JS-CMP Beta Test Plan

1. Introduction

The beta version of **JS-CMP** aims to validate the core functionality of transpiling JavaScript to C++ and compiling it into a binary. This test plan outlines the key features to be tested, the testing environment, user journeys, and success criteria. It also includes instructions for testers and a structured approach to reporting issues and incorporating feedback.

---

2. Objectives

- Validate the core functionality of transpiling JavaScript to C++ and compiling it into a binary.
- Gather user feedback to identify issues, improve usability, and ensure the project meets user expectations.
- Ensure that all implemented features work as expected and that no critical bugs are present.

---

3. Scope

3.1 Key Features to Test

All fully implemented features from the ECMA-262 specification that has been implemented can be found here. This includes:

- Transpiling JavaScript to C++.
- Compiling the generated C++ code into a binary.
- Executing the binary and verifying its behavior.

3.2 Testing Environment

- **Supported Platforms:** Linux and macOS.
- **Prerequisites:**
  - A C++ compiler (e.g., `g++` (version: 13.2.0), `clang++` (version: 14.2.0_1)).
  - `git` for cloning the repository.
  - `cmake` (version: 3.31.4) for building the project.
  - Boost libraries installed. (version: 1.8).
  - Doxygen (version: 1.13.2)
  - Submodules initialized (`git submodule update --init --recursive`).

3.3 User Roles

- **Developers:** Testers will primarily be developers who will test the transpilation and compilation process.
- **End-Users:** Testers will also include end-users who will execute the compiled binaries and provide feedback on usability.

---

4. Testing Procedure

4.1 Setup Instructions

1. **Clone the Repository:**

   ```
   git clone https://github.com/JS-CMP/JS-CMP.git
   cd JS-CMP
   ```

2. **Initialize Submodules:**

   ```
   git submodule update --init --recursive
   ```

3. **Install Dependencies:**
   - **Linux:**

     ```
     sudo apt install libboost-all-dev doxygen libicu-dev
     ```

   - **macOS:**

     ```
     brew install boost doxygen icu4c
     ```

4. **Build the Project:**

   ```
   cmake . && make
   ```

5. **Build the Documentation:**

   ```
   doxygen Doxyfile
   ```

4.2 Running Tests

1. **Transpile and Compile a JavaScript File:**

   ```
   ./js_cmp <input-file>.js
   ```

    - This command will transpile the JavaScript file to C++ and compile it into a binary named `<input-file>`.

2. **Execute the Binary:**

   ```
   ./<input-file>
   ```

4.3 Features to Test

4.3.1 Transpilation

- **Description:** Test the transpilation of JavaScript code to C++.
- **Example:**

  ```
  // input.js
  function greet(name) {
      console.log("Hello, " + name + "!");
  }
  greet("World");
  ```

    - Run:

      ```
      ./js_cmp input.js
      ```

    - Verify:
        - The command generates a `input.cpp` file.
        - The `input.cpp` file is syntactically correct and compiles without errors.

4.3.2 Compilation

- **Description:** Test the compilation of the transpiled C++ code into a binary.
- **Example:**
    - Use the same `input.js` file as above.
    - Run:

      ```
      ./js_cmp input.js
      ```

    - Verify:
        - The command generates a binary named `input`.
        - The binary executes without errors.

4.3.3 Execution

- **Description:** Test the execution of the compiled binary.
- **Example:**

- Use the same `input.js` file as above.
- Run:

  ```
  ./input
  ```

- Verify:
  - The binary executes and produces the expected output: `Hello, World!`.

4.3.4 Objects and Operators

- **Description:** Test the transpilation and execution of JavaScript code involving objects and various operators.
- **Examples:**

  Object Test:

  ```
  // object.js
  var person = {
      name: "Alice",
      age: 30,

  };
  console.log(person.name + " has " + person.age + "!")
  ```

  - Run:

    ```
    ./js_cmp object.js
    ```

  - Verify:
    - The command generates a `object` file.
    - The binary executes and produces the expected output: `Alice has 30!`.

4.3.5 Operation

Operator Test:

```
// operators.js
var a = 10;
var b = 5;
console.log("Addition: " + (a + b));
console.log("Subtraction: " + (a - b));
console.log("Multiplication: " + (a * b));
console.log("Division: " + (a / b));
console.log("Modulus: " + (a % b));
```

- Run:

```
./js_cmp operators.js
```

- Verify:
  - The command generates an `operators` file.
  - The binary executes and produces the expected output:

    ```
    Addition: 15
    Subtraction: 7
    Multiplication: 50
    Division: 2
    Modulus: 0
    ```

4.3.6 Mixed Types Test

```
// mixed_types.js
var num = 42;
var str = "The answer is: ";
console.log(str + num);
```

- Run:

  ```
  ./js_cmp mixed_types.js
  ```

- Verify:
  - The command generates a `mixed_types` binary file.
  - The binary executes and produces the expected output: `The answer is: 42`.

4.3.7 Usability

- **Description:** Evaluate the ease of use for developers when transpiling and compiling code.
- **Example:**
  - Test the clarity of error messages and the overall user experience.
  - Gather feedback on the usability of the `js_cmp` command and its output.

---

5. Reporting Issues

- Testers will report issues through a structured report, which will be submitted as issues on the GitHub repository: JS-CMP Issues.
- Include the following details in the report:
  - Description of the issue.
  - Steps to reproduce the issue.

    - Expected behavior.
    - Actual behavior.
    - Screenshots or logs, if applicable.

---

6. Success Criteria

- All core functionalities work as expected.
- No critical bugs are found.
- Positive user feedback on usability and functionality.
- Successful transpilation, compilation, and execution of test cases.

---

7. Deliverables

- **Beta Test Plan Document:** This document.
- **User Guide:** Instructions for setting up the testing environment and running tests.
- **Installation Instructions:** Detailed steps for installing dependencies and building the project.
- **Test Case Templates:** Structured templates for reporting issues and providing feedback.

---

8. Feedback Incorporation

- Feedback will be collected through GitHub issues.
- Issues will be prioritized and addressed in the main branch as they are fixed.
- Updates will be pushed to the repository to reflect changes based on feedback.

---

9. Security and Compliance

- Ensure that the transpiler and compiler do not leak memory or introduce security vulnerabilities.
- Test for memory safety and security in the generated C++ code.
- The project is licensed under GPLv3, and all contributions must comply with this license.