Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Department of Computer Science

Software Engineering Orientation

Bachelor thesis

2024

Functional language compiler to WebAssembly

Requirements specification

Noah Godel

Supervisors: Jacques Supcik
Serge Ayer

Experts: Baptiste Wicht
Valentin Bourqui

Fribourg, 5 June 2024

Version 1.0

Hes·so

Table of versions

Version	Date	Modifications
1.0	5.6.2024	Minor corrections and final version.
0.2	30.5.2024	Reduce the scope of the project to a subset of an existing functional language and direct mentions to LLVM.
0.1	27.5.2024	First version

Contents

1. Context 1
2. Objectives 3
3. Tasks 4
4. Planning 6

Section 1
Context

The functional programming paradigm offers significant advantages for certain types of problems, such as data transformations, parallel processing, and mathematical computations. However, it has limitations, and there are many use cases where imperative or object-oriented programming paradigms are more suitable. Ideally, developers should be able to leverage the strengths of different programming paradigms within the same codebase, using the most appropriate approach for each part of the project. Unfortunately, integrating functional languages into existing codebases written in other programming languages can be challenging.

WebAssembly (Wasm) is a portable and high-performance bytecode format designed to execute code at near-native speeds. It allows code written in various programming languages like C, C++, Rust, and others to be compiled to Wasm bytecode, which can then run in environments such as web browsers and Wasm runtimes like Wasmer.

By developing a compiler for a functional programming language, or in this case, a subset of an already existing one, that compiles to Wasm, we can combine the benefits of functional programming with the performance and portability of Wasm. This approach would enable seamless integration and embedding of high-performance functional code into existing codebases written in different languages. Developers could utilize the strengths of functional programming for specific parts of their projects, while leveraging other paradigms for the remaining codebase.

Embedding is the process of integrating code written in one programming language into a codebase written in another language. In this project, we aim to demonstrate the embedding of the new functional programming language compiled to Wasm into existing codebases written in other languages. This will showcase the interoperability and potential for combining different programming paradigms within the same project. Figure 1 illustrates the concept of embedding a Wasm module into a codebase.

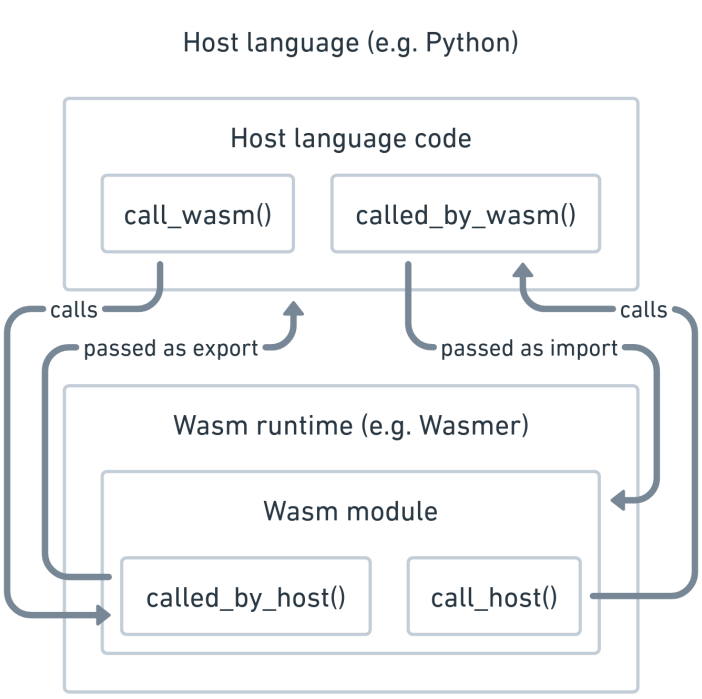

Figure 1: Illustration of embedding a Wasm module into a codebase.

Section 2
Objectives

Upon the completion of the project, the following objectives will be achieved:

- **Functional Programming Language Specification**: A specification of a functional programming language that is a subset of an existing functional language, tailored for efficient compilation to Wasm bytecode and seamless embedding into existing codebases. A subset of the standard library will be defined to support the language features.
- **Functioning Compiler**: A fully operational compiler capable of translating the defined functional programming language into efficient Wasm bytecode, enabling high-performance execution across various environments and platforms. The compiled code should be able to seamlessly interact with other programming languages its embedded into.
- **Language Documentation**: A documentation detailing the usage and development of the new functional programming language, including examples, and reference materials to facilitate learning and adoption by developers. Examples of embedding the language into existing codebases of different languages will be provided.

By achieving these objectives, the project will deliver a well-defined functional programming language optimized for Wasm execution, along with a functioning compiler, documentation, and embedding demonstrations. While not production-ready after the 7-week timeline, the project will serve as a proof of concept and a foundation for potential further development.

Section 3
Tasks

1. **Define Language Specification**
   1.1. Conduct research on existing functional programming languages and choose a suitable language to base the new language on.
      - *Deliverable*: A chapter in the project report detailing the chosen language, the subset of features to include, and the modifications required for efficient compilation to Wasm.
      - *Estimated workload*: 2 days
   1.2. Design the requirements and structure for a minimal standard library.
      - *Deliverable*: A chapter in the project report outlining the standard library features that will be implemented.
      - *Estimated workload*: 1 day
2. **Develop Compiler**
   2.1. Research different code generation strategies and tools for compiling to Wasm and choose the most suitable approach.
      - *Deliverable*: A chapter in the project report detailing the choice of code generation strategy and tools and a small proof of concept.
      - *Estimated workload*: 2 days
   2.2. Implement a minimal viable compiler (lexer, parser and code generation) for the new functional language with limited features.
      - *Deliverable*: A working compiler that can generate simple Wasm bytecode from the input language and a chapter in the project report detailing the implementation.
      - *Estimated workload*: 5 days
   2.3. Implement the remaining language features and optimizations to complete the compiler.
      - *Deliverable*: A fully functional compiler capable of translating the entire language subset to Wasm bytecode.
      - *Estimated workload*: 6 days
   2.4. Implement a simple standard library for the language.
      - *Deliverable*: A working standard library that supports the language features and a chapter in the project report.
      - *Estimated workload*: 3 days

3. **Testing and Validation**
   3.1. Develop an automated test suite to validate the correctness of the compiler and the Wasm bytecode.
   - *Deliverable*: Working test suite and a chapter in the project report detailing the testing strategy.
   - *Estimated workload*: 3 days

   3.2. Perform testing of code embedding and execution in different environments and platforms.
   - *Deliverable*: A chapter in the project report detailing the testing results and validation.
   - *Estimated workload*: 2 days

   3.3. Conduct performance benchmarking and analysis to evaluate the efficiency of the compiled code.
   - *Deliverable*: A chapter in the project report detailing the benchmarking methodology, results, and analysis.
   - *Estimated workload*: 1 day

4. **Create Language Documentation**
   4.1. Develop examples to facilitate learning and adoption of the language and its embedded use cases.
   - *Deliverable*: A set of examples demonstrating the language features and embedding capabilities.
   - *Estimated workload*: 3 days

   4.2. Draft a documentation covering the language syntax, semantics, features, and a reference for the standard library.
   - *Deliverable*: A small documentation of the language and standard library.
   - *Estimated workload*: 2 days

5. **Project Documentation**
   5.1. Write a requirements specification document outlining the context, objectives, tasks, and planning for the project.
   - *Deliverable*: A detailed requirements specification document.
   - *Estimated workload*: 4 days

   5.2. Prepare a detailed project report documenting the design, implementation, and evaluation of the language and compiler.
   - *Deliverable*: A comprehensive project report.
   - *Estimated workload*: 8 days (ongoing throughout the project, included in the tasks themselves)

Section 4
Planning

Table 1 illustrates the project timeline and tasks for the 7-week duration.

Week	Description	Deliverables
Week 1	Write the requirements specification document and start the project report.	The requirements specification document.
Week 2	Define the language specification, design the standard library and research code generation strategies.	The chapters in the project report detailing the language specification, standard library design, and code generation strategy.
Week 3	Implement the lexer, parser, and code generation for a minimal viable compiler.	A working compiler that can generate simple Wasm bytecode and a chapter in the project report detailing the implementation.
Week 4	Implement some remaining language features.	A functional compiler capable of translating a big part of the language to Wasm bytecode.
Week 5	Finish the compiler implementation and implement a simple standard library.	A fully functional compiler capable of translating the entire language subset to Wasm bytecode and a chapter in the project report detailing the standard library implementation.
Week 6	Develop an automated test suite, test code embedding and execution, and conduct performance benchmarking.	A chapter in the project report detailing the testing strategy, results, and benchmarking methodology, results, and analysis.
Week 7	Draft comprehensive language documentation and examples.	A chapter in the project report detailing the documentation structure and content, and examples for learning and adoption.

Table 1: Project timeline and tasks.

The project will be divided into 7 weeks, with each week focusing on specific tasks and deliverables. The timeline is designed to ensure a structured and organized approach to the project, allowing for the completion of the defined objectives within the allocated time frame.

The project will follow an iterative development process, with continuous testing and validation (where appropriate) to ensure the quality and correctness of the compiler and language implementation.

Project management tools

The project will utilize git for version control, allowing for tracking changes to the codebase. Gitlab issues will be used to manage tasks, and track progress. An issue will be created for each task, detailing the task description, estimated workload, and deadline.

Figure 2 shows the Gantt chart representing the project timeline and tasks.

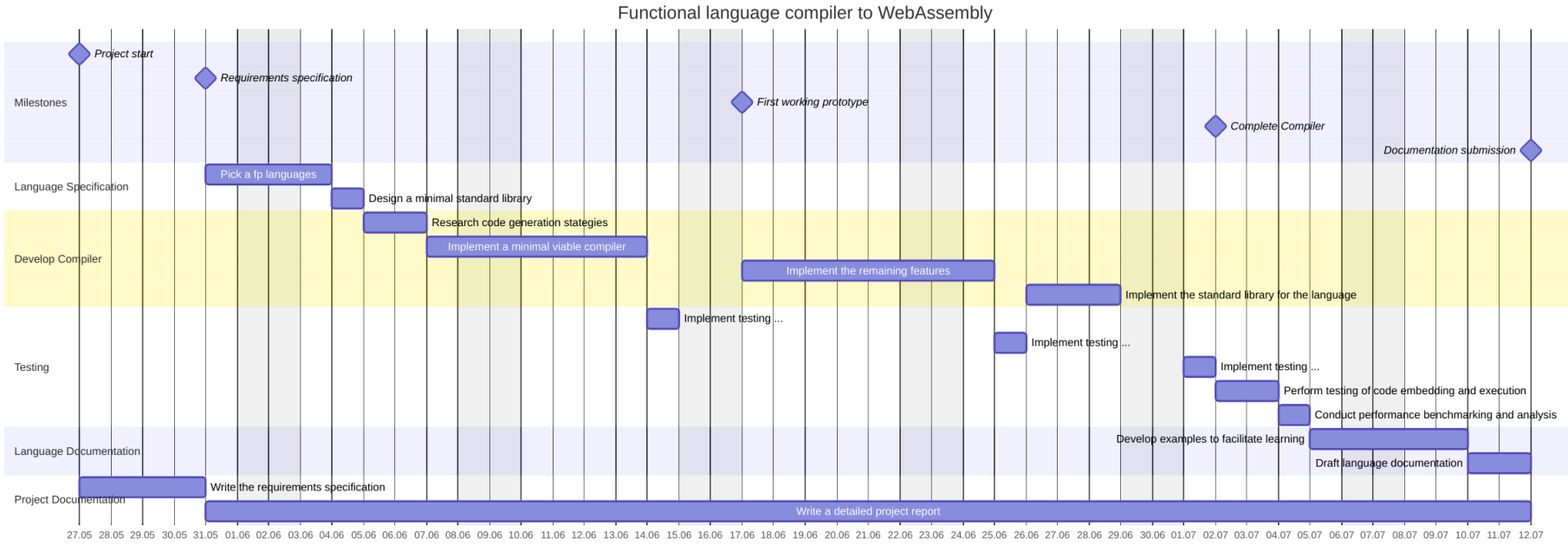

Figure 2: Gantt chart showing the project timeline and tasks.