Risk Analysis

Introduction

This document outlines the risk analysis for the Syntaxiar project. It aims to identify potential risks in the development of our compiler and propose appropriate mitigation strategies to minimize project disruptions and ensure stable, correct, and reliable execution.

Detailed Risk Descriptions

1. Incorrect Code Generation

- What it means: Your compiler might generate wrong machine code even if the input source code is correct. This is often due to bugs in semantic analysis or code generation.
- Why it matters: Wrong output, program crashes, or completely unpredictable behavior can occur.
- Mitigation:

  Use step-by-step debugging to track where things go wrong.

  Perform unit testing on each compiler stage (especially semantic analysis and code generation).

---

2. Incomplete Error Handling

- What it means: If the compiler doesn't catch and report syntax or semantic errors properly, it could crash or ignore invalid input without telling the user.
- Why it matters: Users won't know why their code fails, leading to frustration and lack of trust in the compiler.
- Mitigation:

Add proper semantic validations (e.g., undeclared variables, type mismatches).

Test the compiler with invalid inputs to see how it handles them.

---

3. Intermediate Representation (IR) Misalignment

- What it means: The IR is a bridge between parsing and code generation. If the IR isn't correctly built from the parser's output, the final machine code will be flawed.
- Why it matters: IR issues can silently break the compilation flow.
- Mitigation:

  Write tests specifically for the IR generation module.

  Ensure consistency by reviewing how each parsing rule transforms into IR.

---

4. Toolchain Compatibility Issues

- What it means: Your compiler might produce output that isn't compatible with tools like RARS or RISCV GCC.
- Why it matters: Even if your compiler seems correct, it might not run or be accepted by external tools.
- Mitigation:

  Test early and often using both RARS and RISCV GCC.

  Build a test suite that includes running your compiled output on target tools.

---

5. File I/O and Format Errors

- What it means: Issues in reading the source code file or writing the output (tokens, IR, machine code) due to incorrect formatting or parsing.
- Why it matters: It may corrupt input/output or stop compilation midway.

- Mitigation:

    Stick to standard formats (e.g., .txt).

    Include exception handling for file operations.

---

6. Module Integration Failures

- What it means: When you combine different modules like the lexer, parser, semantic analyzer, IR generator, and code generator, they might not fit together well due to mismatched assumptions or data structures.

- Why it matters: Even if individual parts work, the compiler could fail as a whole.

- Mitigation:

    Use integration testing after each major stage (lexer → parser → IR, etc.).

    Maintain logs or trace files to track how one module passes control/data to the next.

---

8. Version Control Conflicts

- What it means: When many people are working on different modules and commit to Git, merge conflicts can overwrite or duplicate work.

- Why it matters: You might lose important code or introduce bugs by accident.

- Mitigation:

    - Adopt a clear Git branching strategy (e.g., one branch per feature)/Folders per contributor

    - Encourage frequent pulls from the main branch to stay updated.

---

9. Platform Dependency

- What it means: compiler might behave differently on different systems (Linux vs Windows) due to file path formats, system calls, or library behavior.
- Why it matters: Portability issues reduce usability across platforms.
- Mitigation:
    - Build and test on multiple platforms if possible.
    - Use cross-platform libraries and avoid OS-specific assumptions.

---

10. Limited Error Feedback to User

- What it means: If error messages are vague or non-existent, users won't know what went wrong or how to fix it.
- Why it matters: Poor user experience and loss of trust in the tool.
- Mitigation:

  Give clear, specific messages (e.g., "Line 4: Unexpected token ';' after expression").

  Highlight line numbers and even token details in errors.

---

11. Lack of Performance Optimization

- What it means: Compiler might work but take a long time due to inefficient algorithms or redundant passes.
- Why it matters: Performance matters especially when compiling large files.
- Mitigation:

  Use profiling tools to find slow parts of your code.

  Optimize critical paths like parsing or code generation.

Risk Analysis Matrix

Impact-From scale of 1 to 5 classified from low to severe
Likelihood-from scale of 1-3 classified as low to high

Risk	Likelihood (L)	Impact (I)	Risk Level (L x I)	Mitigation
Incorrect Code Generation	2 (Medium)	5 (Severe)	10 (high)	Thorough testing and code reviews
Incomplete Error Handling	3 (High)	4 (High)	12 (High)	Add semantic checks and test invalid inputs
IR Misalignment	2 (Low)	4 (High)	8 (Medium)	Test IR module independently
Toolchain Compatibility Issues	1 (Low)	5 (Severe)	5 (low)	Test early with both RARS and GCC
File I/O and Format Issues	2 (Medium)	3 (Medium)	6 (Medium)	Validate all inputs and use exception handling
Module Integration Failures	3 (High)	4 (High)	12 (High)	Integration testing after each phase
Poor performance	3 (High)	2 (Medium)	6 (Medium)	Use profiling tools to find slow parts of your code.