min(int x, int y)`: It returns the minimum of x and y.

- ★ Function `int max(int x, int y)`: It returns the maximum of x and y.
- ★ Function `int sqrt(int x)`: Returns the integer part of the square root of x.

String:

- ★ This class implements the String data type and various string-related operations.
- ★ Constructor `String new(int maxLength)`: Constructs a new empty string (of length zero) that
- ★ can contain at most `maxLength` characters.
- ★ Method `void dispose()`: Disposes this string.
- ★ Method `int length()`: Returns the length of this string.
- ★ Method `char charAt(int j)`: Returns the character at location j of this string.
- ★ Method `void setCharAt(int j, char c)`: Sets the j'th element of this string to c.
- ★ Method `String appendChar(char c)`: Appends c to this string and returns this string.
- ★ Method `void eraseLastChar()`: Erases the last character from this string.
- ★ Method `int intValue()`: Returns the integer value of this string (or at least of the prefix until
- ★ a non numeric character is found).
- ★ Method `void setInt(int j)`: Sets this string to hold a representation of j.
- ★ Function `char backSpace()`: Returns the backspace character.
- ★ Function `char doubleQuote()`: Returns the double quote (") character.
- ★ Function `char newLine()`: Returns the newline character.

Array:

- ★ Construction and destruction of arrays are enabled by this class.
- ★ Function `Array new(int size)` constructs a new array of the given size.
- ★ Method `void dispose()` disposes the array

Output:

- ★ This class allows writing text on the screen.
- ★ Function `void moveCursor(int i, int j)`: Moves the cursor to the j'th column of the i'th row and erases the character located there.
- ★ Function `void printChar(char c)`: Prints c at the cursor location and advances the cursor one column forward.

- ★ Function void printString(String s): Prints s starting at the cursor location, and advances the cursor appropriately.
- ★ Function void printInt(int i): Prints i starting at the cursor location, and advances the cursor appropriately.
- ★ Function void print(): advances the cursor to the beginning of the next line.
- ★ Function void backSpace(): Moves the cursor one column back.

Screen:

- ★ The class screen allows graphics on the screen. Column indices start at 0 and are left to right.
- ★ Row indices start at 0 and are from top to bottom. The screen size is hardware dependent.
- ★ Function void clearScreen(): Erases the entire screen.
- ★ Function void setColor(boolean b): Sets the screen color to be used for all further draw XXX commands.
- ★ Function void drawPixel(int x, int y) draws the (x,y) pixel.
- ★ Function void drawLine(int x1, int y1, int x2, int y2) draws a line from pixel (x1,y1) to pixel (x2,y2).
- ★ Function void drawRectangle (int x1, int y1, int x2, int y2) draws a filled rectangle.
- ★ Function void drawCircle(int x, int y, int r) draws a filled circle of radius r around(x,y).

Keyboard:

- ★ The keyboard class allows reading the inputs via the keyboard.
- ★ Function char keyPressed(): Returns the character of the currently pressed key on the keyboard; if no key is currently pressed, returns 0.
- ★ Function char readChar(): Waits until a key is pressed on the keyboard and released, and then echoes the key to the screen and returns the character of the pressed key.
- ★ Function String readLine(String message): Prints the message on the screen, reads the next line (until a newline character) from the keyboard, echoes the line to the screen, and returns its value. This method handles user backspaces.
- ★ Function int readInt(String message): Prints the message on the screen, reads the next line (until a newline character) from the keyboard, echoes the line to the screen, and returns the integer until the first non numeric character in the line. This method handles user backspaces.

Memory:

- ★ The memory class guarantees straight access to the main memory.
- ★ Function `int peek(int address)`: Returns the value of the main memory at this address.
- ★ Function `void poke(int address, int value)`: Sets the value of the main memory at this address to the given value.
- ★ Function `Array alloc(int size)`: Allocates the specified space on the heap and returns a reference to it.
- ★ Function `void deAlloc(Array o)`: De-allocates the given object and frees its memory space.

Sys:

- ★ Execution related services are supported by the Sys class.
- ★ Function `void halt()`: Halts the program execution.
- ★ Function `void error(int errorCode)`: Prints the error code on the screen and halts.
- ★ Function `void wait(int duration)`: Waits approximately duration milliseconds and returns.

Subroutine calls

- The general syntax `subroutineName` is used in subroutine calls to invoke methods, functions, and constructors for their intended effects (argument-list).
- The subroutine's declared parameters, as well as the number and type of arguments passed in, must match. Even if the argument list is empty, the parentheses must still be present.
- Each argument could express any level of complexity. For instance, Jack's standard library's Math class has a square root function with the declaration `function int sqrt (int n)`.
- Calls like `Math.sqrt (17)`, `Math.sqrt ((a * Math.sqrt (c- 17) + 3)`, and so forth can be used to call this function.

Parser

Every programming language needs some form of help to help it understand what is code ,what is comment and what is WHAT !

Similarly it also needs help in showcasing where error is , what error it is .

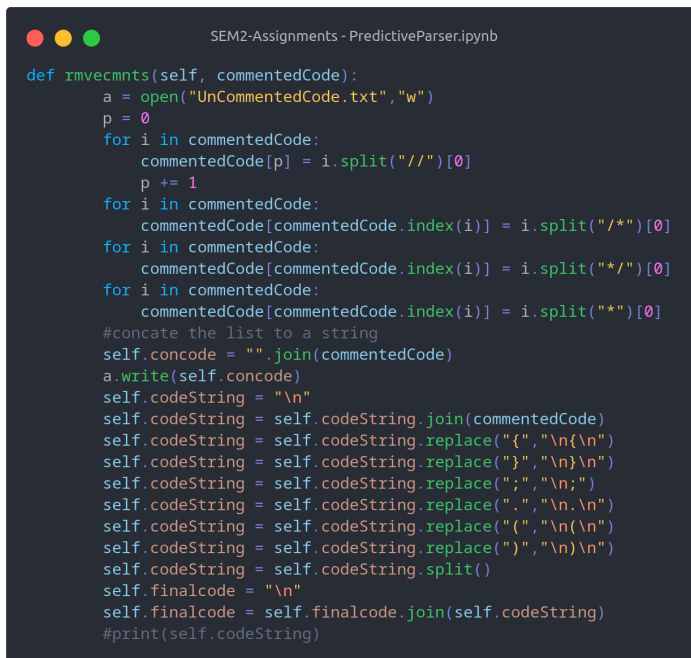
Parser does exactly the same work !

Here we made a parser with similar functionalities for Jack Language from scratch.

Splitting up the work!

- First we need to remove comments which the compiler rejects.
- Then comes the tokenizing part , where we give a token to each and every element in the input file.
- Then we make definitions for each statement in the jack language.
- We need to keep in mind that Jack Language follows certain grammar , we need to abide by it.

Removing Comments :



```
SEM2-Assignments - PredictiveParser.ipynb

def rmvecmnts(self, commentedCode):
    a = open("UnCommentedCode.txt", "w")
    p = 0
    for i in commentedCode:
        commentedCode[p] = i.split("//")[0]
        p += 1
    for i in commentedCode:
        commentedCode[commentedCode.index(i)] = i.split("/*")[0]
    for i in commentedCode:
        commentedCode[commentedCode.index(i)] = i.split("*/")[0]
    for i in commentedCode:
        commentedCode[commentedCode.index(i)] = i.split("**")[0]
    #concat the list to a string
    self.concode = "".join(commentedCode)
    a.write(self.concode)
    self.codeString = "\n"
    self.codeString = self.codeString.join(commentedCode)
    self.codeString = self.codeString.replace("{", "\n{\n")
    self.codeString = self.codeString.replace("}", "\n}\n")
    self.codeString = self.codeString.replace(";", "\n;")
    self.codeString = self.codeString.replace(".", "\n.\n")
    self.codeString = self.codeString.replace("(", "\n(\n")
    self.codeString = self.codeString.replace(")", "\n)\n")
    self.codeString = self.codeString.split()
    self.finalcode = "\n"
    self.finalcode = self.finalcode.join(self.codeString)
    #print(self.codeString)
```

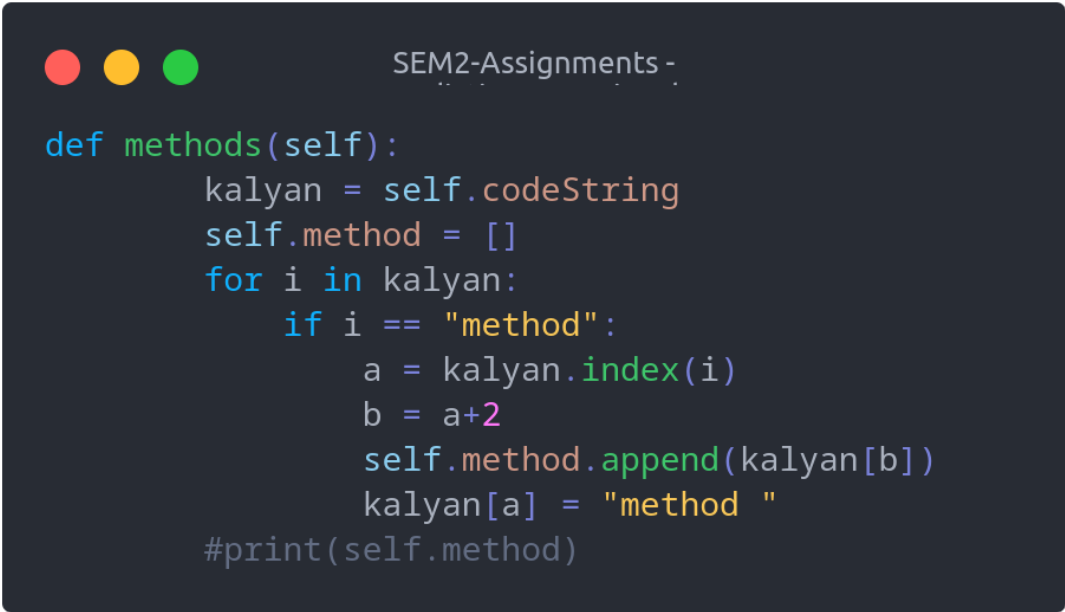
Here we first make a file called

UnCommentedCode which will have files free of comments.

We open it in write format.For each line in commentedCode(source code) if “//” or “/*” or “/” is found it splits at that point .

And if symbols are found it a new line gets added after and before it.Finally we join all the splitted and replaced lines into a single list.

Method Finding:



```
def methods(self):
    kalyan = self.codeString
    self.method = []
    for i in kalyan:
        if i == "method":
            a = kalyan.index(i)
            b = a+2
            self.method.append(kalyan[b])
            kalyan[a] = "method "
    #print(self.method)
```

This method checks if a method is made and adds it to a list for future reference.

Logic is for each line in self.codeString if that i = “method” , i.e, that element has method in it , then it adds the method name to the method list.

The index of “method” + 2 will always be the method's name .

Dot Processing :

```
SEM2-Assignments - PredictiveParser.ipynb

def dotprocessing(self):
    self.tokenizeddots = []
    self.splittingdots = []

    for i in self.codeString:
        if "." in i:
            self.splittingdots.append(i.split("."))
            self.splittingdots[-1].insert(1, '.')
        else:
            self.splittingdots.append(i)
```

This method starts by making a list called splitting dots .

For each element in self.codeString if there is a dot we split at that point and add the previous element to its place .

We essentially add a dot if it is found in the original element.

xml tokenizing :

```
SEM2-Assignments - PredictiveParser.ipynb

def xmltokenizer(self):
    self.tokenedcode = []

    temp1 = 0
    for words in self.codeString:
        for keywords in self.kw:
            if words == keywords:
                self.tokenedcode.append("<keyword>" + words + "</keyword>")
            else:
                temp1 += 1
```

We make a list called tokenedcode , and for each word in self.codeString if that word matches self.kw then it must be a keyword !

```
SEM2-Assignments - PredictiveParser.ipynb

for keywords in self.symbols:
    if words == keywords:
        self.tokenedcode.append("<symbol>" + words + "</symbol>")
    else:
        temp1 +=1
```

If the words are present in self.symbols it must be an symbol .

```
SEM2-Assignments - PredictiveParser.ipynb

for keywords in self.stringconstants:
    if words == keywords:
        self.tokenedcode.append("<stringConstant>" + words + "</stringConstant>")
    else:
        temp1 +=1
for keywords in self.integerconstant:
    if words == keywords:
        self.tokenedcode.append("<intConstant>" + words + "</intConstant>")
    else:
        temp1 +=1
for keywords in self.op:
    if words == keywords:
        self.tokenedcode.append("<unaryoperator>" + words + "</unaryoperator>")
    else:
        temp1 +=1
for keywords in keywords:
    if words not in self.kw and words not in self.symbols and words not in self.stringconstants and words not in self.op and words not in self.integerconstant:
        self.tokenedcode.append("<identifier>" + words + "</identifier>")
    else:
        temp1 +=1
```


Same way if it is in self.stringConstant , self.integerConstant , op it is stringConstant,integerConstant and unary operator respectively.

```
SEM2-Assignments -

a = open("tokenizedcode.xml", "w")
a.write("<tokens>\n")
for i in self.tokenedcode:
    a.write(i + "\n")
a.write("</tokens>\n")
```

We then write all these tokenedcode into an xml file and save it in the same directory as the .ipynb file.

Symbol Check :



```
def symbolcheck(self):
    self.expressioncount = 0
    for i in self.codeString:
        if i == '{':
            self.expressioncount += 1
        elif i == '}':
            self.expressioncount -= 1
        elif i == '[':
            self.expressioncount += 1
        elif i == ']':
            self.expressioncount -= 1
        elif i == '(':
            self.expressioncount += 1
        elif i == ')':
            self.expressioncount -= 1
    if self.expressioncount == 0:
        print("Symbols OK")
    elif self.expressioncount > 0:
        print("Symbols not closed")
```

This definition is to check if the symbols are equally enclosed !

If not then it asks the user to check the symbols again.

This definition goes through all elements of the codeString list and has a counter .

If i is found it adds 1 to the counter, if the corresponding closing bracket is found it reduces the count variable.

OUTPUT:

Jack code as input

```
SEM2-Assignments - Main.jack

// This file is part of www.nand2tetris.org
// and the book "The Elements of Computing Systems"
// by Nisan and Schocken, MIT Press.
// File name: projects/09/Average/Main.jack

// Inputs some numbers and computes their average
class Main {
    function void add() {
        var Array a;
        var int length;
        var int i;

        var int sum;

        let length = Keyboard.readInt(" How many numbers? ");
        let a = Array.new(length); // constructs the array
        let k = a + 1;
        let i = 0;
        while (i < length) {
            let a [ i ] = Keyboard.readInt(" Enter a number: ");
            let sum = sum + a [ i ];
            let i = i + 1;
        }

        do Output.printString(" The average is ");
        do Output.printInt(sum / length);
        return;
    }
}
```

Output of the parser

```
Symbols OK
Error: Variable name not declared
['let', 'k', '=', 'a', '+', '1']
Number of if statements: 0
Number of else statements: 0
Number of while statements: 1
Number of do statements: 0
Number of return statements: 1
Number of let statement: 2
Number of variable declarations: 4
```

Output of tokenizer

```
<tokens>
<keyword>class</keyword>
<stringConstant>Main</stringConstant>
<symbol>{</symbol>
<keyword>function</keyword>
<keyword>void</keyword>
<identifier>add</identifier>
<symbol>(</symbol>
<symbol>)</symbol>
<symbol>{</symbol>
<keyword>var</keyword>
<keyword>Array</keyword>
<identifier>a</identifier>
<symbol>;</symbol>
<keyword>var</keyword>
<keyword>int</keyword>
<identifier>length</identifier>
<symbol>;</symbol>
<keyword>var</keyword>
<keyword>int</keyword>
<identifier>i</identifier>
<symbol>;</symbol>
<keyword>var</keyword>
<keyword>int</keyword>
<identifier>sum</identifier>
<symbol>;</symbol>
<keyword>let</keyword>
<identifier>length</identifier>
<uniaryoperator>=</uniaryoperator>
<keyword>Keyboard</keyword>
<symbol>.</symbol>
<identifier>readInt</identifier>
<symbol>(</symbol>
<symbol>"</symbol>
```

CHAPTER-2 : BATTLESHIP

Using these in jack, we created a game most commonly known as **BattleShip** in the modern day. So, what exactly is battleship, and how do you play it? Well, here are the basics you need to know to play it.

This is a two-player game. Basically, there are 2 three-size ships and 2 two-size ships. There exists a grid in which we are able to place our ships either horizontally or vertically. The purpose of this arrangement is so that the other player tries to guess where exactly the ships are by giving a grid location having both row and column. Then, the player who arranged the ships will say either 'HIT' or 'MISS'. If all the locations of the particular ship are hit, then the ship is considered sunk and lost. This continues until all the ships are lost and the player eventually wins.

[illegible]

When we take an example as shown in the picture above, the first player decided to place their 2-size ships at 3a - 3b, 3e - 4e and 1j - 2j. Similarly, the 3-seize ships were decided to be placed at 8c - 8e, 6f - 6h and 8j - 10j. Now, the opposing player has to guess where the ships are located. If the guess is correct, it is considered a hit and if it is incorrect, it is considered as a miss. The game ends when all the ships are sunk.

To actually play this game, the user needs to give inputs to choose where they want to place their 4 ships. So when it prompts you to type where you want to place the ships, you input the row and column value where you want the ship to start respectively and then for the next prompt, you enter the row and column values where you want the ship to end at respectively.

For example, in the image above when the user wanted to input the first ship, they entered '3' as the row and then 'a' as the column value for where they wanted the ship to start. When the next prompt was given, they had to type '3' as the row and 'b' as the column. This results in the ship being in the way it looks at the top left on the 3rd row. And if the ship is of size three, then the beginning and ending distance would be of three instead of two. For example we can take the ship in the bottom right corner, When the user is prompted to add the ship, the first input would have been '8' and second would be 'j'. And for the next prompt, for when it ends, the user shall type '10' and 'j'. This is how all the ships are arranged in an order according to the first user's wish.

Now, we come to the second player's part. In this part, the player doesn't know where the first player has hidden the ships. This means that the rows and columns of the ships are hidden from the second user. This person gets a prompt to give an attempt to guess where the ships are hidden. The user can type the row and column of any random grid and if the ship is there, the interface will show a 'HIT' beside the grid and if the user guesses an empty space where the ship doesn't exist, the interface will show a 'MISS' beside the grid. In either case, the user will keep getting prompts to keep on guessing until all the ships are hit and they sink. At the end of the game, after the second user has sunk all the ships, There will be a display message saying: 'YOU WIN'.

We accomplished to make this game using a programming language known as jack. Jack is nothing but a simple object-based language which can be used to write high level programs. The basic programming unit in Jack is a class.

Each class exists in a separate file and can be compiled individually. The class declaration have the format as:

```
class name  
  
{  
  
    static variable declarations  
  
}
```

Jack programs:

A jack program consists of one or more classes. One of the classes must be named 'Main' just like in java. When you instruct to execute a jack program from a directory, the host platform will automatically start running the **Main.main** function.

Variables in Jack must be explicitly declared before they are used. There are several kinds of variables: fields, static variables, local variables, and parameters, each with its associated scope.

Data structures can exist in either primitive or object type. Primitive are like: (int, char, boolean) which are pre defined.

On the other hand, Object type are defined manually by the programmer, and can be named according to the programmer's needs.

Arrays:

Arrays are declared using the in-built class 'Array'. They are single dimensional and the first index is always '0'. Array entries do not have a declared type, and different entries in the same array may have different types.

Strings:

Strings are declared using the built-in class String. The compiler also recognizes the syntax: "enter your string here", basically a character or group of characters enclosed within the double quotes. String contents can be accessed and modified using the methods of the String class.

Statements:

The Jack language features 5 statements. They are:

- Let
- If
- while
- do
- return

Code:

```
SEM2-Assignments - AttackJack

class Attack {

    static int rowInput, columnInput, score, hits;
    static Array ship1Row, ship1Column, ship2Row, ship2Column, ship3Row, ship3Column, ship4Row, ship4Column;

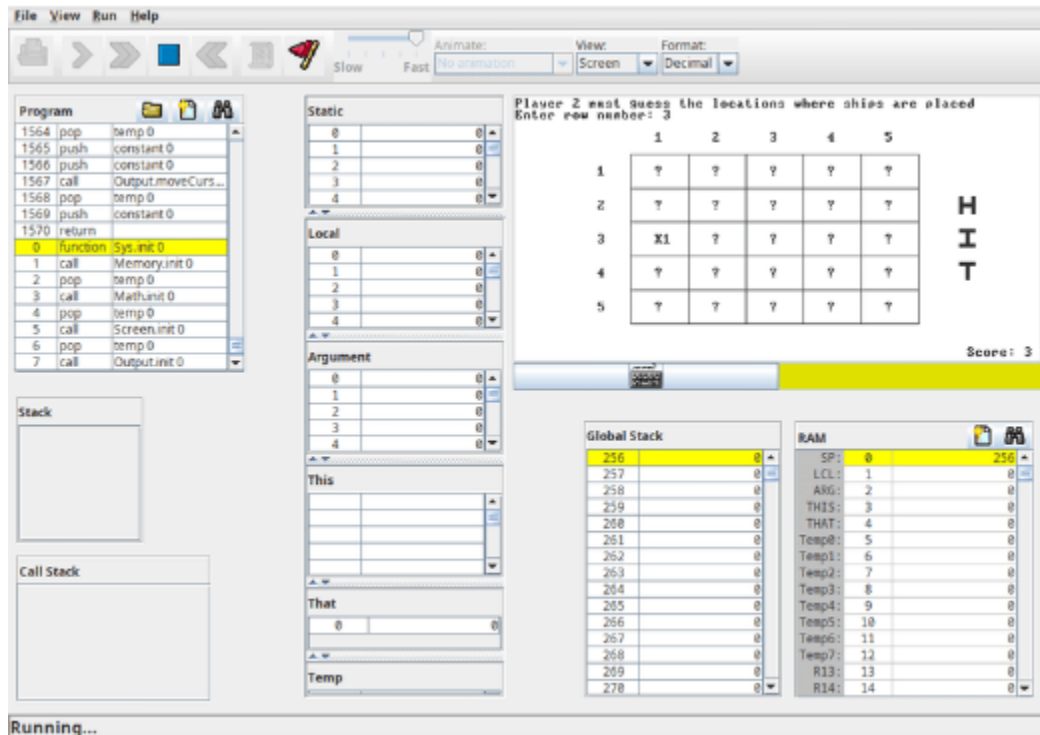
    function void setArrs(){
        do Attack.getshipRows();
        do Attack.getshipColumns();
        return;
    }

    function void askForAttack() {
        do Attack.setArrs();
        do Output.printString("Player 2 must guess the locations where ships are placed ");
        let rowInput = Keyboard.readInt("Enter row number: ");
        let rowInput = 6 + (3 * (rowInput - 1));
        do Output.moveCursor(2, 0);
        let columnInput = Keyboard.readInt("Enter column number: ");
        let columnInput = 17 + (7 * (columnInput - 1));
        do Attack.compare();
        return;
    }

    function void compare() {
        var int iter, bs, memAddress;
        // to compare to ship 1 stuff
        let iter = 0;
        let bs = 22;
        while (iter < 3) {
            if (rowInput = ship1Row[iter]){
                if (columnInput = ship1Column[iter]) {
                    do Output.moveCursor(rowInput, columnInput);
                    do Output.printString("X1");
                    while (bs > (-1)){
                        do Output.moveCursor(2, bs);
                        do Output.backSpace();
                        let bs = bs - 1;
                    }
                }
            }
        }
    }
}
```

Code snippet of Attack.jack

Output :



First we ask the user 1 to input the coordinates of ship 1 and user 2 to input the coordinates of ship 2 .

Then we ask the users to guess the coordinates of each other's ship.

If the guess is correct we add 1 to the score board and display HIT.

Contributions:

M.Kalyana Sundaram - Parser

Kaushik Jonnada - Theory & Debugging

Sarvesh ShashiKumar - BattleShip

Subikksha - Theory & Debugging