Lab 9: Rust and Web Assembly

Part 1: Starting Web Assembly

Rust gives programmers low-level control and reliable performance. It is free from the non-deterministic garbage collection pauses that plague JavaScript. Programmers have control over indirection, monomorphization, and memory layout.

WebAssembly (wasm)

WebAssembly (wasm) is a simple machine model and executable format with an extensive specification. It is designed to be portable, compact, and execute at or near-native speeds.

As a programming language, WebAssembly is comprised of two formats that represent the same structures, albeit in different ways:

1- The .wat text format (called wat for "**W**eb**A**ssembly **T**ext") uses S-expressions.
2- The .wasm binary format is lower-level and intended for consumption directly by wasm virtual machines.

For reference, here is a factorial function in wat:

```
(module
  (func $fac (export "fac") (param f64) (result f64)
    local.get 0
    f64.const 1
    f64.lt
    if (result f64)
      f64.const 1
    else
      local.get 0
      local.get 0
      f64.const 1
      f64.sub
      call $fac
      f64.mul
    end))
```

For more information about the webAssembly syntax and text format, please refer to : https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format

To transform the above wat code to wasm (and run the code), you can use the wat2wasm **(https://webassembly.github.io/wabt/demo/wat2wasm/)** demo with the above code (need to change the two “addTwo” instances to “RecursiveCount” in the JS window in order for your code’s output to appear in the JS LOG window).

Question 1: Change the above code as follows:

1- Change the function name into ***RecursiveCount***
2- Change the function argument and return type to i32
3- Change the logic of the function, so instead of returning the factorial of a number, it returns the sum of all the integers in a range between any number (between 1 and 9) and the number 10.
For example RecursiveCount(9)=9+10 =19

RecursiveCount(7)= 7+8+9+10=34

Transform the code after these changes to *.wasm* and **DEMO this deliverable to the lab instructor. (Hint: all you need to write is already in the program)**

Part 2：Hello，World！

To set up the toolchain for compiling Rust programs to WebAssembly and integrate them into JavaScript, you have to follow the following steps:

1- You need the rust toolchain (rustup, rustc, and cargo), you should already have it installed in your machine. For more information, review Lab 1.

2- You need wasm-pack, which is your one-stop-shop for building, testing, and publishing Rust-generated WebAssembly:

```
cargo install wasm-pack
```

3- You need cargo-generate to help you get up and running quickly with a new Rust project by leveraging a pre-existing git repository as a template. To install cargo-generate run the following command:

```
cargo install cargo-generate
```

4- Finally, you need npm, which is a package manager for JavaScript. We will use it to install and run a JavaScript bundler and development server. To install npm, follow the instructions from here: https://docs.npmjs.com/downloading-and-installing-node-js-and-npm .

Now, let's create our first WebAssembly project, a web page that alerts "Hello World!".

1- To start the project, you can clone a project template which comes pre-configured with sane defaults, so you can quickly build, integrate, and package your code for the web. To clone the project template, use the following command (you may need to install git or github to run this command):

```
cargo generate --git https://github.com/rustwasm/wasm-pack-template
```

2- Running the command should prompt you for the new project's name. Name your project "wasm-is-prime"

3- Then take a look at the project contents:

```
cd wasm-is-prime
```

```
wasm-is-prime/
├── Cargo.toml
├── LICENSE_APACHE
├── LICENSE_MIT
├── README.md
└── src
    ├── lib.rs
    └── utils.rs
```

4- Let's take a look at a /src/lib.rs
The src/lib.rs file is the root of the Rust crate that we are compiling to WebAssembly. It uses `wasm-bindgen` to interface with JavaScript. It imports the window.alert JavaScript function, and exports the greet Rust function, which alerts a greeting message.

```
#[wasm_bindgen]
pub fn greet() {
    alert("Hello, wasm-is-prime!");
}
```

5- To build the project, we use wasm-pack to do the following task

- Ensure that we have Rust 1.30 or newer and the wasm32-unknown-unknown target installed via rustup.
- Compile our Rust sources into a WebAssembly .wasm binary via cargo.
- Use wasm-bindgen to generate the JavaScript API for using our Rust-generated WebAssembly.

To do all of that, run this command inside the project directory:

```
wasm-pack build
```

6- When the build has completed, we can find its artifacts in the pkg directory, and it should have these contents:

```
pkg/
├── package.json
├── README.md
├── wasm_is_prime_bg.wasm
├── wasm_is_prime.d.ts
└── wasm_is_prime.js
```

7- Now, to build into a web page, you can use the create-wasm-app JavaScript project template by running the following command within the project directory:

```
npm init wasm-app www
```

Here's what our new wasm-is-prime/www subdirectory contains:

```
wasm-is-prime/www/
├── bootstrap.js
├── index.html
├── index.js
├── LICENSE-APACHE
├── LICENSE-MIT
├── package.json
├── README.md
└── webpack.config.js
```

8- To ensure that the local development server and its dependencies are installed, run npm install within the wasm-is-prime/www subdirectory:

```
npm install
```

Note that this command only needs to be run once, and will install the webpack JavaScript bundler and its development server.

9- To incrementally develop our project, we need to specify the dependencies to include wasm-is-prime
Do that by opening up wasm-is-prime/www/package.json and adding "dependencies" to also add the "wasm-is-prime ": "file:../pkg" entry:

```
{
  // ...
  "dependencies": {
    "wasm-is-prime": "file:../pkg", // Add this line!
  }
}
```

10- Next, modify wasm-is-prime/www/index.js to import wasm-is-prime instead of the hello-wasm-pack package:

```
import * as wasm from "wasm-is-prime";
```

```
wasm.greet();
```

11- Since we declared a new dependency, we need to install it:

```
    npm install
```

12- As of Node version 17 and later, you may also need to run the following code in the terminal to avoid the OpenSSL error (if you are using Visual Studio Code, you may need to run on a terminal that is NOT built in Visual Studio Code for the command to work). Note: This command only needs to be run once per terminal session:

For Windows:

```
    set NODE_OPTIONS=--openssl-legacy-provider
```

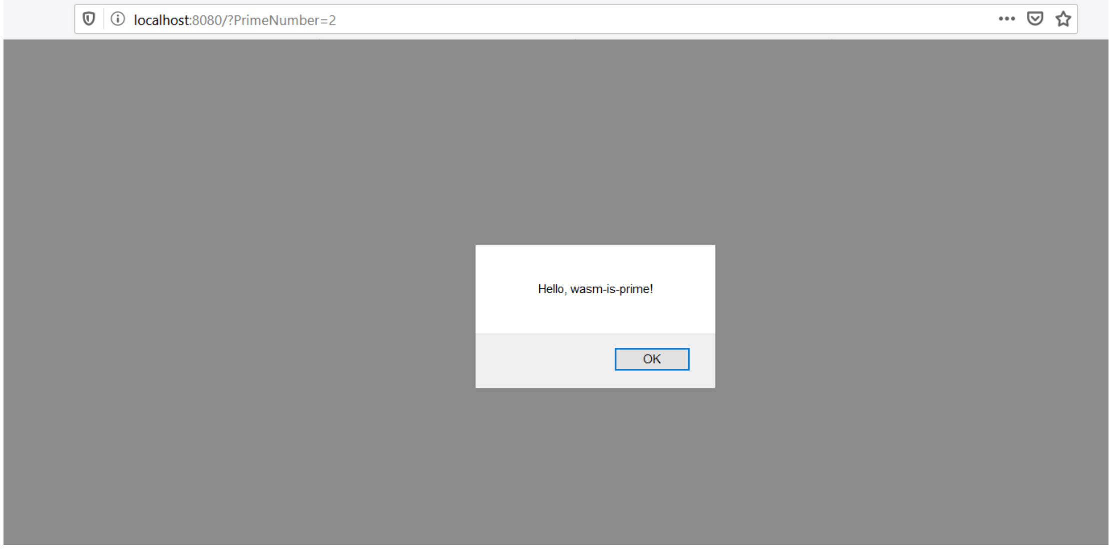
For Mac/Linux:

```
    export NODE_OPTIONS=--openssl-legacy-provider
```

13- Finally, open a new terminal for the development server. Running the server in a new terminal lets us leave it running in the background, and doesn't block us from running other commands in the meantime. In the new terminal, run this command from within the `wasm-is-prime/www` directory:

```
    npm run start
```

Navigate your Web browser to http://localhost:8080/, and you should be greeted with an alert message:



Hello, wasm-is-prime!

Anytime you make changes and want them reflected on http://localhost:8080/, just re-run the wasm-pack build command within the wasm-is-prime directory.

Summary to run code:

Run in wasm-is-prime directory:

```
    wasm-pack build // Run only if rust code in /src/lib.rs changes.
```

Run in www directory:

```
     npm install // Run only if code in the www folder changes or if wasm-pack build was run.

     set NODE_OPTIONS=--openssl-legacy-provider // (Windows) Run if new terminal session.

     export NODE_OPTIONS=--openssl-legacy-provider // (Mac/Linux) Run if new terminal session.

     npm run start // Runs code.
```

For more tutorials refer to the online book: https://rustwasm.github.io/book/game-of-life/introduction.html#tutorial-conways-game-of-life

- **DEMO this deliverable to the lab instructor.**

Part 3: Implementing is-prime app

is-prime app is a small application that takes input from the user (the input must be an integer). Then, it checks if this input is prime and alerts the user with the results of the check. For example, if the user inputs a prime number, the application should show an alert message that says, “your input is a prime number.” Otherwise, the app should display a message that says, “your input is NOT a prime number.”

1- In the last part, we cloned an initial project template. We will modify that project template now. Let's begin by modifying the wasm-is-prime/src/lib.rs by adding the following two functions:

```
#[wasm_bindgen]
pub fn check_prime(s: &JsValue) {
    let input: String = s.as_string().unwrap();
    match input.parse::<u32>() {
        Ok(num) => {
            if is_prime(num) == 1 {
                alert("Input is Prime");
             } else {
                alert("Input is NOT Prime");
             }
        },
        Err(_) => alert(&format!(“Couldn’t parse {}”, input)),
    }
}

fn is_prime(n: u32)-> u32 {
     // add your code to check prime here
     n % 2 // This line currently checks if a number is odd or even.
}
```

2- Now, let’s update the wasm-is-prime/www/index.html file to add a text input and a button as follows:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Is Prime App!</title>
  </head>
  <body>
    <noscript>This page contains webassembly and javascript content,
              please enable javascript in your browser.</noscript>
    <script src="./bootstrap.js"></script>
    <form>
      Enter a number: <input type="text" value="2"  id="PrimeNumber"><br>
    </form>
    <button id="CheckNumber">Check Number</button>
  </body>
</html>
```

3- Finally, let's glue everything together using javascript. So, edit wasm-is-prime/www/index.js to import to read user input and calls the CheckNumber function in rust:

```
import * as wasm from "wasm-is-prime";
const textbox1= document.getElementById("PrimeNumber");
document.getElementById("CheckNumber").addEventListener("click", event => {
     wasm.check_prime(textbox1.value);
});
```

4- Remember that you need to rebuild your project and then run it using npm (you can find more information about npm at https://www.npmjs.com/

```
     wasm-pack build // wasm-is-prime directory
     npm install // www directory
     npm run start // www directory
```

5- The app should be up and running now. We can pass data from Javascript to Rust and use it to perform work. Even though is_prime only checks even/odd, you can still test the general control flow of the program.

6- The alert messages are not particularly professional. Let's pass the answer from Rust back to Javascript and pretty print it. First modify src/lib.rs so that check_prime returns 1 if the input is prime and 0 otherwise. The function signature of check_prime should now be:

```
#[wasm_bindgen]
pub fn check_prime(s: &JsValue) -> u32 {
    let input: String = s.as_string().unwrap();
    match input.parse::<u32>() {
        Ok(num) => {
            if is_prime(num) == 1 {
                alert("Input is Prime");
                return 1;
            } else {
                alert("Input is NOT Prime");
                return 0;
            }
        }
        Err(_) => {
            alert(&format!("Couldn't parse {}", input));
            return 0;
        },
    }
}
```

7- Now we'll create a canvas to pretty print our answer. Add the following lines to www/index.html:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <title>Is Prime App!</title>
</head>
<body>
  <noscript>This page contains webassembly and javascript content,
    please enable javascript in your browser.</noscript>
  <script src="./bootstrap.js"></script>
  <form>
    Enter a number: <input type="text" value="2" id="PrimeNumber"><br>
  </form>
  <button id="CheckNumber">Check Number</button>
  <canvas id="board" width="100" height="100" style="border:1px solid grey">
    Your browser doesn't support html5 canvas
  </canvas>
</body>
</html>
```

8- Finally, we'll accept the input from check_prime and use it to draw a fancy result. First, add the fancy drawAnswer function to your www/index.js file:

```
//...

const canvas = document.getElementById("board");
const ctx = canvas.getContext("2d");
function drawAnswer(yn) {
    ctx.beginPath();
    let xpos = 50;
    let ypos = 50;
    ctx.arc(xpos, ypos, 25, 0, 2 * Math.PI);
    if (yn == 0) {
        ctx.fillStyle = 'red';
    } else {
        ctx.fillStyle = 'green';
    }
    ctx.fill();
    ctx.font = '24pt Calibri';
    ctx.fillStyle = 'white';
    ctx.textAlign = 'center';
    if (yn == 0) {
        ctx.fillText('?', xpos, ypos); // Symbol on red.
    } else {
        ctx.fillText('?', xpos, ypos); // Symbol on green.
    }

}
```

9- Now all we have to do is change the event handler to display the result:

```
import * as wasm from "wasm-is-prime";
const textbox1 = document.getElementById("PrimeNumber");
```

```
document.getElementById("CheckNumber").addEventListener("click", event => {
    const answer = wasm.check_prime(textbox1.value);
    drawAnswer(answer);
});

//... (*code from previous step)
```

10- This should show a red circle with a question mark in it when an even number is entered and a green circle when an odd number is entered. However, it does not apply a primality test. All you need to add is the code in lib.rs that checks if the user input is prime or not. **(Hint: you can use any crate that performs primality tests which you may have used in Lab1).**

11- As a final flourish, please change the question mark in the circle to a check-mark (✅) when the input is prime and an x-mark (❌) when it isn't. (Note: You can copy and paste the ✅ and ❌ from this text to your code).

DEMO this deliverable to the lab instructor.