DTU Compute
Department of Applied Mathematics and Computer Science

# Design and Implementation of a WebAssembly Compiler Back-End
for the High-Level Programming Language Hygge

Master Thesis

Design and Implementation of a WebAssembly Compiler Back-End
for the High-Level Programming Language Hygge

Master Thesis
Feb, 2024

By
Troels Lund

Cover photo: AI generated image, 2024
Published by: DTU, Department of Applied Mathematics and Computer Science, Richard Petersens Plads, Bygning 324, 2800 Kgs. Lyngby Denmark
compute.dtu.dk

Approval

This thesis has been prepared over six months at the Section of Software Systems Engineering, Department of Applied Mathematics and Computer Science, at the Technical University of Denmark, DTU, in partial fulfilment for the degree Master of Science in Engineering, MSc Eng.

Troels Lund - s161791

[illegible]

Signature

12/2 2024

Date

Abstract

Compilers are tools that typically translate a high-level language to a lower-level language. This thesis delves into the process of designing and developing a compiler back-end that targets WebAssembly (Wasm). The front-end is reused from an existing implementation. The *source language* used by the compiler is called *Hygge*. The main objective of the thesis is to generate valid WebAssembly code. The generated code is furthermore optimized with various techniques to produce smaller, more efficient executables. The compiler that utilizes the proposed WebAssembly back-end is named HyggeWasm.

HyggeWasm supports a comprehensive set of language features, including conditional statements, pattern matching, recursive functions, closures, loops, arrays, and structs. Moreover, it offers several memory modes, including one that enables garbage collection and two system interfaces, where one enables universally executable binaries. Furthermore, does HyggeWasm support two different writing styles of WebAssembly.

The project has achieved its objectives with success and has been thoroughly tested with a considerable test suite.

Acknowledgements

Alceste Scalas, Associate Professor, Technical University of Denmark (DTU)
Thesis supervisor.

Radoslaw Jan Rowicki, PhD Student, DTU
Thesis co-supervisor.

Daniel Larsen, Consultant, Netcompany
Reviewer of this thesis.

Helene Zgaya, Security Systems Engineer, TDC Erhverv
Reviewer of this thesis.

Christian Nicolai Nielsen, MSc Civil Engineering, DTU
Creator of this thesis template.

I thank my supervisors, Alceste Scalas and Radoslaw Jan Rowicki, for their willingness to engage in numerous meetings throughout the project. Their support was instrumental, as they inspired me with ideas, provided valuable feedback, and engaged in discussions on various solutions, contributing significantly to the success of my work.

I extend my thanks to my wonderful friends for taking the time to review my work and ensuring its clarity and understanding.

Finally, I want to express my heartfelt gratitude to my girlfriend, Emilie Elsvad, for her unwavering moral support.

Contents

Preface . . . ii
Abstract . . . iii
Acknowledgements . . . iv

1 Introduction 1
1.1 Prior Knowledge . . . 1
1.2 Outline . . . 3
1.3 Problem statement . . . 4
1.4 Methodology . . . 4

2 Background 5
2.1 Introduction to the Hygge programming language . . . 5
2.1.1 Phases of the Hygge compiler . . . 5
2.1.2 Syntax of Hygge . . . 6
2.1.3 Type system . . . 6
2.1.4 From Hygge program to type annotated AST . . . 6
2.1.5 Examples of Hygge programs . . . 7
2.2 Introduction to WebAssembly . . . 9
2.2.1 WebAssembly design goals . . . 10
2.2.2 Existing compiler toolchains for WebAssembly . . . 10
2.2.3 WebAssembly data types . . . 11
2.2.4 Core concepts . . . 11
2.2.5 Semantic phases . . . 11
2.2.6 Anatomy of a WebAssembly program . . . 12
2.2.7 Execution model . . . 13
2.2.8 Stack-based ISA vs register-based ISA . . . 13
2.2.9 Virtual ISA . . . 14
2.2.10 WebAssembly text format (WAT) . . . 14
2.2.11 Application Binary Interfaces (ABI) . . . 16
2.2.12 WebAssembly Runtime's . . . 16
2.3 Summary . . . 17

3 Design 18
3.1 Learning and Development tool . . . 18
3.1.1 Designing effective debugging experience . . . 18
3.1.2 User interface . . . 19
3.1.3 HyggeWasm runtime . . . 19
3.2 WAT Generation Framework (WGF) . . . 19
3.2.1 Design research . . . 20
3.2.2 Intermediate representation and API . . . 20
3.2.3 Producing the textual format . . . 21
3.3 HyggeWasm Compiler back-end . . . 21
3.3.1 Phases of code generation . . . 21
3.3.2 Compilation target . . . 22
3.3.3 Compiler modes . . . 23
3.3.4 Memory management and operation modes . . . 24
3.3.5 Writing Style modes . . . 27

3.3.6 Input/Output and System interface modes 28
3.3.7 Entry point 29
3.3.8 Program termination 29
3.3.9 Label naming 30
3.3.10 Function types 30
3.4 Summary 30

4 Implementation **32**
4.1 Development and learning tool 32
4.1.1 Loading WebAssembly modules 32
4.1.2 Implementation of the HyggeWasm runtime in TypeScript 33
4.1.3 WASI limitations 36
4.1.4 User Interface 36
4.2 WAT Generation Framework (WGF) 36
4.2.1 Instructions 36
4.2.2 Module 37
4.2.3 Produce textual format 38
4.3 HyggeWasm Compiler back-end 39
4.3.1 Solution structure of the HyggeWasm Compiler 39
4.3.2 WasmTime driver 40
4.3.3 Implementation of the HyggeWasm runtime in C# 40
4.3.4 Code generation environment 42
4.3.5 Stack management 42
4.3.6 Language features 44
4.4 Summary 78

5 Bringing garbage collection to HyggeWasm **79**
5.1 Memory layout 79
5.2 Introducing heap types 79
5.3 New reference instructions 79
5.4 Code generation 80
5.4.1 Arrays 80
5.4.2 Structs 82
5.5 Example of closure 83
5.6 Experimenting with the garbage collector 84
5.7 Limitations of the heap memory mode implementation 85
5.8 Capabilities Overview 85
5.9 Summary 85

6 Optimizations **86**
6.1 Optimizing the reading and writing of local variables 86
6.2 Dead-code Elimination 86
6.3 Special case for dead-code elimination 87
6.4 Constant folding 88
6.5 Branch-level tree shaking 90
6.6 Summary 90

7 Evaluation **91**
7.1 Internal vs external memory mode 91
7.2 Assembler Support 91
7.3 Testing 91

7.4 Optimizations 92
7.4.1 The impact of optimization 93
7.5 Summary 94

8 Future work 95

9 Conclusion 97

Bibliography 98

10 Acronyms 102

A Requirements specifications 104
A.1 Code generation and Language features - Requirements 104
A.2 Developer and learning tools - Requirements 107

B User interface - Early wireframe 108

C User interface 110

D Manual for learning and development tool 111
D.1 Software Version Requirements 111
D.2 Run the learning and development tool locally 111
D.3 Use the learning and development tool 111
D.4 Debug program 115

E Manual for using the CLI 117
E.1 Run '.wat'-file 117
E.2 Run test suite 117
E.3 Compile Hygge program 117

F UTF-16 Strings 118

G Hygge System interface specification 120
G.1 Interface 120

H Optimisation data set 122

I Visual studio code configurations 123

J Module API 126

K 02247 Compiler Construction - Project Report 128

L Function restructuring and organization 129

M Example of Internal bump allocation 130

N Unrealized Endeavors 131
N.1 Code generation exclusively for the linear writing style of WAT 131

O Source code 133

1 Introduction

Compilers are essential tools designed to translate a *source language* into a *target language*, typically translating from a higher-level language to a lower-level language. Compilers allow software engineers, computer scientists, and developers to write programs in high-level languages. Programs written in a high-level language are far easier for humans to read and understand than low-level machine instructions or byte code. This allows programmers to solve problems on a higher abstraction level and hide unnecessary details, thus significantly simplifying the development process.

Compiler construction is the study of designing, building, and implementing compilers. Compiler construction is a very successful branch of computer science, partly because it is a very well-defined problem within the field of computer science[1, p. 4]. As a software engineer, the motivation for studying compiler construction can be many. One reason may be that understanding how a programming language is implemented and relates to low-level instructions will allow them to write better and more efficient code.

The primary objective of this thesis is to design and implement a compiler back-end targeting WebAssembly (abbreviated Wasm). A programming language called *Hygge* will be the *source language* of the compiler. *Hygge* is a high-level programming language utilized in teaching compiler construction at DTU. To streamline the process, the preceding stages of the compilation related to the front-end of the compiler, which originate from an existing compiler called *Hyggec*, will be repurposed. Originally, *Hyggec* featured a back-end that generated code for a *register-based* Instruction Set Architecture (ISA) known as *RISC-V*[2]. Further details will be expounded upon in Chapter 2.

The motivation for using WebAssembly in this project is that it fundamentally differs from *RISC-V*[3], which will be a considerable challenge when designing the new back-end. Moreover, WebAssembly is a relatively new technology that is evolving fast. WebAssembly's efficiency and performance, combined with its integration with existing web technologies, make it a technology that can unlock many new applications and innovations of the web[4][5]. Given these promising prospects, numerous compilers have added WebAssembly as a *target language* since its creation.The details of WebAssembly and how it differs from *RISC-V* will be described in section 2.2.

A secondary objective of the project is to utilize its discoveries to teach students about compiler design principles and code generation for modern stack-based assembly language. This has influenced the design of the compiler back-end and the project as a whole. One notable impact of this is that the back-end produces a human-readable textual format, known as WebAssembly Text Format (WAT), rather than the binary format of WebAssembly. Furthermore, this lead to the development of an educational tool. The educational tool allows users to easily load, execute, and debug a binary WebAssembly module in a web browser alongside a runtime to handle Input/Output (I/O) and memory allocation.

1.1 Prior Knowledge

The compiler will be implemented in *F#*[6]; therefore, it will be a great advantage if the reader is familiar with functional programming and concepts related to functional programming. The reader does not need prior knowledge of WebAssembly, as this will be described in detail. However, the reader is assumed to be familiar with compiler construction and programming language design.

1.2 Outline

Chapter 1 - Introduction:
The rest of the Introduction will address the project's goal as a set of objectives.

Chapter 2 - Background:
Background provides the context and research foundation, including the core understanding of WebAssembly and the *source language* Hygge.

Chapter 3 - Design:
Design focuses primarily on general design of the compiler and techniques used in code generation. Additionally, design of memory layout, operation modes of the compiler, and of the WAT Generation Freamework (WGF) intermediate representation (IR) will also be described in this section.

Chapter 4 - Implementation:
Implementation delves into implementation details, illustrated with examples of both source code and the produced target code.

Chapter 5 - Bringing garbage collection to HyggeWasm:
The chapter details how Garbage Collection is enabled in a separate operation mode using a new memory model along with new WebAssembly primitives.

Chapter 6 - Optimizations:
Optimizations examines what optimizations can be applied to the Intermediate Representation (IR) to produce more effective code and how this is implemented in practice.

Chapter 7 - Evaluation:
Evaluation assesses the impact of the optimization phase on the generated code.

Chapter 8 - Future work:
Future work outlines the project's future work, including potential improvements and known issues.

Chapter 9 - Conclusion:
Conclusion summarizes the primary findings and results of the project, providing a concise overview of what has been achieved. This will be held up against the problem statement described in Section 1.3.

1.3 Problem statement

The problem addressed by this thesis is formulated as the questions below. Chapter 9 will evaluate the extent to which they have been addressed.

P1 How can high-level programming language features of Hygge be synthesized to the low-level constructs found in the *target language* of WebAssembly (Wasm)?

P2 Are there any specific limitations or challenges in the Hygge-to-Wasm compilation process, and what are the potential solutions or workarounds?

P3 How can the WebAssembly code be optimized, and how does the optimized code compare to the non-optimized version?

1.4 Methodology

At the project's beginning, it was carefully considered which technologies would be used for the compiler, including the choice of programming language. The original hygge compiler was written in the functional programming language *F#*. The implementation was fairly easy to understand and thus also to extend. Hence, it was deemed unnecessary to change the implementation language. Additionally, *F#* is particularly well-suited for implementing interpreters and compilers due to its strong type system and pattern-matching capabilities[7, p. VI]. These features allow for a more concise and clear implementation compared to languages lacking such attributes.

For this project, the practices of Test-Driven Development (TDD) were utilized. Tests were written before implementing new features, ensuring that introducing new features did not break any existing features. Each language feature has been developed incrementally during the implementation, with each iteration building upon the previous foundation. The Learning and Development tool has been used to assess the compiled code and therefore has been vital in the project's implementation phase. Section 3.1 will address the Learning and Development tool.

2 Background

This section will present the background of the thesis by examining the *source language*, *Hygge* and the *target language*, *WebAssembly*.

2.1 Introduction to the Hygge programming language

The *Hygge* programming language is the *source language* of the Hygge compiler, *HyggeC*. *Hygge* is designed by Alceste Scalas for educational purposes. The language is used in the graduate-level Compiler Construction course at DTU. In the course, an initial version of a Hygge compiler was provided called *Hygge0*. It supplied some basic functionality that the students could then extend through the course. Such an extended version of *HyggeC* serves as the starting point of this project.

2.1.1 Phases of the Hygge compiler

Multiple phases are involved in *Hyggec*, which are represented in Figure 2.1. The front-end of *Hyggec* includes the phases of Lexing, Parsing, and Type Checking.

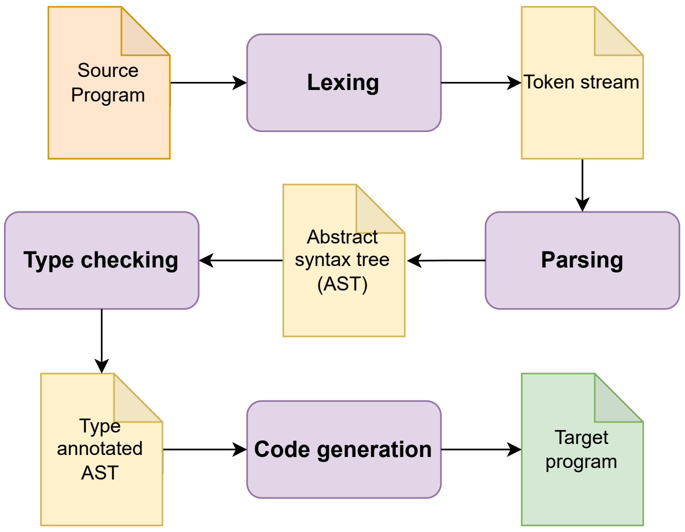


Figure 2.1: Phases of the Hygge compiler

This thesis reuses the front-end of *HyggeC*. The back-end is completely reimplemented to synthesize WebAssembly.

The first phase called *lexing*, produces a stream of individual tokens based on the raw text input[1, p. 1-20]. The token stream is then forwarded to the *parser* phase that produces an Abstract Syntax Tree (AST). The lexer and parser are automatically generated in *HyggeC*. The *lexer* is generated based on definitions of the individual tokens used while *parser* is based on a Context-Free Grammar (CFG). The CFG describes how a syntactically valid program is formed[1, p. 8].

After this, *type checking* is performed. *Type checking* verifies and enforces type constraints of the program. This phase results in a type annotated AST. The type annotated AST is the input of the *code generation* phase. The *code generation* phase synthesizes the *target language*. *HyggeC* produced *RISC-V* assembly. This thesis will focus on reconstructing the *code generation* phase to fit a new *target language*.

Appendix A.1 contains all language features of this new version of the Hygge compiler, from here on denoted as *HyggeWasm*. The language features were picked during the project's planning phase and were constructed as a requirement specification in the MoSCoW format.

2.1.2 Syntax of Hygge

In Hygge, a program consists of only one expression that subsequently can hold sub-expressions. Hygge is, therefore, expression-oriented and has no distinction between statements and expressions. This style is often used for functional languages, where most imperative languages distinguish between expressions and statements and have different rules about how the two can be used. The syntax of Hygge is inspired by the *ML* family of languages [7, p. 8] and is therefore comparable to the syntax found in languages like *F#* and *Scala*. A simple example of a Hygge program can be seen in code snippet 2.2. Multiple examples of Hygge programs are shown in section 2.1.5. Hygge's formal syntax specification can be found in the course notes 02247 - Compiler Construction [8] and Appendix K.

```
// declare n as an integer and assign it the value 16
let n: int = 16;
// function to calculate the nth term of the Fibonacci sequence
fun fibRec(n: int): int = {
    if (n <= 1) then {
        n
    }
    else {
        fibRec(n - 1) + fibRec(n - 2)
    }
};
// print the result
println("The 16th term of the Fibonacci sequence is:");
println(fibRec(n))
```

Figure 2.2: Recursive function to calculate the nth term of the Fibonacci sequence

2.1.3 Type system

Hygge is a statically and strongly typed language and provides a rather basic typing system with subtyping. Hygge has primitive types such as *unit*, *int*, *bool*, and *float*. It also includes composite data types like *strings*, *structs*, *tuples*, *unions*, and *arrays*. Hygge programs can contain type declarations that can be type aliases or define composite data types. The type system uses *structural typing* as found in languages like *TypeScript*[9] when evaluating *structs*. A structural type system considers only the members of a type when comparing types. Functions can have multiple arguments but only one return value and are treated as first-class values that can be passed around and returned.

2.1.4 From Hygge program to type annotated AST

This section clarifies the relationship between the Hygge source program and a type-annotated AST. The code generation uses the type-annotated AST to create the target program. The type-annotated AST is therefore critical to comprehend to understand the content of this thesis.

To understand this, let's consider the example 2.3, which shows a simple Hygge program that asserts that $4 + 5$ equals 9. The type-annotated AST depicted in Figure 2.4 is the result of the compiler's processing of the input program illustrated in 2.3.

```
assert(4 + 5 = 9)
```

Figure 2.3: Simple assert program

The root node is the `Assertion` derived from the *assert* function invocation being the top-level expression of the program. The `Assertion-node` has the type of *unit* since the built-in *assert* function does not evaluate to a value. The assert function takes an argument; in this case, the argument consists of the expression $4 + 5 = 9$. Equality wraps around the addition in this expression, resulting in a node `Eq`. The `Eq` node has a right and left side and evaluates to a boolean value; the right side holds the integer value **9**, and the left holds the addition of 4 and 5. The leaf nodes marked with yellow are all literal values.

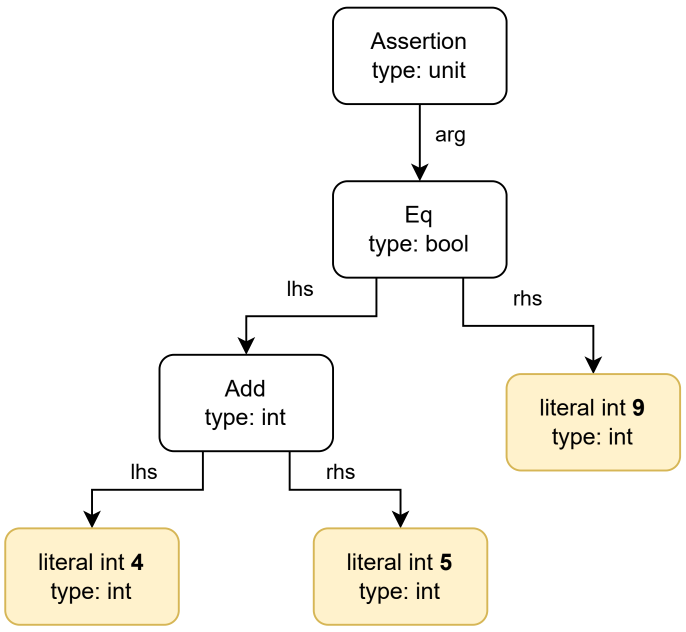


Figure 2.4: Exsample of the type annotated AST

2.1.5 Examples of Hygge programs

To exemplify the characteristics of the Hygge programming language, this section will present concise examples of programs utilizing the inherent features of Hygge.

2.1.5.1 Exemplifying closures, functions and structures

Example 2.5 is a Hygge program that implements an elementary calculator. A type declaration of the *struct* return by the *makeCal* function is created called *Cal*. The *Cal struct* contains four function pointers in this example. Each function pointer references to an anonymous function that performs an arithmetic operation between the *res* variable and an input variable, *v*. Each anonymous function is defined directly inside the returned *struct*. These functions capture their lexical environment and enclose the variable *res*; thus, the returned functions operate on the same encapsulated state in the *makeCal* function.

```
1  type Cal = struct {add: (int) -> int; // type declaration of 'Cal'
2                     sub: (int) -> int;
3                     mul: (int) -> int;
4                     div: (int) -> int};
5
6  fun makeCal(): Cal = {
7      let mutable res: int = 0;
8      // return structure with the anonymous functions
9      struct { add = fun (v: int) -> { res <- res + v };
10              sub = fun (v: int) -> { res <- res - v };
11              mul = fun (v: int) -> { res <- res * v };
12              div = fun (v: int) -> { res <- res / v } } : Cal
13 };
14
15 let c1: Cal = makeCal(); // creating first function instance
16 assert(c1.add(2) = 2); // do calculations with anonymous functions
17 assert(c1.add(4) = 6);
18 assert(c1.mul(2) = 12);
19
20 let c2: Cal = makeCal(); // creating second function instance
21 assert(c2.add(10) = 10);
22 assert(c2.div(2) = 5);
23 assert(c2.sub(2) = 3)
```

Figure 2.5: Implementation of Calculator in Hygge, using a closure capture state

makeCal is called in *line 15*, effectively creating a function instance. The functions of the returned structure are accessed and invoked with the dot notation, subsequently changing the *count* variable of that closure.

Then, in *line 20*, *makeCal* is called again, making a new function instance that exists independently of the other function instance.

2.1.5.2 Recursive function, mutable variables and arrays

Example 2.7 is a Hygge program that implements the recursive sorting algorithm *Quick sort*, showcasing Hygge's feature of recursive function declarations and recursive function calls. The *Quick sort* algorithm operates on an array of integer values, and Hygge has common capabilities of working with arrays. These can be seen in Table 2.6.

Hygge also supplies conditional statements and loop statements in the varieties *for*, *while*, and *do-while*. Variables are immutable unless declared *mutable* with the keyword as part of the let binder, similar to *F#*.

Array Operation	Expression
Array constructor	$array($length, initial data$)$
Access data at an index	$arrayElem($array reference, index$)$
Assigning a value to an element	$arrayElem($array reference, index$) \leftarrow$ new value
Obtain the length of an array	$arrayLength($array reference$)$

Figure 2.6: Array operations

```
fun partition (arr: array{int}, low: int, high: int): int = {
    // Choose the rightmost element as the pivot
    let pivot: int = arrayElem(arr, high);

    // Index of the smaller element
    let mutable i: int = low - 1;
    let mutable j: int = 0;

    for (j <- low; j < high; j++) {
        // If the current element is smaller than or equal to the pivot
        if (arrayElem(arr, j) <= pivot)
        then {
            // Swap arr[i] and arr[j]
            i <- i + 1;
            let temp: int = arrayElem(arr, i);
            arrayElem(arr, i) <- arrayElem(arr, j);
            arrayElem(arr, j) <- temp;
            ()
        }
        else {()}
    };

    // Swap arr[i+1] and arr[high] (pivot)
    let temp: int = arrayElem(arr, i + 1);
    arrayElem(arr, i + 1) <- arrayElem(arr, high);
    arrayElem(arr, high) <- temp;

    // Return the pivot index
    i + 1
};

fun quickSort(arr: array{int}, low: int, high: int): unit = {
    if (low < high)
    then {
        // Partition the array and get the pivot index
        let pivotIndex: int = partition(arr, low, high);

        // Recursively sort the subarrays on both sides of the pivot
        quickSort(arr, low, pivotIndex - 1);
        quickSort(arr, pivotIndex + 1, high);
        ()
    }
    else {()}
};
```

Figure 2.7: Quick sort implemented in Hygge

2.2 Introduction to WebAssembly

WebAssembly is a low-level bytecode format designed for efficient execution on the web, providing a platform-independent runtime environment. WebAssembly has gradually evolved since it was first announced in 2015, and it was later released as an Minimum Viable Product (MVP) in 2017 [10][11]. At this point, version 1.0 of WebAssembly has been implemented in all major browser engines[12], and the WebAssembly specification version 2.0 is being drafted[13]. WebAssembly is intended to be an effective compilation target for source languages like *C++*, *Rust*, and *Assemblyscript* [14]. Considering this, together with its high efficiency, WebAssembly opens a range of new opportunities in web development.

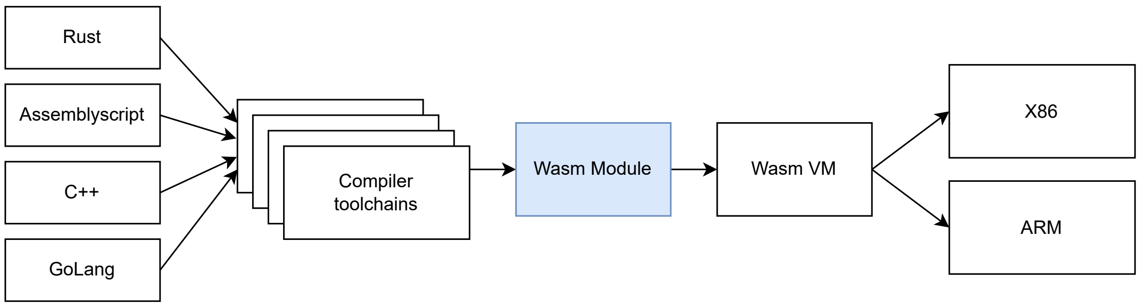


Figure 2.8: High-level architecture of WebAssembly framework

The WebAssembly bytecode is an intermediate language for a stack-based Virtual Machine (VM). The machine code is later produced by the VM based on the host system's hardware. The VM can either produce the machine code at runtime with a just-in-time (JIT) compiler or ahead-of-time (AOT), producing a file with the machine code itself, typically with a '.cwasm' extension. Figure 2.8 shows a top-level architecture of this.

2.2.1 WebAssembly design goals

The WebAssembly ISA was designed to take advantage of common hardware capabilities to deliver near-native execution speeds on a wide range of platforms. WebAssembly is defined formally to make it easy to reason about its behavior and subsequently adopted as a compilation target. As the name implies, Wasm is designed to execute and integrate with existing web platforms[15].

WebAssembly allows web developers to offload compute-intensive tasks to the VM while integrating fairly easily with *JavaScript*. WebAssembly also has applications outside the web. It can be executed in a memory-safe, sandboxed environment and is ideal for executing untrusted code[16], making it impossible for the untrusted code to access unauthorized data.

2.2.2 Existing compiler toolchains for WebAssembly

Two of the most used compiler toolchains for WebAssembly are *Emscripten*[17] and *Binaryen*[18].

Emscripten can compile languages using *LLVM* to WebAssembly. It generates outputs compatible with web environments and various WebAssembly runtimes. When *Emscripten* compiles for a web-based environment, it produces a binary WebAssembly module alongside a *JavaScript* file. The *JavaScript* bundle bridges the gap between APIs found in the source language that the host system must address due to the limitations of WebAssembly. *Emscripten* can also produce standalone Wasm modules.

Binaryen is a compiler and toolchain infrastructure library designed for WebAssembly. *Binaryen* is the toolchain behind languages like *Grain*[19] and *AssemblyScript*[20]. The latter has been a great source of inspiration for the project because it produces fairly compact Wasm modules that are easier to analyze and understand. This is because many languages compile extensive runtimes into the modules.

This project draws inspiration from both of these compiler toolchains. The rest of the report will occasionally refer to specific features inspired by the two toolchains.

2.2.3 WebAssembly data types

WebAssembly has four numeric value types: `i32` (32-bit integer), `i64` (64-bit integer), `f32` (32-bit floating point) and `f64` (64-bit floating point).

Integer values are not inherently signed or unsigned[21]. Instead, the bits are interpreted based on the instructions used on the values. Furthermore, Wasm provides a vector type that can be used with single instruction multiple data (SIMD) operations and two reference types, *funcref*, to reference functions and *externref* external data structures. This project uses only the 32-bit instructions of the ISA. More on this can be found in section 3.3.2.

The type *funcref* represents the infinite union of all references to functions, irrespective of their specific function types. The type *externref* denotes the infinite union of all references to objects held by the embedding environment. Objects can be passed into WebAssembly as this type. Reference types are opaque, indicating that their size and bit pattern remain unobservable.

2.2.4 Core concepts

In this section, we will explore some fundamental concepts of WebAssembly.

2.2.4.1 Traps

Certain instructions may trigger a *trap* during program execution, resulting in the immediate termination of the program. *Traps* are not manageable within WebAssembly code. However, they are communicated to the external host environment, where they can be handled.

2.2.4.2 Embedding environment

A WebAssembly module will always run embedded inside a host system since it runs in a VM. This could, for example, be a web browser or non-web environment, such as a server-side application. The terms embedding environment, host environment, or embedder, will be used interchangeably to refer to this host system running the WebAssembly VM.

2.2.4.3 Linear memory model

WebAssembly uses a *linear memory* model[22]. Therefore, the memory of a module is referred to as the *linear memory*. The *linear memory* is a mutable array of raw bytes. *linear memory* is divided into pages with the size of 64KiB. WebAssembly presently only allows for 32-bit addressing, meaning that the maximum amount a module can allocate is, $2^{16} \times 64KiB = 4GiB\ bytes$[23]. The memory has an initial size defined in the module but can be grown at runtime. Memory access outside the bounds of the memory will trigger a trap that will end program execution. Memory can, if exported, be accessed and manipulated by the embedder. When inspecting this from *JavaScript*, it will be represented as an *ArrayBuffer*[23].

2.2.5 Semantic phases

The semantics of WebAssembly can be split into three distinct phases. The WebAssembly specification outlines the phases in detail.[13, p. 4]

1. **Decoding:**
   In the decoding phase, the binary modules are translated into their corresponding abstract syntax, forming an internal representation of the module.

2. **Validation:**
   This phase examines various conditions to guarantee the module is well-formed. If all conditions are satisfied, the module can be considered valid. Notably, it conducts checks such as type validation of functions and control structures, and the sequences of instructions within these structures.

   A crucial aspect of this validation is confirming the operand stack is consistent.

3. **Execution:**
   Only a valid module can be executed. Execution can be further separated into two phases:

   a. **Instantiation:**
      A module instance represents a module, enclosing an internal state and an active execution stack. This process involves initializing global variables, memories, and tables. The module can also involve invoking a start function if specified in the module. Finally, the instantiation returns the instances of the module's exported components.

   b. **Invocation:**
      WebAssembly code can be executed when instantiating a module or invoking an exported function. Instantiation of a module and the invocation of functions are done by the *embedding environment*.

2.2.6 Anatomy of a WebAssembly program

A WebAssembly program consists of a module. The module is split into multiple sections. The module can be represented in the textual format WAT or as binary. As the back-end produces WAT, this section examines the module in this format. All sections of a WAT module are shown in Table 2.1.

Section	Description
`Type`	Declares function signatures
`Import`	Declares imports
`Function`	Functions in the module
`Table`	Used for indirection by storing references
`Memory`	Linear memory for the module
`Global`	Declaration of global variables
`Export`	All exported functions to the host
`Start`	Index to the function to be invoked at module initialization
`Element`	Initializes imported modules
`Custom`	Any other kinds of custom data
`Data`	Data to be loaded in the *linear memory* during initialization

Table 2.1: Sections in a WAT WebAssembly module

2.2.6.1 Sections

This section will describe each section in the WAT format. All sections are optional, and the module may be empty. The **Type** section declares function signatures used within the module. The function types can be defined independently or as part of the function declaration. The **Function** section has all functions, including their instructions. The **Global** section contains all global variables, which can be accessed from the entire module. The **Start** section defines a function to run at instantiation of the module. The **Memory** section defines an initial memory size and can optionally set an upper bound for memory growth. The **Import** section allows for importing functions, and data from outside the module. Every resource from outside the module should be declared here with the proper data types to be accessible within the module. Subsequently, the **Export** section defines everything the embedder can access within the module. This enables modules to regulate their interactions with the host environment precisely. The **Table** section contains tables. A table contains elements of a particular type. It can be dynamically mutated at runtime by the host and the instructions inside the module. The table lives outside of WebAssembly's *lin-*

ear memory and is completely separated. Tables can store values of a reference type that can be referred to by an index. Tables can be initialized through element segments found in the **Element** section. The **Data** section facilitates the initiation of memory segments. This resembles the functionality of the *.data* segment found in *RISC-V*.

2.2.7 Execution model

WebAssembly is a stack-based ISA. Instructions operate on a stack called the *operand stack*. A stack is a data structure that follows the Last In First Out (LIFO) principle, where elements are added (pushed) or removed (popped) from the top of the stack. The *operand stack* stores short-lived temporary values. To perform computations, instructions pop values from the *operand stack* as input, and push the result of the computation back to the *operand stack*. In other words, the *operand stack* can be conceptualized as a collection of unnamed registers that instructions can implicitly reference. An instruction that pops two `i32` values and pushes one back can be described as transformation: $[i32,\ i32] \rightarrow [i32]$. The WebAssembly specification [13] defines such transformation for all instructions. The validation rules of WebAssembly guarantee a consistent stack. For instance, if a result type is declared like $(result\ \ f32)$, the stack must conclude with precisely one $[f32]$. A violation of this will throw a type error, and the program can not be instantiated. WebAssembly has an *implicit call stack* that handles the execution flow. All functions can call each other, including recursively. The call stack is managed by the WebAssembly runtime and is not directly accessed or manipulated by WebAssembly code.

2.2.8 Stack-based ISA vs register-based ISA

A stack-based ISA functions fundamentally differently than a register-based ISA. A register-based ISA uses dedicated storage locations called registers. It enhances computational speed and reduces memory access overhead.

To illustrate these differences, two simple programs are provided, each performing the same computation. The example uses the register-based ISA *RISC-V* and WebAssembly. Both programs accept two operands and perform an addition operation. The programs can be seen in 2.9a and 2.9b.

```
li a0, 2  # Load immediate value 2 into register a0
li a1, 2  # Load immediate value 2 into register a1
add a0, a0, a1  # Add a0 and a1, store result in a0
```

(a) Simple RISC-V program that adds two numbers together

```
;; stack starts empty -> []
i32.const 2 ;; push 2 on stack -> [2]
i32.const 2 ;; push 2 on stack -> [2, 2]
i32.add ;; pop two elements and add them together -> [4]
```

(b) Simple WebAssembly program that adds two numbers together

Figure 2.9: Simple addition with RISC-V Vs WebAssembly

Register-based The register-based ISA will have to load data from memory into the registers to perform operations on that data. A key advantage is that when data is in registers, operations can be done very efficiently. Moreover, the result is saved in a target register. It is explicitly stated in each instruction which registers are used, as seen in the example 2.9a. The data is loaded into registers *a0* and *a1*, then the *add* instruction computes the sum of these and stores the result in *a0*.

Stack-based The WebAssembly instructions are, on the other hand, executed on a stack machine. The example program and how it is executed can be seen in 2.9b. At the start of program execution, the stack is empty. Then $i32.const\ 2$ is executed two times. Both push the number 2 onto the stack. The $i32.add$ instruction then pops two values from the stack and pushes the sum of the two operands to the stack, leaving the number 4.

2.2.9 Virtual ISA

As previously mentioned, the WebAssembly execution engine is implemented in software as a VM and is an abstraction on top of the actual hardware. An advantage of this is that the VM can be adapted to run on many different underlying hardware architectures; this is an important feature of WebAssembly since it should be able to run in browsers on many different devices, making everything compiled to WebAssembly essentially cross-platform. This is not a new idea. Many may remember Java's old slogan Write once, run anywhere (WORA), which was possible because of the Java Virtual Machine (JVM), which is also a stack-based machine.

2.2.10 WebAssembly text format (WAT)

Like other assembly languages, WebAssembly has a human-readable textual representation called WebAssembly Text Format (WAT). WAT allows for comments inside the code itself, both as single and multi-line comments. Comments are an important feature of WAT in this project since this allows for code explanations directly in the compiled code, facilitating a better understanding of the produced code.

WebAssembly in text form is usually stored in a '.wat'-file. The textual representation can then be transformed into the binary format of WebAssembly. The binary module is normally stored in a '.wasm'-file. The binary file can be executed in the WebAssembly VM.

2.2.10.1 Instructions

Instructions are syntactically grouped into plain and structured instructions. Table 2.2 contains commonly used plain instructions. The instructions *block*, *loop*, and *if* are structured instructions. These instructions serve to delimit and organize nested sequences of instructions, referred to as *blocks*. A structured instruction may consume an input and produce an output on the *operand stack* according to its annotated *block type*. The structured instructions are in table 2.3.

It is important that *blocks* are well-formed in agreement with the grammar of WebAssembly. These structured instructions must be correctly nested to ensure proper program structure. Otherwise, the validation phase will fail.

2.2.10.2 Structured control flow

Unlike more conventional assembly languages, WebAssembly does not support standard jump instruction. WebAssembly does not allow jumps to arbitrary label locations but uses structured control flow constructs. This means that jump is restricted within a control structure, such as a block or loop. To perform jumps, the branch instructions *br* (or other variations of a branch) are used inside a control structure, and it will dictate where the jump in code execution will go. The control structure can be referenced with a label associated with that block.

Instruction	Description
`nop`	No operation
`i32.const`	Push a 32-bit integer constant onto the stack
`f32.const`	Push a 32-bit floating-point constant onto the stack
`i32.add`	Add two 32-bit integers
`f32.add`	Add two 32-bit floating-point numbers.
`i32.sub`	Subtract two 32-bit integers
`i32.mul`	Multiply two 32-bit integers
`i32.div_s`	Signed division of two 32-bit integers
`i32.rem_s`	Signed remainder of two 32-bit integers
`i32.and`	Bitwise AND of two 32-bit integers
`i32.or`	Bitwise OR of two 32-bit integers
`i32.xor`	Bitwise XOR of two 32-bit integers
`i32.shl`	Left shift of a 32-bit integer
`i32.shr_s`	Arithmetic right shift of a 32-bit integer
`i32.shr_u`	Logical right shift of a 32-bit integer
`i32.eq`	Compare two 32-bit integers for equality
`i32.lt_s`	Signed less than comparison of two 32-bit integers
`f32.sqrt`	Computes the square root of a 32-bit floating-point number.
`drop`	Remove the top value from the stack

Table 2.2: Commonly Used 32-bit WebAssembly Instructions

Instruction	Description
`block`	Begin a block of instructions with a label
`loop`	Begin a loop block with a label
`if`	Begin an if statement block with a label
`else`	Pseudo-instructions to begin the ”else” branch of an if statement
`end`	Pseudo-instructions to end a block, loop, or if statement

Table 2.3: Structured control instructions

2.2.10.3 Module

In WebAssembly, the most fundamental unit is the *module*, and every WebAssembly program must contain a module to be valid. The module can have *global variables*, whereas functions can hold *local variables*. *global variables* can be accessed in the entire module, and *local variables* can only be accessed within the function’s scope.

The *module* is written as an S-Expression [24] that wraps around all module instructions. The S-Expression stands for *Symbolic Expression* and is a concept used in *Lisp* [25]. The S-Expression allows the code to be structured in a tree-like structure.

2.2.10.4 Writing style

S-expressions are used for more than just the module declaration but for functions, data section entries, etc. Instructions can optionally use the S-Expression syntax, also called its *folded* form[26]. In that case, the S-Expression is *syntactic sugar* [27]. When instructions are not written as an S-Expression, it can be referred to as the *linear* writing style.

An example of both writing styles is shown in figure 2.10; the examples are semantically equivalent and yield the same result. Note how the nested instructions of a folded instruction are executed before executing the parent instruction so that the execution order is the same as the linear writing style.

```
global.get $var_a
global.get $var_b
i32.add
global.set $var_c
```

(a) Linear writing style

```
(global.set $var_c
    (i32.add
        (global.get $var_a)
        (global.get $var_b)
    )
)
```

(b) Folded writing style

Figure 2.10: Examples of writing style in WAT

The nested structures make the code syntactically look more like a high-level language. Therefore, it may help programmers/students who do not have experience with other assembly languages better understand this writing style.

2.2.11 Application Binary Interfaces (ABI)

In the case of WebAssembly, the term Application Binary Interface (ABI) describes how WebAssembly modules interact with the host system. The WebAssebly module can, via the ABI, access underlying host system capabilities through a collection of functions that resemble a Portable Operating System Interface (POSIX)[28].

Some of the most popular ABIs include *Emscripten* and The WebAssembly System Interface (WASI)[28][17][29]. *Emscripten* aims to deliver POSIX-like functionality for programs it produces, while WASI aims to create a standardized ABI to enable portable, modular, and runtime-independent executables.

2.2.12 WebAssembly Runtime's

In order to facilitate automated testing, the incorporation of a runtime was needed. This section provides a brief overview of the reasons behind our decision to select the *Wasm-Time* runtime. This was done by comparing popular standalone runtimes.

The standalone runtime should be used for running tests of the code compiled by *Hygge-Wasm*. Thus, the following criteria are of vital importance:

- **Reliability:** The runtime should be reliable and, therefore, have a few bugs that can influence testing.
- **Easy integration:** The runtime should be easily incorporated into *.NET*, since the project is written in *F#*.
- **System interfaces:** The runtime should have an implementation of WASI.
- **Efficiency:** efficiency was considered since this could influence the speed at which the test could run.

A lot of WebAssembly runtimes have emerged since the introduction of WebAssembly. In the research for this master thesis, two stood out, namely *Wasmtime* and *Wasmer*.

When it comes to reliability, it seems like both *Wasmtime* and *Wasmer* are fairly mature, with only a few bugs[30]. Therefore, the choice came down to how easy it was to embed into the existing *.NET* application. Both runtimes are available as *Nuget* packages. Wasmer's package has poor documentation[31] and only *6.3K*[32] downloads compared with *198.6K* for the corresponding *Wasmtime* package[33]. Therefore, *Wasmtime* was selected as the main runtime for testing.

2.3 Summary

This chapter provides an overview of the existing Hygge compiler for *RISC-V* and explains its various phases. It establishes that the front-end of this compiler will be reused, and the code generation phase will be redesigned to produce the *target language* WebAssembly.

Further, the chapter introduces the *source language*, Hygge, its type system, and key features. It highlights the relationship between a Hygge program and the typed AST. The chapter also provides several examples of Hygge programs to give readers an idea of the syntax and capabilities of Hygge.

Furthermore, WebAssembly and its key concepts are introduced, and it explains the stack-based execution model. The chapter also explores the textual format of WebAssembly, known as WAT. It covers the module concept, instructions, data types, and writing styles of WAT. Lastly, the chapter discusses the selection of a WebAssembly runtime for testing. After conducting a comparative analysis, the chapter selects *Wasmtime* as the WebAssembly runtime.

3 Design

This chapter clarifies the design choices undertaken throughout the project, accompanied by relevant technical details. The chapter outlines the project into three distinct sections:

3.1 Learning and Development tool

3.2 WAT Generation Framework (WGF)

3.3 HyggeWasm Compiler back-end

3.1 Learning and Development tool

The Learning and Development tool is designed to make it easy to load, run, and debug WebAssembly programs produced by the *HyggeWasm* compiler. A formal requirement specification of the tool can be found in appendix A.2 and a manual for using the tools can be found in appendix D.

3.1.1 Designing effective debugging experience

The research focused on offering students and developers the best possible debugging experience with little to no knowledge of WebAssembly and its textual format WAT.

Debugging tools for WebAssembly have been investigated. It has been observed that various methodologies exist for debugging WebAssembly programs. The chosen approach was to use the built-in debuggers and development tools of modern browsers investigated[34]. This option was chosen for the reasons listed below:

- The best feature support, including *WasmGC*[35].
- Familiar debugging experience and User Interfaces (UI).
- The possibility of supporting both *HyggeSI* and WASI trough the `JavaSript` Application Programming Interface (API).

The web browsers have a very strong feature set and support compared to traditional debugging environments. In particular, Google Chrome and Mozilla Firefox are heavily invested in WebAssembly tooling. As of the time of writing, only Google Chrome version 119 and later and Mozilla Firefox version 120 and later will be able to run programs compiled with the *Heap* memory allocation strategy since *WasmGC* is only supported in these browsers. Developing has primarily been done using Google Chrome; therefore, launching the application in this browser is highly recommended.

Other approaches that were investigated include the use of traditional debuggers such as *GDB* and *LLDB* alongside a supported runtime such as *WasmTime* or the use of tooling specifically developed for debugging WebAssembly. *GDB* does not yet support Apple Silicon[36], since the primary computer used to develop this project is based on Apple Silicon, was *GDB* not further investigated. *LLDB* was tested and has all the features needed to debug WAT programs. *LLDB* has a Graphical User Interface (GUI) mode, which is actually a Text-based User Interfaces (TUI) shown in the terminal [37]. The experience of using this mode can be very frustrating due to the lack of a real GUI. Therefore, it was also considered to build a simple GUI on top of *LLDB* with the *LLDB* Python Scripting API[38]. This idea was abandoned due to it being a potentially substantial undertaking.

3.1.2 User interface

During the project's initial phase, wireframes of the UI were created. The wireframes went through multiple iterations. These iterations are in the appendix B. Please note that wireframe B.1 was made before the technology was chosen, and therefore, it was not meant to be a web application.

When it was clear that the learning and development tool would be a web application, the wireframe design B.2 was chosen. A screen grab of the web application running can be seen in appendix C.

3.1.3 HyggeWasm runtime

HyggeSI is the interface to provide I/O for the Wasm module, while memory allocation functionality is used through it. The two components combined are referred to as the HyggeWasm runtime. The Learning and Development tool implements the HyggeWasm runtime.

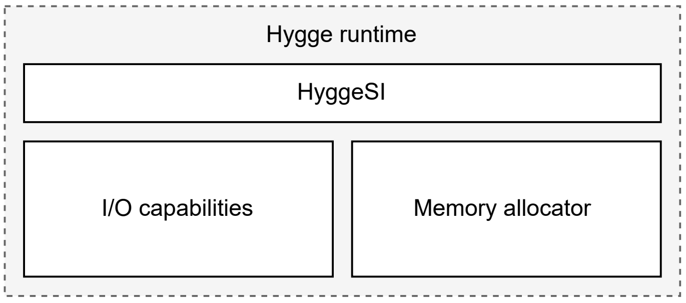


Figure 3.1: HyggeWasm runtime

The HyggeWasm runtime was created as a simple alternative to WASI that integrates well with the Learning and Development tool and provides external memory allocation. Furthermore, the HyggeWasm runtime enables the compiled modules to delegate the responsibility of memory allocation to the host system, resulting in smaller and more concise WebAssembly modules.

This approach was inspired by *AssemblyScript* and *Emscripten* that also can offload some memory management to their respective runtimes[39]. *Emscripten* compiles the needed runtime functionality into a *JavaScript* bundle file associated with one WebAssembly module. This is further described in 2.2.2. A key difference from the approach of *Emscripten* is that the compiler does not emit any *Javascript* alongside the WebAssembly module. Instead, the Learning and Development tool provides this functionality.

The Hygge programs that require external functionality are programs compiled with the *external* memory mode and/or The I/O supplied by the HyggeSI.

3.2 WAT Generation Framework (WGF)

This section describes the design of the WAT Generation Framework, from this point called *WGF*. *WGF* was designed as a separate segment from the compiler components.

WGF defines a representation of the WAT module, essentially serving as an IR. Furthermore, it supplies a API for creating and manipulating the IR called the Module API. Additionally, it implements an algorithm for producing the WAT format from this representation.

As previously described, one of the desired features of the compiler is embedding comments into the output WAT file. One of the responsibilities of *WGF* is to perform this task.

3.2.1 Design research

During the project's initial phase, research was conducted into how other compilers represented and produced WebAssembly. The research found that most WebAssembly compilers produce the binary format, not the WAT format. Since comments can not be preserved in the binary format, it was decided to keep the WAT format as the target language; this will be further explored in section 3.3.2. The research, therefore, revolved around solutions that could represent WAT.

One source of inspiration found in the research was the *Binaryen* toolchain. It has a compiler library for WebAssembly called *Binaryen.js*[40]. *Binaryen.js* provides an API for creating the *Binaryen* IR, optimizing the module and producing both binary modules and WAT modules[41][40]. This would have been an excellent choice for this project if the original compiler front-end was written in *JavaScript*, even though it lacked the comments feature. *.NET* bindings for *Binaryen* exist, but this project has not received an update in more than 6 years[42], and it did therefore not seem promising. Furthermore, none of the existing solutions allowed for comments in the code, thus not satisfying the project's requirements.

Therefore, a decision was made to develop a new representation called WGF IR. While this entailed a considerable increase in workload, it also afforded significantly greater flexibility and complete control over the internal mechanisms of the IR. The Module API of *WGF* was loosely inspired by *Binaryen.js*.

3.2.2 Intermediate representation and API

The *WGF* IR can easily be manipulated, combined, and reconstructed within the code generation phase. It hides unnecessary complexity and simplifies the development of the code generation. The WGF IR closely resembles WAT code written in the folded form; this way, the IR captures the nesting of instructions. The WGF IR can be modified through the Module API. The Module API supply functionality adds imports, exports, local and global variables, etc. Another key feature of the WGF IR is its ability to merge two *WGF* IR modules. This is used in code generation to put together fragments of compiled code. The WGF IR offers different ways of merging modules by overloading the operators '+' and '++'. The '+' operator can be used to combine modules `module_0 + module_1` resulting in the instructions of *module1* to be appended to *module0*. Moreover, can a list of instructions be prepended to a module with the '++' operator like so, `instructions ++ module`. Furthermore, instructions can be appended to a module using the function *AddCode*. The full API design can be found in appendix J.

3.2.2.1 Example of using the intermediate representation

To illustrate the use of the WGF IR, consider the code example found in 2.10b. This WAT program can be represented in the *WGF* IR as shown in code example 3.2. Notice that the code snippet 2.10b example omits the module sections and only shows the function body and, therefore, does not show the declaration of the global variables. In this example, the `I32Add` node is the parent of the two `GlobalGet` nodes. This indicates that WebAssembly instruction *i32.add* will consume the values pushed to the operand stack by the *global.get* instructions, and this association between instructions can be used in the optimization phase and allow the WAT printing algorithm to both print in the linear and the folded writing style when generating the WAT module.

```
Module()
  // Declare global variables in the module
  .AddGlobal(("var_c", (I32, Mutable), (I32Const 0, "init value of var_c")))
  .AddGlobal(("var_a", (I32, Immutable), (I32Const 2, "init value of var_a")))
  .AddGlobal(("var_b", (I32, Immutable), (I32Const 2, "init value of var_b")))
  // Add the instructions that act on the global variables
  .AddCode(
        [ (GlobalSet( // global.set of variable 'var_c'
              Named("var_c"),
              [ (I32Add( // i32.add of var_a and get var_b
                    [ (GlobalGet(Named("var_a")), "get var_a") // read var_a
                      (GlobalGet(Named("var_b")), "get var_b") ] // read var_b
                  ),
                  "add var_a and var_b") ]
           ),
           "set var_c") ]
    )
```

Figure 3.2: Example of constructing WAT modules with WGF

3.2.3 Producing the textual format

A core responsibility of *WGF* is to translate the IR of the WebAssembly module to a correctly formatted WAT module. Moreover, the algorithm formats the code in linear and folded writing styles. this includes proper code indentation based on the code's nesting level and/or placement within the module.

The motivation for supporting the linear and the folded writing style was primarily to support a broader range of assemblers. Please see Section 7.2 for more details.

The secondary motivation was to accommodate diverse user preferences and enhance readability, allowing individuals to choose the format that best suits their preferences. To enable the generation of both writing styles, the IR has to capture the nesting of the folded form. An earlier version of the *WGF* represented instructions as a simple sequence, resembling the linear style. This was much simpler, but it wasn't easy to translate to the folded form. See appendix N.1 for more details.

3.3 HyggeWasm Compiler back-end

This section will explore key design decisions related to the code generation itself.

3.3.1 Phases of code generation

The code generation has been split into four phases, as illustrated in fig. 3.3. The **initial code generation** stage produces the IR defined by the WGF. Section 4.3 describes how the code generation phase is designed and implemented.

The **local variables' promotion** phase then refines the IR. This approach simplifies the code generation process since all let-binders can be initially treated as local variables. All top-level local variables must be promoted to global for the program to work correctly. Top-level local variables are defined by being in the global scope of the original Hygge program and, subsequently, located within the *_start* function in the WAT module. This is necessary because the promoted variables can be used across any function inside the WAT module.

When this step is completed, the module is valid and can be translated to WAT and run correctly. Optionally, the module can be **optimized** in a third phase. This phase is described in Chapter 6.

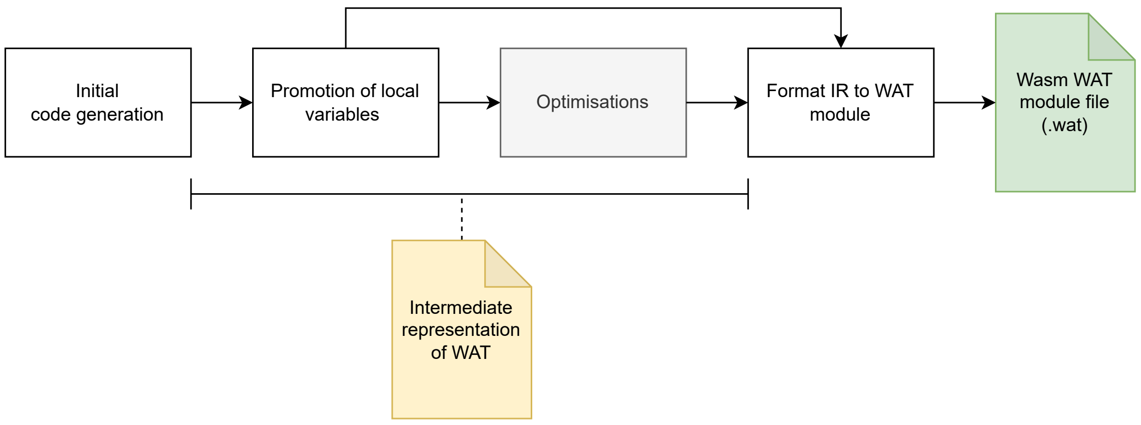

Initial code generation
Promotion of local variables
Optimisations
Format IR to WAT module
Wasm WAT module file (.wat)
Intermediate representation of WAT

Figure 3.3: Phases of code generation

These steps yield an IR of the WebAssembly module, which is subsequently translated into the actual WebAssembly Text (WAT) module. Further details on this process are provided in Section 4.2. Refer to Section 3.3.2.1 for insights into how the WAT module transforms into an executable program.

3.3.2 Compilation target

The compiler has been designed to produce *WASM32* as the target architecture and does so in the textual format of WebAssembly. *WASM32* is a flavor of WebAssembly where integers, floating points, and pointers (*int*) are all 32-bit.

Most compilers targeting WebAssembly produce the binary format directly, to produce an executable file. This approach affords greater control, eliminating the necessity for intermediate software to obtain the executable file. This is a further advantage if the module includes additional information such as Debugging With Attributed Record Formats (DWARF) symbols for debugging, where modifications to the code section potentially lead to the wrong offsets in the source code.

HyggeWasm is for didactic reasons designed to produce the textual format of WebAssembly. This allows students to examine the compiled code directly in a human-readable format, facilitating a more in-depth understanding of the intricacies of the generated program. Additionally, it allows the compiler to generate accompanying comments for the individual instructions within the code, enhancing the comprehensibility of the output. Furthermore, this approach simplifies the mapping of the constructs in the AST directly to a coherent sequence of instructions, enabling students to establish a clear correspondence between the high-level constructs and the low-level code. Furthermore, this approach ensures that the student/developer can inspect the WAT module in the human-readable form before it is potentially altered to obtain the executable binary format.

3.3.2.1 Obtain the binary executable

Figure 3.4 shows the full procedure of going from a Hygge program (.hyg file) to an executable binary module (.wasm file). The procedure involves two steps:

1. **Compile the Hygge source program:**

   The hygge source code is compiled using the HyggeWasm and translated into WAT.

2. **Use WebAssembly Assembler:**

   The compiled code is then processed by the WebAssembly Assembler, translating the textual representation of WebAssembly into the binary format.

The Hygge program can be compiled using the Command-Line Interface (CLI) of HyggeWasm. appendix E provides a complete guide for doing this.

Then, a binary tool like *Wat2Wasm*[43] can be used to obtain a binary executable module. The tool *Wat2Wasm* does not change the module but tries to do a one-to-one transformation from the textual to the binary format. Consequently, it preserves the integrity of the original code, allowing for consistent examination and analysis, which is particularly important for educational and debugging purposes. Therefore, using *Wat2Wasm* as the primary assembler is highly recommended. The tool *wasm-as* has the ability to optimize the WebAssembly module. This process might encompass multiple optimization steps and can change the module and instructions significantly. Also, variable names and labels may be renamed to internal names utilized by the tool. Following the original code can become complicated due to this.

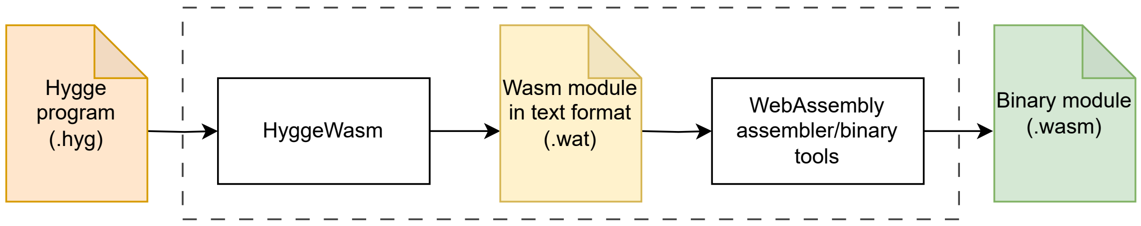

Hygge program (.hyg)
HyggeWasm
Wasm module in text format (.wat)
WebAssembly assembler/binary tools
Binary module (.wasm)

Figure 3.4: Process of obtaining an executable module

Alternatively, the CLI can be used with a Visual Studio Code launch configuration. This makes running files within the Visual Studio Code Integrated Development Environment (IDE) easy and automates the entire process. The configuration files for achieving this can be found in appendix I.

3.3.3 Compiler modes

The HyggeWasm compiler has been designed with several modes of operation. This section will overview these and explain the purpose of the different modes. Subsequent sections will delve into the intricacies of each mode, exploring their limitations and advantages, while also elucidating the fundamental concepts that underlie their design.

- **Memory Modes**: Control how memory is allocated and what storage mechanism is used.
  - **Internal**: Memory is allocated from within the WebAssembly module, the *linear memory model* is used.
  - **External**: Memory is allocated by the embedder, and the *linear memory model* is used.
  - **Heap**: Memory is allocated by the VM and garbage collection is enabled, the *Heap memory model* is used.
- **Writing Style Modes**: Writing style of the produced WAT module.
  - **Linear**: WAT is produced in the flat linear writing style.
  - **Folded**: WAT is produced in the nested folded writing style using *S-Expressions*.

- **System Interface Modes**: Defines what interface is used for I/O.
  - **HyggeSI** (Hygge System Interface): Interface specially designed for Hygge-Wasm.
  - **WASI**: Standard interface.

3.3.3.1 Execution in WASI environment

The binaries produced can be executed in a WASI execution environment when the correct compiler flags are set. An overview is shown in table 3.5.

Mode	Internal	External	Heap
Strategy	Linear memory		Heap
HyggeSI	No	No	No
WASI	Yes	No	Yes

Figure 3.5: Run modules in WASI environment

Be aware that the *heap* mode can only be run in an implementation of the WebAssembly VM that supports the *WasmGC*[44].

3.3.4 Memory management and operation modes

This section is concerned with when to use the different memory modes and outlines how they operate.

The ***external*** mode is the mode that best reflects how memory was handled in the hyggeC compiler that produced *RISC-V*. In that version, all memory allocation was handled by interacting with the operating system (OS) via system calls. Similarly, the *external* mode does not rely directly on the operating system (OS) to provide a memory block but calls via the Hygge system interface (HyggeSI) that provides a runtime that can handle memory allocation. This approach offloads some of the logic to the runtime and, therefore, produces smaller, more concise WAT files that are easier to read. A disadvantage is that the compiled binary can only be executed in an environment where the *HyggeSI* is implemented.

The ***internal*** mode is self-contained and does not rely on *HyggeSI* for memory allocation. This is a step towards a universal executable module. A disadvantage is the larger WAT modules because the logic for handling memory is embedded in the module. For a deeper understanding of this size difference, refer to Section 7.1.

The ***heap*** mode transfers control of the dynamically allocated structures to the VM using features in the *WasmGC* proposal. The heap is completely managed by the VM, meaning no allocation and memory management logic is needed inside the module. Additional type declarations are needed for *heap types*; these are added to the type section and use specialized instructions to create and manipulate structures on the heap. This also allows for universal executable modules, even though the support for the *WasmGC* proposal is limited at the time of writing. In this mode, *linear memory* is exclusively used for static data.

3.3.4.1 Memory layout and bump allocation

This section delves into the design of memory layout and management in the *internal* and *external* modes using *linear memory*. *Linear memory* is, as before mentioned, essentially an untyped binary array; thus, the indices of the array can be seen as a memory address represented as an integer value. Memory management of the heap mode will be described in Chapter 5.

Bump allocation Bump allocation is a simple yet limited approach to memory allocation[45]. In bump allocation, we possess a chunk of memory and maintain a pointer within that memory. When allocating a structure, an assessment ensures that sufficient capacity is left in the current page(s). In case there is not enough capacity, the memory is expanded. We then update the pointer by the size of the allocated structure, completing the procedure. The expansion is not typically part of bump allocation but was added to allow programs where the memory needed can not be determined before execution. Bump allocation is utilized in HyggeWasm for static and dynamic allocation. Note that no memory is ever deallocated; only the disposal of the module instance will free the memory.

Static data is allocated during compile time, and Heap memory is allocated during runtime. This heap memory is not the same as used in *heap* mode. Static data begins at index zero and ends at an index that will be denoted `heap_base`. `heap_base`, marking the beginning of the dynamically allocated memory space. Subsequently, `heap_end` indicates the end of set memory space. At the start of program execution, `heap_base` and `heap_end` are equal.

In *external* mode, the `heap_base` is an exported immutable global variable, always holding the point where the static data ends. This value is read by the runtime and used as the initial point of dynamic allocation. Every time runtime allocation is done, the `heap_end` will be moved by the size of the allocated space. `heap_end` is a value maintained by the external runtime.

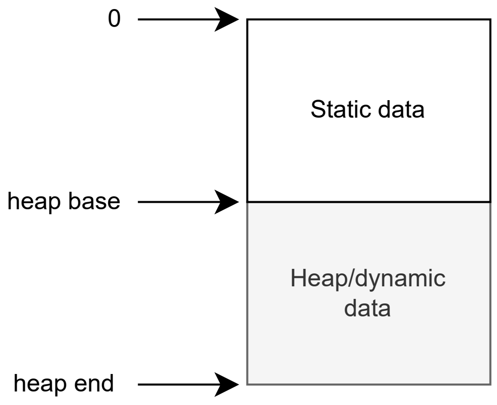


Figure 3.6: Memory layout for External mode

In *internal* mode, `heap_base` is an exported mutable global variable. The Learning and Development tool only uses the export for module validation. In this mode, `heap_base` is modified from within the module to reflect the next memory block's start address. In *internal* mode, there is no distinction between static data and heap data.

3.3.4.2 Statically growing memory

A module compiled by HyggeWasm will always have at least one memory page available. In cases where the amount of static data is greater than one page, the memory is statically grown at compile-time by increasing the number of pages available at the module's initialization. The static data will overlap multiple pages, as shown in 3.7. This could be a scenario where the module contains a large string(s), as seen in the test *memory-static.hyg*.

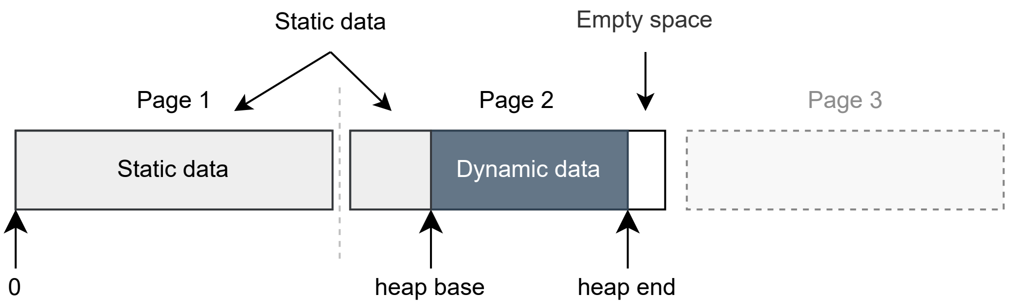


Figure 3.7: Static data trigger expansion

3.3.4.3 Dynamically growing memory

This section will address dynamically growing memory at runtime within *Linear memory*.

External The *external* mode moves the responsibility of keeping track of the memory space to the host environment. In this case, mechanisms for growing memory are needed to ensure that Hygge programs are not restricted at runtime. WebAssembly has an upper bound of *4GiB* as described in section 2.2.4.3. An example of a program that needs to expand memory at runtime can be seen in figure 3.9. The dynamic data will span multiple pages in such a case, as illustrated in Figure 3.8.

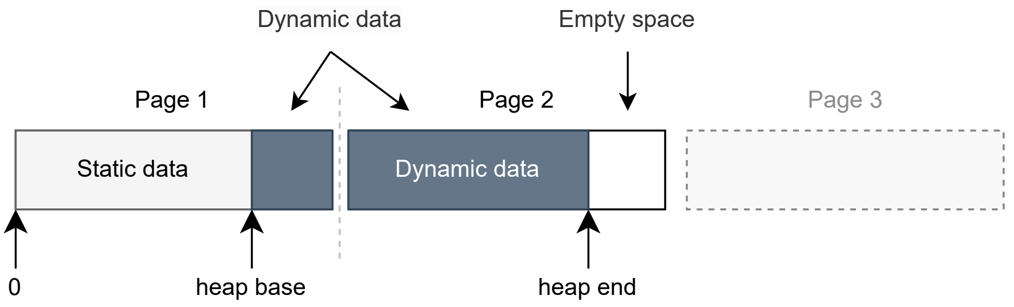


Figure 3.8: Dynamic data trigger expansion

This example shown in Figure 3.9 has no real application and is only for demonstration purposes. The program creates a *struct* of *8 bytes*, verifies the data within the struct, and repeats this *10,000* times. This is enough to fill the initial page and trigger a memory expansion.

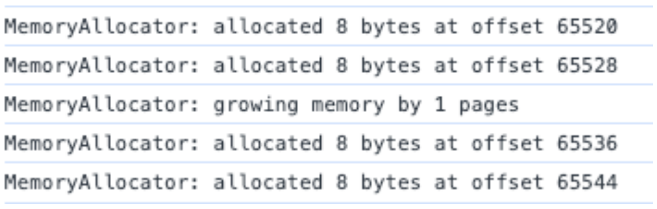
This can be easily observed by compiling the program in *External* mode and running the program in the developer tools with verbose logging enabled. This will produce a console output, a section of which can be seen in figure 3.10.

Since the page size is *65,536 bytes*, it is unsurprising that it is at this threshold the memory is grown, since the next allocation starts at the very end of the first page and goes *8 bytes* into the next page. In more general terms, the memory is expanded when an amount of memory is requested that can not be contained within the current space available.

Internal When compiling for memory mode *internal*, the bump allocation is included as WebAssembly in the module. Appendix M shows an example of this included Wasm code. The WebAssembly code checks if a memory expansion is needed whenever memory

```
let mutable x: int = 0;
let stop: int = 10000;

while (x < stop) do {
    let s1: struct {f: int; i: int} = struct {f = 42 + x; i = 42};
    assert(s1.i = 42);
    assert(s1.f = 42 + x);
    print(x);
    x++
};

print("done")
```

Figure 3.9: Allocating memory for 10000 structs



```
MemoryAllocator: allocated 8 bytes at offset 65520
MemoryAllocator: allocated 8 bytes at offset 65528
MemoryAllocator: growing memory by 1 pages
MemoryAllocator: allocated 8 bytes at offset 65536
MemoryAllocator: allocated 8 bytes at offset 65544
```

Figure 3.10: Log from developer tool

is requested. If needed, is the memory extended just before a new memory address is returned. The memory-specific instructions `memory.size` and `memory.grow` are used. `memory.size` will leave the number of pages the module memory instance currently has. The instruction `memory.grow` will expand the memory with *n* number of pages. The current `heap_base` value is left on the stack and then updated to accommodate the newly allocated memory space.

3.3.5 Writing Style modes

The HyggeWasm compiler can produce two styles of writing WAT, the nested *folded*- and the flat *linear*-writing style. The two writing styles correspond to examples shown in code example 2.10. Both writing styles have advantages and can, in principle, be combined. This also means WebAssembly does not restrict a WAT module to contain either writing style exclusively. HyggeWasm, on the other hand, aims to generate either one or the other exclusively.

Writing Style influences what assemblers can be used. Despite the assertion within the WebAssembly specification that the folded writing style merely constitutes *syntactic sugar*, the practical reality is that numerous tools and infrastructure associated with WebAssembly exhibit a lack of maturity. Consequently, the adopted writing style can substantially impact the parsing capabilities of assemblers when dealing with WebAssembly Text (WAT) modules. An illustrative instance of this is the assembler *wasm-as*, an integral component of Binaryen's toolchain. Notably, in its present iteration, *wasm-as* singularly operates with the folded *S-Expression* format. How this affects HyggeWasm is clarified in section 7.2.

The ***Folded*** writing style mode nests instructions under a parent's instruction, making it easier to grasp the relationship between instructions. It can, therefore, be a good option when exploring code.

The ***Linear*** writing style mode is most useful when debugging. This is because Google

Chrome decompiles the binary module back to the *linear* Style of WAT when using the debugging tool. The decompilation from the binary module to WAT omits the comments since comments can not be represented in the binary format.

Therefore, a design goal has been to make the linear style look close to identical to what Google Chrome displays to make it easier to follow the original code with comments when debugging with the development tool. In addition, elements within the module have been automatically assigned comments indicating the element's index within its respective section of the module. For instance, the first declared global variable that has an index of zero will have `(;0;)` placed after the variable name. This feature can be particularly useful when referencing a global variable, which is not referenced by its name but rather by its index. An example illustrating this concept is presented in code snippet 3.11.

```
(module
  (type $i32_i32_=>_i32 (;0;) (func (param i32) (param i32) (result i32)))
  (import "env" "malloc" (;0;) (func $malloc (param i32) (result i32)))
  (memory (;0;) (export "memory") 1)
  (global $exit_code (;0;) (mut i32) (i32.const 0))
  (global $fun_f*ptr (;1;) (mut i32) (i32.const 0))
  (global $fun_g*ptr (;2;) (mut i32) (i32.const 4))
  (global $fun_g/anonymous*ptr (;3;) (mut i32) (i32.const 8))
  ;; rest of the module ...
```

Figure 3.11: Indexed element sections inside WAT module

3.3.6 Input/Output and System interface modes

HyggeWasm allows for either using the Hygge system interface (HyggeSI) or WASI for I/O.

HyggeSI has implemented all functionality described in appendix G. However, the WASI implementation is limited and only supports reading an integer as input and printing strings to standard output. Consequently, it serves more as a proof of concept than a comprehensive implementation. It is therefore recommended to use the *HyggeSI*.

3.3.6.1 Hygge System Interface (HyggeSI)

This section will not cover all functionalities of HyggeSI. Instead, it will focus on the design of string printing as an example.

A namespace and a function name describe all functions in the interface. In the case of string printing, the namespace "env" and the function name "writeS". The signature of the function is as follows:

$$(address : int, length : int, newline : int) \rightarrow ()$$

WriteS is designed to access the module's memory. As mentioned earlier, the WebAssembly module is encapsulated. To allow the host system access to the linear memory of the module, the module must explicitly export the memory.

The first argument of *WriteS* is the address, an offset in linear memory where the string data starts. The second argument is the length of the string in bytes. The last argument signals if the string should be printed with a line break, the value 1 or higher will result in a line break.

3.3.7 Entry point

As stated earlier, a Hygge is an evaluable expression which means it does not need a main function. This is common for scripting languages such as *Python*, *Ruby*, and *JavaScript*. Thus, an entry point of execution is not explicitly defined. The program is typically executed sequentially from the top down in these languages.

The Hygge programming language operates similarly to typical scripting languages in this regard and has no main function. In contrast, WebAssembly is specifically designed to encapsulate all executable code within functions. To address this divergence between the two approaches, Hygge can enforce that every program includes a main function.

This method is less desirable since Hygge is a well-defined language, and this would add a seemingly arbitrary new rule. Additionally, there is an existing Hygge compiler for the *RISC-V* ISA, and it is preferable that both compilers can handle the same Hygge programs. Moreover, numerous Hygge test programs have already been written for the aforementioned ISA that preferably should be reused in the test suite.

Therefore, a more pragmatic approach was chosen. The proposed solution encapsulates the global scope of the Hygge program in an implicit function inside the WebAssembly module. Optionally, Hygge could allow for an explicitly defined function with a unique name, such as "main" or "start," to let the Hygge programmer capture this behavior. In this project's scope, this will be perceived as future work.

To conform with the WASI conventions the main function of a WASI-compatible program must be exported with the name `_start`. Therefore, all `HyggeWasm` generated modules will have such an export. This function can then be invoked from the host environment, and a result can be returned to the host from the WebAssembly module function.

Another approach used by, for instance, *AssemblyScript*, is to define a start function that will execute when a WebAssembly module is instantiated. This is done in the start section of the WebAssembly module like this: `(start $label)`. One problem with this is that the host environment does not retrieve the return value of the main function. It is necessary to separate the instantiation and execution processes to enable the Learning and Development tool to load and instantiate a module without immediate execution. This allows users to set breakpoints and observe the module instance before executing any code.

3.3.8 Program termination

The program can either execute as intended, reaching a point where the `_start` function returns, thereby ending the module's execution, or it can enter a faulty state and, consequently, need to be terminated. This may happen when an invalid input is given, for instance, trying to create an Array of size -1 or when the *Assert* expression is evaluated with a faulty boolean statement. The `unreachable` instruction is used to indicate such errors.

The `unreachable` instruction signals that the current program counter (PC) is not valid during normal program execution. Thus, it can be used as an assertion, signaling that a bug or unexpected behavior has been reached. The *unreachable* instruction will cause an unconditional trap when executed and will return control to the host system. The `unreachable` instruction does not return any additional information to the host system. Thus, another way of signaling an exit code had to be established. Therefore, *HyggeWasm* is designed to set a global value called `exit_code` when a runtime error occurs, such as an assertion failing. The `exit_code` value is exported so the host can obtain the value after termination. The `exit_code` determines whether each test yielded the expected result while running the automated test suite.

Early versions of the *HyggeWasm* back-end explored whether the return value of the `_start` function could be treated as an exit code of the program. This was scrapped since this meant that the program would continue execution even after a faulty assertion was made. Furthermore, a global variable holding the exit code is needed to assert statements embedded in other structures like other functions or control structures. Otherwise, all asserts would have to be used at a top level, an undesirable restriction not part of the Hygge language specification.

3.3.9 Label naming

As part of the original *RISC-V* compiler, several utility functionalities were provided. One of these ensured the unique naming of labels by appending a number to a variable name. It was found that this had to be changed to ensure unique names for HyggeWasm. The problem arose when the hygge programmer chose a name equal to the internal name used by the compiler.

For example, if two variables were named *i* in the same function, the first variable would get its internal name `var_i` and the second `var_i_0`. If a new variable is introduced in the same function with the name `i_0`, the compiler will recognize it as a unique name and, therefore, not append any additional numbers to it. This would give it the internal name `var_i_0` and, therefore, create a name collision with the second variable. To fix this problem, the naming changed to use the scheme `var$num`. This ensures unique names since a valid variable name in Hygge can not contain the `$`-character.

3.3.10 Function types

Types reside in their section in each WebAssembly module. Functions need to declare their signature as a type. In an early version of *HyggeWasm*, the types were declared in line with the function declaration. This was later changed to deliver more readable code and more flexibility. When a type is declared by itself, it can be reused and referred to when, for instance, doing an indirect call to a function.

In early versions of *HyggeWasm*, indirect calls did not use the type declarations but explicitly wrote out the entire function signature every time it was called. This was simple from a code generation standpoint since the function signature could be found in the typed AST.

Types had three forms throughout the development. The explicit one is discussed above. The first version had a very naive approach where every function had a type labeled with the function name and a suffix "_type". This was a valid solution but did have a disadvantage that there was no type of reuse, and type names had to be looked up through the function reference table every time instructions for an `Application` (invocation) had to be done.

A new method was implemented to enable reuse and easy usage of the function types, inspired by *AssemblyScript*. Instead of having a type name that one-to-one mapped to a function, the naming scheme was changed to reflect the type it represented. This way, multiple functions can share a type declaration. Furthermore, the type label can be easily recreated based on the type annotated AST, eliminating the need to look up function type relationships.

3.4 Summary

The chapter begins with an exploration of the design considerations behind the Learning and Development tool, focusing on debugging methods and the decision to utilize a web application to leverage modern browser-based development tools.

It then introduces the design and functionalities of the HyggeWasm runtime and HyggeSI, highlighting HyggeSI's role as the interface for I/O and memory allocation within the WebAssembly module, forming the core of the HyggeWasm runtime.

The WGF IR is introduced as the foundational structure for code generation, followed by outlining key design decisions for code generation, such as the compilation target and the method for generating executable WebAssembly modules.

The chapter also explains HyggeWasm's operation modes, including memory modes, writing styles, and system interfaces, detailing their inner workings and showcasing configurations for producing WASI executable modules.

Additionally, it discusses general design decisions like internal label naming, control structure labels, and function type identifiers.

Moreover, it describes the creation of the target program's entry point and its design to enable the automated test suite to verify program execution correctness and handle termination in case of incorrect execution.

4 Implementation

This chapter documents the implementation details of key components of the project. The components are:

4.1 Development and learning tool

4.2 WAT Generation Framework (WGF)

4.3 HyggeWasm Compiler back-end

The source code of all components can be found in appendix O.

4.1 Development and learning tool

This section describes the implementation of the web application that enables the loading, execution, debugging, and I/O of modules produced by *HyggeWasm*. The tool is implemented as a web application written in *TypeScript* using the framework *React*[46]. This application relies on the web browser's built-in development tools to supply all debugging functionality.

4.1.1 Loading WebAssembly modules

All the functionality used for loading binary WebAssembly modules can be found in the component *wasm-loader* in the file *src/components/wasm-loader.tsx*. To load files into the browser, the Node Package Manager (npm) package *use-file-picker* is used [47][48]. The npm package *use-file-picker* implements a hook[49] that encapsulates the functionality for loading and reading files via a popup file selector window. The behavior of *use-file-picker* can be customized to fit the use case, by applying the arguments to the hook. In this case, only binary WebAssembly module (.wasm) files can be loaded; only one file can be loaded simultaneously. When the file is loaded the data is represented as a *ArrayBuffer*.

The data found in the *ArrayBuffer* can then be used to compile the WebAssembly module via the *WebAssembly JavaScript API*[50]. The *WebAssembly.compile* method compiles WebAssembly binary code into a *WebAssembly.Module* object. Normally, the *WebAssembly.instantiate* method can be used directly on the binary data, but in this case, we want to validate some attributes of the module first.

The validation ensures that the file loaded will work correctly with the tool. This validation will approve all files produced by *HyggeWasm*. The process includes confirming that a heap base pointer value is exported; in this case, there is no such export, and the user is given an error. It does not stop the user from trying to run the module. Similarly, it is ensured that the *_start* function is exported in the module. The described process is shown in code snippet 4.1.

```
// code loading the file
// ...
// Compile the module into an object
const module: WebAssembly.Module = await WebAssembly.compile(bytes);
// perform checks
const exports = WebAssembly.Module.exports(module);
const heapBase = exports.find((e: exportValue) => e.name == "heap_base_ptr" ?
    true : false);
const haveEntryPoint = exports.find((e: exportValue) => (e.name == "_start" &&
     e.kind == "function") ? true : false);
// ...
// code that manipulates the component's state to show errors in the UI if
    needed.
```

Figure 4.1: Compile and check module for errors

When creating an instance of the WebAssembly module, an object containing all the imports used in the module is supplied. Because the Learning and Development tool should be able to run with both *HyggeSI* and WASI, the two objects are combined into one object that holds all the functions of WASI and HyggeSI. The WASI imports are coming from the npm package *wasmer/wasi*[51]. This package polyfills essential WASI functionality that the browser lacks.

Consider the code snippet 4.2, this shows how the instantation is done in *JavaScript* with the needed imports. The *wasmModule* refers to the compiled module in code snippet 4.1.

```
const combinedImports = {
  ...wasiImports, // WASI imports
  ...getImports(memoryAllocator, isDebug) // HyggeSI "custom" imports
};

const instance: WebAssembly.Instance = await WebAssembly.instantiate(
    wasmModule, combinedImports);
```

Figure 4.2: Creating module instance in JavaScript

4.1.2 Implementation of the HyggeWasm runtime in TypeScript

HyggeSI is implemented in the file *src/services/ImportService.ts*. Code snippet 4.3 shows an extract of the HyggeSI implementation. This implementation conforms to the interface described in appendix G.

```
export function getImports(memoryAllocator: MemoryAllocator, isDebug: boolean
    = false): object {
    return {
        env: {
          writeS(address: number, length: number, nl: number) {
            const mem = new Uint8Array( // read memory into a Uint8Array
              (memoryAllocator.memory as WebAssembly.Memory).buffer
            );
            const data = mem.subarray( // create sub-array of string data
              address,
              address + length
            );
            // Convert array into a string as utf-8.
            const decoder = new TextDecoder("utf-8");
            const text = decoder.decode(data);

            if (nl) { // add new line if requested
              console.log(text + "\n");
            }
            else {
              console.log(text);
            }
          },
          writeInt(i: number, nl: number){
            if (nl) {
              console.log(i + "\n");
            }else {
              console.log(i);
            }
          },
          malloc(size: number) {
            let pointer = memoryAllocator.allocate(size);
            if (isDebug) { console.log("malloc", size, "pointer", pointer); }
            return pointer;
          },
          readInt() {
            var num;
            do {
              var val = prompt("Input an integer");
              if (val == null) {
                continue;
              }
              num = parseInt(val);

              console.log("User provided input:", num);
              return num;
            }
            while (num && isNaN(num));
            return 0;
          }
        }
      },
      // more ... (writeFloat, readFloat)
    }
```

Figure 4.3: HyggeSI TypeScript implementation

Three examples will be described: *writeS*: Print console (output), *readInt* Read an integer (Input) and *malloc*: Allocate new memory block.

4.1.2.1 Output

writeS uses the memory allocator. The Memory allocator will be described in Section 4.1.2.3; for now, it is sufficient to know that it refers to the memory of the module instance for the currently running program. *writeS* utilizes the memory allocator's reference to read the memory into a Uint8Array. The array is then used to create a subarray that only contains the string data that should be printed. The raw data is then converted into a `UTF-8` string using the `TextDecoder`; see section 4.3.6.13 to learn more about strings in *HyggeWasm*. Then, the string can be printed with *console.log*. Based on the *nl* argument, a new line character is appended to the string.

4.1.2.2 Input

readInt asks for input as a string until a string can be passed successfully as an integer value. It asks the user for input using a window prompt[52]. How this looks in the UI can be seen in fig. D.6. When an input is successfully passed, the host system returns control to the running module instance, and the input value is placed on top of the *operand stack*.

4.1.2.3 Allocating new memory block

The memory allocator is in the file *src/services/MemoryAllocator.ts*. *malloc* uses the memory allocator to resolve the pointer to the new memory block with the *allocate* function. The *allocate* function takes one argument, the size of the requested memory block, and returns a pointer to where the new structure can be placed in memory. The implementation of *allocate* can be seen in code snippet 4.4. The implementation follows the bump allocation strategy described in Section 3.3.4.1. There is only one instance of the memory allocator since it encapsulates the memory state.

```
/// Allocate a new block of memory of the given size in bytes.
/// Returns the offset of the allocated block.
allocate(size: number): number {
// if the offset + size is greater than the current size of the memory
// then grow the memory by the required number of pages
if (this.offset + size > (this.currentSize * this._pageSize)) {
    // find required number of pages to needed
    const requiredPages = (this.offset + size) / this._pageSize;
    // round required pages to the next integer
    const roundedPages: number = (Math.ceil(requiredPages) as number);
    // difference between current size and the new required size
    const growBy = roundedPages - this.currentSize;
    // grow by n number of page(s)
    this._memory?.grow(growBy); // grow by n number of page(s)
    this.currentSize += growBy; // update current size

    if (this._isDebug) console.log(`MemoryAllocator: growing memory by ${
    growBy} pages`);
}
const addrees = this.offset; // save current offset
this.offset += size; // increment offset by size
return addrees; // return address
}
```

Figure 4.4: Bump Allocator implemented in TypeScript

4.1.3 WASI limitations

The Learning and Development tool is limited when working with WASI programs. While the tool accepts integer inputs and display string outputs when using WASI, the input must be provided before execution of the module. This is due to the unavailability of callbacks every time the Hygge function *readInt* is called. The input prompt is shown based on the presence of the *fd_read* import. As a result, the tool can only accept one integer input in WASI programs. It is important to note that this limitation is not imposed by the compiled code but by the tool itself.

Instead of printing to output during runtime, the tool currently prints the entire output at once after execution.

4.1.4 User Interface

The UI of the web application is very simple. It is implemented in *React* with only a few components defined using *JSX*.

The *verbose logging* switch maintains its state by persisting it in local storage through the use of the localStorage API[53]. This implementation ensures that the toggle's state persists when the page is reloaded.

4.2 WAT Generation Framework (WGF)

The implementation of WGF defines the IR of the module and each instruction.

4.2.1 Instructions

The instructions are defined in *WGF/Instructions.fs*, and an extract of the used instructions are shown in code snippet 4.7. Instructions that consume values from the operand stack holds a list of commented WebAssembly instructions. Some instructions are defined twice with a "_" at the end. These are the memory instructions that can take static immediate memory arguments. Instructions are encapsulated inside the *Commented* type alongside a string meant to comment on the instruction. The type can be seen in fig. 4.5.

```
type Commented<'a> = 'a * string
```

Figure 4.5: Commented instructions

```
type Label = string
type Identifier =
    | Named of Label
    | Index of int
    override this.ToString() =
        match this with
        | Named s -> $"${s}" // '$' in front of label
        | Index i -> $"{i}"  // index as string
```

Figure 4.6: Label and Identifier

```
type Wasm =
    | Br of Label
    | BrIf of Label * Wasm Commented list
    | I32Load_ of int option * int option * Wasm Commented list
    | I32Load of Wasm Commented list
    | I32Store_ of int option * int option * Wasm Commented list
    | I32Store of Wasm Commented list
    | I32Const of int32
    | F32Const of float32
    | I32Add of Wasm Commented list
    | I32Sub of Wasm Commented list
    | I32Mul of Wasm Commented list
    | I32Eqz of Wasm Commented list
    | I32Eq of Wasm Commented list
    | I32LtS of Wasm Commented list
    | LocalGet of Identifier
    | LocalSet of Identifier * Wasm Commented list
    | LocalTee of Identifier * Wasm Commented list
    | GlobalGet of Identifier
    | GlobalSet of Identifier * Wasm Commented list
    | Call of Label * Wasm Commented list
    // more instructions ...
```

Figure 4.7: WebAssembly instructions in WGF

The *Label* type names elements within the module. Some instructions refer to other elements using an *Identifier*. An *Identifier* is used when a *Label* or an index is referencing an element. The type definition can be seen in code snippet 4.6.

4.2.2 Module

The module is a Record and holds all the data about each section of a WAT module. Much of the code for manipulating the module is relatively straightforward; therefore, this section will only highlight essential implementation details. Below is a list of the sections in the module and the type used to represent them.

- `types: list<TypeDef>`
- `functions: Map<string, Commented<FunctionInstance>>`
- `memories: Set<Memory>`
- `globals: Set<Global>`
- `exports: Set<Export>`
- `imports: Set<Import>`
- `start: Start`
- `elements: Set<Element>`
- `data: Set<Data>`
- `locals: Set<Local>`
- `funcTableSize: int`
- `tempCode: list<Commented<Instr.Wasm>>`

- `hostinglist: string list`

The module does not represent tables because only the special *func_table* is used. Therefore, all elements are automatically in the *func_table*. To see a complete list of functions the module contains, see appendix J.

To ensure that section elements are not duplicated when combining modules, many of the module sections are represented by a *Set*, thus not allowing duplicate elements. In the sections where the order of the elements is important, such as the temporary code accumulator or the type section, a *list* is used. Functions consist of a *Map* where the key is the name of the function and the value is a `Commented<FunctionInstance>>` that holds all information about the function alongside a comment.

4.2.3 Produce textual format

The module override the *ToString* function and can transform the module representation into the textual format. Consider the code snippet 4.8 that shows the overall structure of the code. *ToString* is implemented by iterating through all the module sections, formatting the elements to the proper WAT syntax in a string, and appending it to the *result* variable. In the end, the *result* will contain the entire WAT module formatted as a string. When generating the module's functions, the function's body is produced by the *generateText* function.

```
// open module tag
let mutable result = "(module\n"

// print all types
for type_ in List.indexed (this.types) do
    result <- result + (printType type_ false)

// prior sections being formatted into the string...
// create functions
let mutable x: int = 0
for funcKey in this.functions.Keys do
    let (f), c = this.functions.[funcKey]
    // using generateText to format body of function
    result <-
        result
        + $"{gIndent 1}(func %s{genrate_name f.name} %s{ic x} %s{
    generate_signature f.signature c} %s{generate_local f.locals}\n%s{
    generateText f.body style}  )\n"

    // increase x
    x <- x + 1

// more sections being formatted into the string...
// close module tag
result <- result + ")"

// return module represented in WAT format as string
result
```

Figure 4.8: Module ToString

The *generateText* function can format every instruction into the proper syntax of the text format in both the folded and the linear style. *generateText* uses an auxiliary function implemented as a recursive accumulator function called *aux*. The *aux* function takes a list of all the instructions for which code should be generated. The second argument is the WAT string accumulator, and lastly, a number that describes the nesting level, which is used to indent the code correctly. This function has some special cases for instructions, including the *block* instruction because it uses pseudo instructions as delimiters and memory instructions with additional intermediate arguments. Some general cases handle the rest. The general cases can be seen in code snippet 4.9, where line 19 shows how the code is rearranged for the linear style.

```
1   let generateText (instrs: Wasm Commented list) (style: WritingStyle) =
2       let rec aux (instrs: Commented<Instr.Wasm> list) (watCode: string) (indent
        : int) =
3           match instrs with
4           | [] -> watCode
5           | head :: tail ->
6               let (instr, c: string) = head // deconstruct instr and comment
7               let space = gIndent indent // compute indent
8               match instr with
9               | F32Eq instrs
10              | F32Store instrs
11              | I32Add instrs when style = Folded -> // general case - folded
12                  let watCode =
13                      watCode
14                      + space
15                      + $"({instrLabel instr}{commentS c}\n{aux instrs emptyS (
        indent + 1)}{gIndent (indent)})\n"
16
17                  aux tail watCode indent
18              // more cases ...
19              | I32Add instrs when style = Linear -> // general case - linear
20                  aux
21                      tail
22                      (watCode
23                       + (aux instrs emptyS indent)
24                       + $"{gIndent indent}{instrLabel instr}{commentS c}\n")
25                      indent
```

Figure 4.9: Format in folded and linear style

4.3 HyggeWasm Compiler back-end

This section offers a detailed perspective on the practical implementation of the compiler back-end. It seeks to elucidate how design decisions and research findings have been applied in practice, providing a comprehensive view of the implemented compiler back-end.

4.3.1 Solution structure of the HyggeWasm Compiler

The *HyggeC* project depends on the *WGF* and *WasmTimeDriver* projects. The *WGF* has already been discussed, and the next section will delve into the implementation details of the *WasmTimeDriver* project.

The *HyggeC* project contains the compiler implementation itself. *WGF* defines the IR of the WebAssembly module and all the WebAssembly instructions used. The *WGF* IR is used in code generation and the proceeding optimization phase. *WasmTimeDriver* runs WebAssembly code with the Hygge runtime.

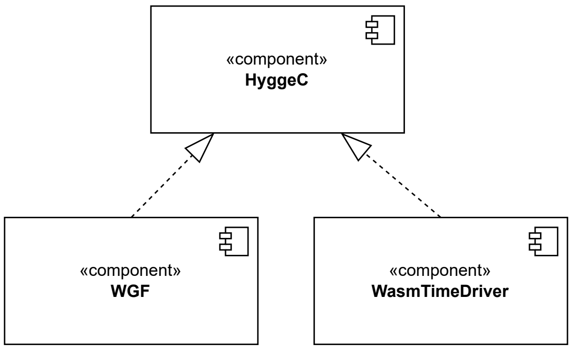


Figure 4.10: Project dependencies

4.3.2 WasmTime driver

WasmTimeDriver is a *C#* project that implements the Hygge runtime and wrap functionality found in the *Wasmtime* Nuget package. This is to make it easy for HyggeWasm to use the functionality for execution and testing of the compiled Wasm programs. The *Wasmtime* Nuget package can operate multiple virtual machines (VM) simultaneously. fig. 4.11 illustrates this in a component diagram.

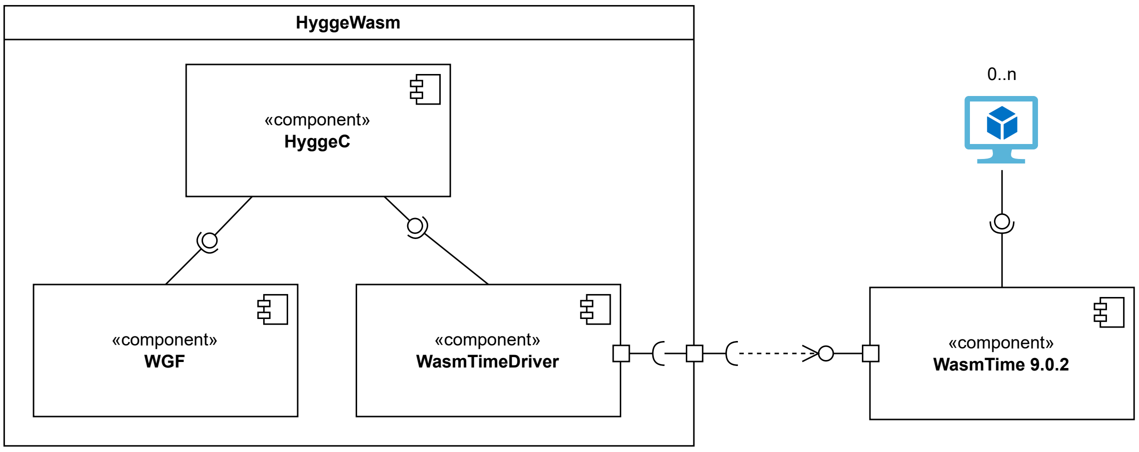


Figure 4.11: WasmTime component diagram

4.3.3 Implementation of the HyggeWasm runtime in C#

The file *WasmTimeDriver/WasmVM.cs* contains all main functionality of HyggeSI as well as supplying functions for running *.wat* files and WAT modules as a string. When running the WebAssembly programs through the *WasmVM* implementation, the HyggeSI is automatically injected. The interface of the WasmVM class can be seen in code snippet 4.12.

```
public interface IWasmVM
{
    object? Run(string wat, string target, string name = "unknown");
    public object? RunFile(string path, string name = "unknown");
    public object? RunFile(string path, string target, string name = "unknown"
    );
    public object?[] RunFileTimes(string path, string target, int n);
    public object?[] RunFileTimes(string path, int n);
    object? RunWatString(string target, string wat, string name = "unknown");
    object? RunWatString(string wat, string name = "unknown");
}
```

Figure 4.12: Hygge code that pushes multiple values to stack

This section highlights key differences as the implementation is similar to the *TypeScript* implementation described in section 4.1.2. Code snippet 4.13 contains a section of the implementation of the HyggeSI. The instance's memory can *malloc* access directly through the *Caller* argument. *readInt* uses *Console.ReadLine* and will, therefore, take input directly in the terminal in contrast to the prompt that the *TypeScript* implementation uses.

```
_linker.Define( // malloc - allocating memory chunk.
    "env",
    "malloc",
    Function.FromCallback(_store, (Caller caller, int size) =>
    {
        IntPtr adreess = _allocator.Malloc(caller.GetMemory("memory"), size);
        return adreess.ToInt32();
    })
);
_linker.Define( // Get an integer as input
   "env",
   "readInt",
   Function.FromCallback(_store, () =>
   {
       try
       {
           string? s = "";
           int res;
           do
           {
               s = Console.ReadLine(); // read line from console
               res = Int32.Parse(s); // parse input as a int
           } while (s is null);
           return res; // return interger value
       }
       catch (Exception e)
       {
           Console.WriteLine("error:" + e);
       }
       return 0;
   })
);
```

Figure 4.13: Hygge runtime implementation in C#

4.3.4 Code generation environment

The code generation algorithm maintains a *Code generation environment*. The *Code generation environment* encompasses various information related to the produced code, including:

- The **name of the function** currently being compiled.
- The **static memory allocator controller** keeping track of memory blocks with bump allocation.
- The **function table controller** keeping track of the current number of elements in the function table.
- The **symbol controller** is a naming utility used to ensure unique names for variables, functions, and IDs within a module.
- The **variable storage** provides information about known variables and where the data associated with a variable is stored. The variable storage is also referred to as *VarStorage*. *VarStorage* keeps track of the following storage types:
    1. **Global** - A *global variable* in the module scope.
    2. **Local** - A *local variable* within a function scope.
    3. **Offset** - A variable residing in memory within a closure environment.

4.3.5 Stack management

As described in section 2.2.5, WebAssembly will do the validation phase, making sure certain constraints are met. This includes that only the expected values are on the operand stack[54], ensuring that the generated code does not keep unused values on the stack. The validation compares the stack values with the result type of *control structures* and *functions*. The result type describes what the *control structure* or *function* evaluates to, thus what value is left on the stack. The result type can be seen in the WAT code as *(result i32)*.

To explore how this affects the code generation of Hygge, consider the code snippet 4.14. This example has the problem that the if-branch will leave two integer values on the stack while the else-branch will leave only one integer. Both branches must have the same return type for the program to be valid. While WebAssembly can have result types that let a control structure leave multiple values on the stack, *HyggeWasm* is designed to strictly allow for only one result value.

```
fun f(arr: array {int}, i: int): array {int} = {
    if (i < arrayLength(arr)) then {   // <-- Result type of (i32)
        arrayElem(arr, i) <- i + 1;    // <-- Push i32 value
        f(arr, i + 1)                  // <-- Function will push address (i32)
    }
    else {
        arr                            // <-- Push i32 value
    }
};
```

Figure 4.14: Hygge code that pushes multiple values to stack

To overcome this problem, the compiler has to recognize what expressions of a sequence that push unused values to the *operand stack*. If a value is not used, that value has to be discarded. Therefore, whenever the compiler generates code for a sequence in the AST it inspects that node's type, meaning what value it evaluates to. The last element of the sequence is expected to be the return value, which is left on the stack. All other elements are evaluated as follows. In the case that the node evaluates to a *unit*, nothing is done, since a *unit* value is not represented in WAT code, and therefore, no value is pushed to the stack. Otherwise, a value will be pushed to the stack, which we know is not the return value if it is not the last in the sequence. A *drop* instruction ensures this value is discarded and the stack is kept clean and consistent. The implementation can be seen in code snippet 4.15.

```
| Seq(nodes) ->
    let lastIndex = (List.length nodes) - 1 // index of return value
    List.fold
        (fun m (i, node) ->
            if (i = lastIndex) then
                // return last node with no modifications
                m + doCodegen env node (Module())
            else
                match node.Type with
                | TUnit -> m + doCodegen env node (Module())
                | _ ->
                    let subTree = (doCodegen env node (Module()))
                    // drop value of 'subTree'
                    (m + subTree.ResetAccCode())
                        .AddCode([ (Drop(subTree.GetAccCode()), "drop") ]))
        m
        (List.indexed nodes)
```

Figure 4.15: Handling sequnces

4.3.5.1 Initilizing static segments of data

Throughout the compiler's back-end development, various scenarios arose where space must be statically allocated during compilation and the data placed in memory before execution. As discussed in section 2.2.6, the WebAssembly module's data section can initialize memory segments during module instantiation by providing a string of data and a start address. From this point on, the term *data string* will be used to refer to a string that contains the data as 8-bit hexadecimal numbers. The *data string* allows the module to initialize large memory blocks statically as hexadecimal digits.

This approach was inspired by *AssemblyScript's* data segment initialization. Code example 4.39 displays its representation in the WAT module after being compiled by Hygge-Wasm.

To create a *data string*, the 32-bit integer is first split into four 8-bit chunks by shifting the values to the right. Then, a mask is used to obtain the 8-bit value for each chunk. Next, each value is converted into a hexadecimal string with the correct format. finally, all the hexadecimal strings are concatenated into one string.

The approach used was important in keeping the module easy to read. By allowing the compiler to initialize large segments in a single data entry instead of creating one for each 32-bit value, the data section of the module became significantly smaller.

The file *util.fs* contains the logic for producing the *data string*. The function *dataString* takes a list of data as input and produces the string.

```fsharp
let intToHex (i: int32) : string = // int to hex value
    let hex = System.Convert.ToString(i, 16)
    let paddedHex = if hex.Length = 1 then "0" + hex else hex
    System.String.Concat("\\", paddedHex)

let intTo32Hex (value: int32) = /// int to hex string
    let mask = 255 // 8 bits set to 1
    List.fold
        (fun acc elem -> acc + intToHex elem)
        ""
        [ value &&& mask  // it is split op into 4 bit chunks
          (value >>> 8) &&& mask
          (value >>> 16) &&& mask
          (value >>> 24) &&& mask ]

let rec dataString l = // combine multiple values in one string
        match l with
        | [] -> ""
        | x::xs -> intTo32Hex x + dataString xs
```

Figure 4.16: How hexadecimal data strings are produced

4.3.6 Language features

This section will describe how the code generation algorithm has been implemented. It will do so by going through each language feature found in appendix A.1 individually.

The features have been divided into the following sections. Please note that some sections will describe multiple features.

1. Literal values
2. Negative numbers
3. Arithmetic and logic operators
4. Variables
5. Input/Output and interaction with host system
6. Assignments
7. Compute-assign operators
8. Pre- and post-increment and decrement operators
9. Conditional statements
10. Short-circuiting logic operators
11. Structs
12. Strings
13. Arrays
14. Discriminated union-type constructor

15. Pattern matching

16. Loops

17. Functions

The code generation is implemented in the file *src/wasm/WasmCodegen.fs*. The function *codegen* initiates the process and sets up the essential structure of the module. This includes defining the module's main function *_start* and adding all generated code based on the top-level scope of the Hygge program to the body of the *_start* function instance. Moreover, it adds the *heapBase* and *exitCode* as global variables and exports them.

The algorithm recursively propagates a typed AST to produce an IR of the *target language*. When a node is evaluated, the *doCodegen* function is recursively applied until reaching the tree's leaf nodes.

4.3.6.1 Literal values

A unit value will produce no additional code and will, therefore, return the current module without modifications. When a Literal value is encountered in the AST the value is pushed to the stack. Booleans are represented by an integer where $0 = false$ and all other values are truthy. Integer and boolean values are pushed with the `i32.const` instruction and floating points with the `f32.const` instruction. The cases in the code generation algorithm can be seen in code snippet 4.17.

```
| IntVal i -> // push 'i' to the stack
    m.AddCode([ (I32Const i, $"push %i{i} on stack") ])
| BoolVal b -> // push 'b' to the stack
    let v = if b then 1 else 0 // ensure be is either 1 or 0
    let s = if v = 1 then "true" else "false" // just for the comment
    m.AddCode([ (I32Const(v), $"push %s{s} on stack") ])
| FloatVal f -> // push 'f' to the stack
    m.AddCode([ (F32Const f, $"push %f{f} on stack") ])
```

Figure 4.17: Handling literal values

4.3.6.2 Negative numbers

Negative integer and floating point values are handled as a unary negation[55]. In the AST, a value can be encapsulated in a `Neg`-node, meaning the value is essentially a negative number. In case the `Neg`-node is directly containing a `IntVal` or `FloatVal`, we can statically add the sign to the number in the instruction, making sure that no further computation is needed. In cases where it does not directly contain literal values, we multiply the entire result by -1 at runtime, ensuring the value has the correct sign. The code handling negative numbers can be seen in code snippet 4.18.

```
| Neg({ Node.Expr = IntVal(v); Node.Type = TInt }) ->
    m.AddCode([ (I32Const(-v), $"push %i{-v} on stack") ])
| Neg({ Node.Expr = FloatVal(v); Node.Type = TFloat }) ->
    m.AddCode([ (F32Const(-v), $"push %f{-v} on stack") ])
| Neg(e) ->
    let m' = doCodegen env e m
    let instrs =
        match (expandType e.Env e.Type) with
        | t when (isSubtypeOf e.Env t TInt) -> m'.GetAccCode() @ [ (I32Const
    (-1), "push -1 on stack"); (I32Mul, "multiply with -1") ]
        | t when (isSubtypeOf e.Env t TFloat) -> m'.GetAccCode() @ [ (F32Const
    (-1.0f), "push -1.0 on stack"); (F32Mul, "multiply with -1.0") ]
        | _ -> failwith "negation of type not implemented"
    m'.ResetAccCode().AddCode(instrs)
```

Figure 4.18: Handling negative numbers

4.3.6.3 Arithmetic and logic operators

Arithmetic and logic operators are implemented in a single procedure. Consider the code snippet 4.19. To apply the operators, both the left-hand-side and right-hand-side nodes are evaluated into modules called *lhs'* and *rhs'*. Then, the proper instruction is determined based on the type of the current node and the operator used; the instruction is put in the variable *opCode*. There are arithmetic instructions that handle either signed or unsigned integers. In such cases, *HyggeWasm* will always handle the integer as signed integers.

The instructions found in the instruction accumulator of *lhs'* and *rhs'* are combined and nested inside *opCode*. To ensure that all other sections of the modules are merged correctly, the *lhs'* and *rhs'* modules are combined without their instruction accumulator, and the code encapsulated by *opCode* is appended to the module.

The operators for equality ('='), greater then ('>'), less then ('<'), less then or equal ('<='), and greater then or equal ('>=') are all implemented in a very similar fashion.

```
| Add(lhs, rhs) // addition '+'
| Sub(lhs, rhs) // subtraction '-'
| Rem(lhs, rhs) // remainder division '%'
| Div(lhs, rhs) // division '/'
| And(lhs, rhs) // and 'and'
| Or(lhs, rhs)  // or 'or'
| Xor(lhs, rhs) // exclusive or 'xor'
| Mult(lhs, rhs) as expr -> // multiplication '*'
    let lhs' = doCodegen env lhs m // code for the left-hand-side term
    let rhs' = doCodegen env rhs m // code for the right-hand-side term
    let opCode = // find instructions based on the current node's type.
        match (expandType node.Env node.Type) with
        | t when (isSubtypeOf node.Env t TInt) -> // type int
            match expr with
            | Add _ -> I32Add
            | Sub _ -> I32Sub
            | Rem _ -> I32RemS
            | Div _ -> I32DivS
            | Mult _ -> I32Mul
            | _ -> failwith "failed to find numeric int operation"
        | t when (isSubtypeOf node.Env t TFloat) -> // type float
            match expr with
            | Add _ -> F32Add
            | Sub _ -> F32Sub
            | Div _ -> F32Div
            | Mult _ -> F32Mul
            | _ -> failwith "failed to find numeric float operation"
        | t when (isSubtypeOf node.Env t TBool) -> // type bool
            match expr with
            | And _ -> I32And
            | Or _ -> I32Or
            | Xor _ -> I32Xor
        | _ -> failwith "failed to find numeric operation"
    (lhs'.ResetAccCode() + rhs'.ResetAccCode())
        .AddCode([ opCode (lhs'.GetAccCode() @ rhs'.GetAccCode()) ])
```

Figure 4.19: Code generation of arithmetic operators

Logical not The Logical NOT is a unary operation and, therefore, handled separately. It follows the same process as the binary operators with the determination that there is only one node as an argument. That argument is evaluated to a module, *m'*. The code of *m'* is nested inside the instruction *i32.eqz* to negate the boolean value of *m'*. As before, the *m'* module is used without the instruction accumulator to combine everything.

```
| Not(e) ->
    let m' = doCodegen env e m
    m'.ResetAccCode().AddCode([ I32Eqz(m'.GetAccCode()) ])
```

Figure 4.20: Code generation for logical not

Square root function The function *sqrt* only accepts floating-point values as input, which is enforced by the type-checking mechanism. WebAssembly has instructions for finding a square root of a floating point; thus, the code generation is fairly straightforward. Consider the code snippet 4.21. The argument *e* is evaluated using the current module and produces a new module *m'*. The code inside the module *m'* is then placed inside the *f32.sqrt* and added to the *m'* module. The code accumulator inside the module *m'* is reset to ensure the code is only appended once to the resulting module.

```
| Sqrt e ->
    let m' = doCodegen env e m // "e" is the argument given to the sqrt
    function
    // the code generated for "e" is placed within the f32.sqrt instruction
    m'.ResetAccCode().AddCode([ (F32Sqrt(m'.GetAccCode()), "sqrt of f32 value"
    ) ])
```

Figure 4.21: Code generation of square root function

Max and min functions The operators have the function of finding the maximum or minimum value in a set of two values.

The *max* and *min* functions are implemented similarly and are therefore handled together. The approach differs between integers and floating point values because WebAssembly includes specialized instructions for finding the maximum and minimum values for floating points.

The code snippet 4.22 shows the instruction pattern generated for finding the maximum and minimum of an integer value. Both follow the same general pattern of pushing the two values that should be applied to the operand stack in two pairs. The first pair is then used to evaluate the relationship between the two by either using the *i32.gt_s* or the *i32.lt_s*, short for greater than, and less than. The *_s* means that the instructions assume the value to be a signed integer. *i32.gt_s* leaves a *1* on the operand stack if the first argument is the largest value of the two; otherwise, it leaves a *0*. lt_s leaves a *1* when the first argument is less than the second argument, otherwise *0*. The select instruction picks the value from the second pair of values remaining on the stack.

```
;; max
i32.const 12 ;; push 12 on stack
i32.const 10 ;; push 10 on stack
i32.const 12 ;; push 12 on stack
i32.const 10 ;; push 10 on stack
i32.gt_s ;; which value is greater?
select ;; select the greater value

;; min
i32.const 12 ;; push 12 on stack
i32.const 10 ;; push 10 on stack
i32.const 12 ;; push 12 on stack
i32.const 10 ;; push 10 on stack
i32.lt_s ;; which value is smaller?
select ;; select the lesser value
```

Figure 4.22: Wasm instructions for max and min on intergers

When compiling for floating point values, the *f32.max* and *f32.min* instructions can be used. Therefore, the instructions needed to perform the minimum or maximum computation are reduced to only three. The two input values are pushed to the stack and then consumed by one of the before-mentioned instructions, leaving the correct value on the stack.

```
;; max
f32.const 42.000000 ;; push 42.0 on stack
f32.const 12.000000 ;; push 12.0 on stack
f32.max ;; select the greater value

;; min
f32.const 42.000000 ;; push 42.0 on stack
f32.const 12.000000 ;; push 12.0 on stack
f32.min ;; select the lesser value
```

Figure 4.23: Wasm instructions for max and min on floating points

4.3.6.4 Variables

When a variable is compiled, the process generates instructions to retrieve its value, based on where it is stored. Consider the code sample 4.24. The *VarStorage* from the code generation environment is inspected to determine the storage type. A variable can be stored as a *local*, *global*, or closure *offset* in memory.

In the case of the *local* or *global* storage types the data can be fetched by using a label that *VarStorage* maps to, and the instructions *local.get* or *global.get* can be used to retrieve the value.

In the case of the *offset* storage type, the address of the closure environment will be available in the function's first argument. The reason for this will be explained in section 4.3.6.19. With the address of the closure environment, the *offset* can be statically applied, directing it to the right position in memory. Subsequently, the load instruction can be used to fetch the value.

Upon returning, the instructions are added to the current module, creating a new module with the access code appended.

```
| Var name ->
    let instrs: List<Commented<WGF.Instr.Wasm>> =
        // find the variable in the storage environment
        match env.VarStorage.TryFind name with
        // push local variable on stack
        | Some(Storage.Local l) -> [ (LocalGet(Named(l)), $"get local var: {l}") ]
        // push global variable on stack
        | Some(Storage.Global l) -> [ (GlobalGet(Named(l)), $"get global var: {l}") ]

        | Some(Storage.Offset(i)) -> // push variable from offset on stack
            // get load instruction based on type
            let li: WGF.Instr.Wasm =
                match (expandType node.Env node.Type) with
                | t when (isSubtypeOf node.Env t TFloat) ->
                    F32Load_(None, Some(i * 4), [ (LocalGet(Index(0)), "get env pointer") ])
                | _ -> I32Load_(None, Some(i * 4), [ (LocalGet(Index(0)), "get env pointer") ])

            [ (li, $"load value at offset: {i * 4}") ]
        | _ -> failwith "could not find variable in var storage"

    m.AddCode(instrs) // append instructions to module
```

Figure 4.24: Code generation for variables

4.3.6.5 Input/Output and interaction with host system

All functionality relying on the host system is fairly trivial in code generation due to the WebAssembly import feature. Functions are imported by name and namespace. The import statement must match the function signature that the host system provides. These function signatures are either based on WASI or HyggeSI.

WGF will make sure to place imports grouped at the top of the WAT module. The imported functions can be used globally in the module. *WGF* makes sure that imports are unique, meaning that multiple imports of the same function will result in only one import statement in the WAT module. When a function is imported, it can be invoked from within the module using the *call* instruction supplied with the label of the function.

Read integer or floating point value as input When encountering a `ReadInt`-node, the *readInt* function is imported and invoked. The embeddeder will retrieve an integer value and then return control to the module instance. Code snippet 4.25 shows how this is implemented for HyggeSI. While HyggeSI will push the input value directly to the operand stack, WASI will put the value in an allocated memory space, which is then loaded to the stack. The read float operation is identical, with the only difference being the use of the *readFloat* function from HyggeSI.

Printing to console The functions *print* and *println* print to the standard output. See code snippet 4.26 showing the implementation.

The functions *print* and *println* can handle multiple types, including *float*, *int*, and *string*. This is reflected in the code generation process, where the functionality must be mapped to type-specific functions in the HyggeSI. Printing follows a similar pattern of importing the

```
| ReadInt ->
    m   // current module 'm'
        // import and call to host function
        .AddImport(getImport "readInt") // import 'readInt' function
        .AddCode([ (Call("readInt", m.GetAccCode()), "call host function") ])
```

Figure 4.25: Read integer value from outside module

required function and subsequently applying it. The expression representing the argument of the *print* or *println* function is evaluated. In the case that the expression evaluates to a *float* or an *int*, the instruction produced will, at runtime, evaluate to a `i32` or `f32` value on the stack. In the case of a string, an `i32` representing an offset in *linear memory* is left on the stack.

```
| PrintLn e
| Print e ->
    let m' = doCodegen env e m
    let nl = if node.Expr = PrintLn e then 1 else 0 // use new line if printLn
    match (expandType e.Env e.Type) with
    | t when (isSubtypeOf node.Env t TFloat) ->
        m' // perform host (system) call
            .ResetAccCode()
            .AddImport(getImport "writeFloat") // import writeF function
            .AddCode([ (Call("writeFloat", m'.GetAccCode() @ [ (I32Const nl, "
    newline") ]), "call host function") ])
    | t when (isSubtypeOf node.Env t TString) ->
        let m'' = // import writeS function
            m
                .AddImport(getImport "writeS")
                .AddCode(
                    // push string pointer to stack
                    [ (I32Load(m'.GetAccCode()), "Load string pointer") ]
                    @ [ (I32Load_(None, Some(4), m'.GetAccCode()), "Load
    string length")
                        (I32Const nl, "newline") ]
                )
        (m' ++ m'') // perform host (system) call
            .ResetAccCode()
            .AddCode([ (Call("writeS", m''.GetAccCode()), "call host function"
    ) ])
    | _ ->
        let m'' = m'.AddImport(getImport "writeInt") // import writeInt
    function
        m'' // perform host (system) call
            .ResetAccCode()
            .AddCode([ (Call("writeInt", m''.GetAccCode() @ [ (I32Const nl, "
    newline") ]), "call host function") ])
```

Figure 4.26: Printing to console using host system

The generated code is placed within the *call* instruction alongside a constant value signifying whether the behavior of *print* or *println* should be used. When the function is invoked at runtime, the values will be consumed by the *call* instruction.

4.3.6.6 Assignments

An assignment is done in Hygge by using the following syntax: $target \leftarrow value$. An assignment can be of the three kinds shown in table 4.1.

Assignment target	**Syntax**
Variable	`x <- value`
Array element	`arrayElem(arr, 0) <- value`
Structure (*struct* or *tuple*)	`x.f <- value` or `x._1 <- value`

Table 4.1: Assignment Operations

To understand how each of these cases is implemented, they will be addressed individually, starting with assigning to a variable.

In the AST, an assignment is expressed by a `Assign` node. The `Assign` node carries a *target* and *value*, the expression of *target* must evaluate to one of the scenarios shown in table 4.1. Pattern matching is used to determine how to proceed.

Assignment to variable When a variable is on the left-hand side of the assignment expression, the value on the right-hand side should be written to that variable. Consider the code snippet 4.27.

The Var-node is unpacked to obtain the name of the target variable. The *VarStorage* determines the storage type based on the name. The lookup should result in a label of a variable or an offset. Thus, only three scenarios have to be considered:

- The variable is local
- The variable is global
- The variable is part of a closure

For a local variable, the right-hand-side node of the `Assign`-node is evaluated to a value. That value is then stored in the local target variable using the instruction `local.tee`. This will leave the value stored on the stack. It is the same for a global variable with the only difference that the `local.tee` instruction is substituted for a `global.set` and `global.get`.

For a target variable inside a closure, the address of the closure is pushed to the stack, and the offset corresponding to the particular variable is used to compute the address in memory where the variable's current value is located. The right-hand-side node of the `Assign`-node is evaluated to a value stored using the address. Finally, the newly stored value is pushed to the stack.

```
| Var(name) -> // left-hand-side is a variable
    match env.VarStorage.TryFind name with
    | Some(Storage.Local l) -> // is local var
        value'
            .ResetAccCode()
            .AddCode([ (LocalTee(Named(l), value'.GetAccCode()), "set local
    var") ])
    | Some(Storage.Global g) -> // is global var
        value'
            .ResetAccCode()
            .AddCode(
                [ (GlobalSet(Named(g), value'.GetAccCode()), "set global var")
                  (GlobalGet(Named(g)), "get global var") ]
            )
    | Some(Storage.Offset(i)) -> // is offset var
        // store value in linear memory ..
    | _ -> failwith "not implemented"
```

Figure 4.27: Assign a value to variable

Assignment to array element In this case, the array element access is encapsulated (by an assignment) as the left-hand side expression.

```
| ArrayElement(target, index) ->
    let selTargetCode = doCodegen env target m // address to target array
    let indexCode = doCodegen env index m // index to position
    let rhsCode = doCodegen env value m // value on the right-hand-side
    // Check index >= 0 and index < length
    let indexCheck = // code that check that index is in bounds
    let storeInstr = // determine store instruction based on array type
        match (expandType value.Env value.Type) with
        | t when (isSubtypeOf value.Env t TFloat) -> F32Store
        | _ -> I32Store
    let instrs = // Calculate address of position and store new value there
        [ (storeInstr (
              [ (I32Add(
                     [ (I32Load(selTargetCode.GetAccCode()), "load data
    pointer") // load the address to array
                       (I32Mul( // multiply index by 4
                            indexCode.GetAccCode() // index on stack
                            @ [ (I32Const 4, "byte offset") ]
                        ),
                        "multiply index with byte offset") ]
                 ),
                 "add offset to base address") ]
              @ rhsCode.GetAccCode() // leave value on the stack
          ),
          "store value in elem pos") ]
          @ rhsCode.GetAccCode()
    // combine the compiled code
    (rhsCode.ResetAccCode() + indexCode.ResetAccCode() + selTargetCode.
    ResetAccCode()).AddCode(indexCheck @ instrs)
```

Figure 4.28: Assign a value to element in array

The *index* is checked to ensure it is in the bounds of the array, that *target* points to. The *index* is multiplied by 4 bytes and added to the address of *target*. The new value is stored at the calculated address and left on the stack. The position has to be calculated at runtime since the address is only known then. Depending on the type of the target array, the store instruction used is either *i32.store* or *f32.store*.

Assignment to structure Code example 4.29 shows how this is represented in a Hygge program.

```
// creating a struct with the field 'f' with the value 3
let s: struct {f: int} = struct {f = 1 + 2};
s.f <- 5; // assign a new value to field 'f'
assert(s.f = 5) // assert the value of field 'f' is now 5.
```

Figure 4.29: Assign a value to struct field in Hygge

An assignment to a structure field is an `Assign`-node where the left-hand side is a field selection. The FieldSelect is deconstructed with pattern-matching and consists of a target and a field. The target is a node that resolves to the address of the struct and the field is the name of the selected field as a string. The right-hand-side node of the assign-node is evaluated to a value. The value is stored in the memory space that matches the struct field given.

The target type is then expanded to evaluate the actual type which allows for subtyping. At this point, the compiler should fail if the target is not of type `TStruct` since this is the only valid target type of this operation. The `TStruct` type contains the type information about the struct, meaning a list of the name and type of each field in the struct.

The index of the applicable field is found; this is done because each field's data is placed sequentially in order into memory, each field occupying a fixed number of bits, in this case 32-bit. Therefore, the index can be used to compute the byte offset of the field statically.

Then, the left-hand side expression, which represents the type of the value to store in the struct, is expanded and evaluated. The type of value is needed since WebAssembly has different instructions to load and store integers and floating points.

Storing integers and floating point values in a field are similar; therefore, the process only describes ones. Note that the load and store instructions differ in the two cases.

First, the value is stored. The store instruction will take an optional offset as an immediate offset computed based on the index. The store instruction consumes a memory address and a value to store from the operand stack. Therefore, the compiled code from the target and the right-hand side are nested inside the store instructions. This is all nested in a load that, similarly to the store, takes an offset and a memory address and leaves the value just stored on the operand stack. This could also have been accomplished by just leaving the instructions of the right-hand side node as the last piece of code, though this store load pattern ensures that the value found in memory is left on the operand stack.

```
| FieldSelect(target, field) ->
      let selTargetCode = doCodegen env target m

      /// Code for the 'rhs' expression of the assignment
      let rhsCode = doCodegen env value m

      match (expandType target.Env target.Type) with
      | TStruct(fields) ->
          /// Names of the struct fields
          let fieldNames, _ = List.unzip fields
          /// offset of the selected struct field from the beginning of
          /// the struct
          let offset = List.findIndex (fun f -> f = field) fieldNames

          /// Assembly code that performs the field value assignment
          let assignCode =
              match (expandType name.Env name.Type) with
              | t when (isSubtypeOf value.Env t TUnit) -> [] // Nothing to do
              | t when (isSubtypeOf value.Env t TFloat) ->
                  // load value just to leave a value on the stack
                  [ (F32Load_(
                        None,
                        Some(offset * 4),
                        selTargetCode.GetAccCode()
                        @ [ (F32Store_(None, Some(offset * 4), selTargetCode.GetAccCode() @ rhsCode.GetAccCode()),
                             "store float in struct") ]
                     ),
                     "load float from struct") ]
              | _ ->
                  // load value just to leave a value on the stack
                  [ (I32Load_(
                        None,
                        Some(offset * 4),
                        selTargetCode.GetAccCode()
                        @ [ (I32Store_(None, Some(offset * 4), selTargetCode.GetAccCode() @ rhsCode.GetAccCode()),
                             "store int in struct") ]
                     ),
                     "load int from struct") ]

          // Put everything together
          assignCode ++ (rhsCode.ResetAccCode() + selTargetCode.ResetAccCode())
      | _ -> failwith "failed to assign to field"
```

Figure 4.30: Assignment to a struct field

Multiple assignment Multiple assignment refers to a type of assignment statement where a single value is assigned to two or more variables simultaneously. In Hygge, this can be observed as $x \leftarrow y \leftarrow z \leftarrow 0$, where the variables x, y, and z are assigned zero in a single statement.

```
let mutable x: float = 1.0f;
let mutable y: float = 2.0f;
let mutable z: float = 3.0f;

x <- y <- z <- x + y + z
```

Figure 4.31: Example of multiple assignments in Hygge

The code example 4.31 results in the sequence of instructions shown in 4.32. First, the initial values are set for each variable as part of their declaration by the let-binders. Then, the right-hand side of the multiple assignment is evaluated. In this case, this means computing the sum of the involved variables, becoming the value that should be assigned to all variables left of an assignment. The assignment itself is done using the `local.tee` instruction. It will set the variable with a specific label and leave the value on the stack. This value can then be used for the next assignment, making it possible to chain assignments indefinitely. The last `local.tee` leaves the result on the stack.

```
    ;; Code produced by the let binders.
    ;; Set the initial values of each variable
    f32.const 1.000000
    local.set $var_x
    f32.const 2.000000
    local.set $var_y
    f32.const 3.000000
    local.set $var_z

    ;; Computing sum of (x + y + z)
    local.get $var_x
    local.get $var_y
    f32.add
    local.get $var_z
    f32.add

    ;; Do the assignment to the variables (x, y, z)
    local.tee $var_z
    local.tee $var_y
    local.tee $var_x ;; <- leaves result on stack
```

Figure 4.32: Example of multiple assignment in WAT

This implementation allows all the assignment types shown in table 4.1 to work seamlessly since they all evaluate to a single value. The code in 4.32 was generated from test case *013-assign-multi-vars-combine.hyg*.

4.3.6.7 Compute-assign operators

Compute-assign operators are implemented by rewriting an `Assign`-node to a regular assignment. The rewritten assignment is assigned to the left-hand side. The right-hand side of `Assign`-node is given a node representing an arithmetic operation between the left-hand and right-hand sides.

```
| AddAsg(lhs, rhs) ->
    doCodegen
        env
        { node with
            Expr = Assign(lhs, { node with Expr = Add(lhs, rhs) }) }
        m
```

Figure 4.33: Gode generation for addition assignment operator

Code example 4.33 shows how the addition assignment operator is implemented. The implementation of the other compute-assign operators is very similar and the only difference is the arithmetic operation node, in this case, it is an *Add*-node.

4.3.6.8 Pre- and post-increment and decrement operators

Pre- and post-increment and decrement operators are rewritten to an assignment that either adds or subtracts 1 from the left-hand side, representing the target variable. An assignment leaves the variable's value on the stack after it has been written to. This means that the `pre`-operator can be rewritten to an assignment without any further modifications. On the other hand, the `post`-operator has to leave the value before it has been modified. This is accomplished as seen in code example 4.34, by first leaving the evaluated value of the variable and then, at the end, dropping the value that is produced by the assignment.

This leads to an unnecessary code and could also be done by using a *virtual register*. This would be a much more complex code since it would have to handle all the cases that the assignment does. Furthermore, some of the unnecessary instructions are removed later in the optimization phase. Therefore, the simpler way of handling this was chosen.

```
| PostIncr(e) ->
  let instrs =
    match (expandType e.Env e.Type) with
    | t when (isSubtypeOf e.Env t TInt) ->
      let assignode =
        { node with
          Expr =
            Assign(
              e,
              { node with
                Expr = Add(e, { node with Expr = IntVal 1 }) }
            ) }
      (doCodegen env e m) + (doCodegen env assignode m)  // <-- putting original value on the stack first
    | t when (isSubtypeOf e.Env t TFloat) ->
      let assignode =
        { node with
          Expr =
            Assign(
              e,
              { node with
                Expr = Add(e, { node with Expr = FloatVal 1.0f }) }
            ) }

      (doCodegen env e m) + (doCodegen env assignode m) // <-- putting original value on the stack first
    | _ -> failwith "not implemented"
  instrs.AddCode([ Drop ])
```

Figure 4.34: Gode generation for post-increment

4.3.6.9 Conditional statements

WebAssembly provides an if-else statement which makes the translation from the AST node to the IR fairly straightforward. Consider code sample 4.35. The condition-node, ”true”-branch node, and ”false”-branch node are all evaluated. The WebAssembly *If* instruction is a *control instruction* and therefore requires a so-called *block type*. This is the type that the if-block will evaluate to. This is done by expanding the node type and then mapping the type to subsequent *WGF* type. Then, all the instructions can be resembled into the *WGF* If representation. The ”false”-branch code is encapsulated in an option because this value doesn’t need to be there for *WGF* to produce valid code.

```
| AST.If(condition, ifTrue, ifFalse) ->
    let m' = doCodegen env condition m // module of the condition
    let m'' = doCodegen env ifTrue m // module of the "true"-branch
    let m''' = doCodegen env ifFalse m // module of the "false"-branch
    // get the return type of the ifTrue branch and, subsequently, the ifFalse
     branch
    let resultType = (expandType node.Env node.Type)
    // find block type
    let mappedResultType = mapType resultType
    // combine modules
    (m' + m'' + m''')
        .ResetAccCode() // reset the instruction accumulator
        // reassemble the module components
        .AddCode(C [ (If(mappedResultType, m'.GetAccCode(), m''.GetAccCode(),
    Some(m'''.GetAccCode()))) ])
```

Figure 4.35: Code generation for conditional statements

4.3.6.10 Assert function

The intrinsic assert function is used to validate the correct execution of test programs. An example can be seen in code snippet 2.3. The code snippet 4.36 shows the generated code of an invocation of the assert function. The expression given to the assert function is evaluated and placed as the condition of the if instruction. This condition is then inverted, so only if the boolean expression is false will the 'then'-block be executed. If the 'then'-block is executed is the `exit_code` set to 42 and the program is terminated with a trap. The number 42 signals an error to the host environment. The program will have exit code 0 at normal execution.

```
(if
  (i32.eqz ;; invert assertion
    (i32.eq ;; equality check
        ;; actual condition
        ;; ...
    )
  )
  (then
    (global.set $exit_code ;; set exit code
      (i32.const 42) ;; error exit code push to stack
    )
    (unreachable) ;; exit program
  )
)
```

Figure 4.36: Code generation for the assert function

4.3.6.11 Short-circuiting logic operators

As part of the logic operators, short-circuiting conjunction and disjunction were implemented just as regular logical operators. The nodes comprised a left-hand and right-hand side expression that are both expected to evaluate to a boolean value.

The logical conjunction *and* (&&) and The logical disjunction *or* (||) are different from the regular *and* and *or*, because they use short-circuit evaluation[56].

In the case of the short-circuiting logical conjunction, if the left-hand side evaluates to *true*, the next boolean expression must be evaluated; otherwise, it should stop evaluation immediately. Consider the implementation shown in 4.37. The *If* instruction is used to accomplish the short-circuiting behavior. This is done by taking the left-hand side and using it as the if statement's condition. If the value is true, the value 1 is pushed to the stack. No further actions are taken. Otherwise, if the 'else'-branch is taken and the right-hand side is evaluated, this will either leave a 1 or 0 on the stack.

In the case of logical disjunction, the left-hand side expression is still used as the condition, but a truthy value results directly in a 1 pushed to the stack, and evaluation is stopped. Otherwise, the evaluation of the expression continues in the 'else'-branch.

```
// short circuit and
| ShortAnd(e1, e2) ->
    doCodegen
        env
        { node with
            Expr = AST.If(e1, e2, { node with Expr = IntVal 0 }) }
        m
// short circuit or
| ShortOr(e1, e2) ->
    doCodegen
        env
        { node with
            Expr = AST.If(e1, { node with Expr = IntVal 1 }, e2) }
        m
```

Figure 4.37: Gode generation for short-circuiting operators

4.3.6.12 Structs

This section will describe the code generation of *Structs*. Notice that assignments to a *struct* is described in section 4.3.6.6.

Tuples are enabled through *structs* and are placed in memory as a *struct*. This conversion is done in the compiler's front-end and is therefore not further described in this report.

Struct constructor *Structs* is one of the basic building blocks for Hygge. *Structs* are parsed by reference and are dynamically allocated in memory. The needed space is computed as $number\ of\ fields \times 4\ bytes$. The memory allocation process will yield an address pointing to the position in memory where the data of the *struct* can be placed. The memory address has to be used multiple times and is therefore stored in a `i32` local temporary variable. The temporary variable can be seen as a *virtual register*.

After this, the code for initializing each field in memory is generated by folding over indexed field names and their types. First, the address of the current field is found by the calculation below:

$$field_{offset} = (index \times 4\ bytes) \tag{4.1}$$

$$field_{pos} = field_{offset} + address\ of\ struct \tag{4.2}$$

The $field_{offset}$ can be computed during compilation because the index is known at compile time. However, the addition operation must occur during runtime where the *struct's* address is determined. To obtain the address of the *struct*, we load it using the temporary local variable. Next, we generate the code for initializing the field's data, which, when executed, will resolve to its value. Finally, this resolved value is stored in memory at the calculated address. Based on the value type that must be stored, either a *f32.store* or a *i32.store* is used. The code generated for initializing the fields is accumulated and concatenated sequentially. At the very end, the temporary local variable pointing to the start address is pushed to the stack to leave the address on the stack.

Struct field access Accessing a field in a *struct* involves reading a specific field by name. Consider code snippet 4.38. The code generation of the field access begins by evaluating the target address. This is the value left on the stack by the *struct* constructor in Section 4.3.6.12. The type of the target is checked. This should always be of type `TStruct`. The field's offset is determined by its index, indicating its position within the *struct*. The byte offset can then be statically calculated as described in Section 4.3.6.12. Depending on the field type, either an `i32` or an `f32` load instruction, is used to retrieve the value from memory.

```
| FieldSelect(target, field) ->
    let selTargetCode = doCodegen env target m // address to target struct

    let fieldAccessCode = // code for accessing the struct field
        match (expandType target.Env target.Type) with  // expanding and matching target type
        | TStruct(fields) -> // target is a struct
            let fieldNames, fieldTypes = List.unzip fields // unzipping names and types
            let offset = List.findIndex (fun f -> f = field) fieldNames // find index of field

            // Retrieve value of struct field
            match fieldTypes[offset] with
            | t when (isSubtypeOf target.Env t TUnit) -> [] // Nothing to do
            | t when (isSubtypeOf target.Env t TFloat) -> // load float
                [ (F32Load_(None, Some(offset * 4), selTargetCode.GetAccCode()), $"load field: {fieldNames[offset]}") ]
            | _ -> // load int (address), bool
                [ (I32Load_(None, Some(offset * 4), selTargetCode.GetAccCode()), $"load field: {fieldNames[offset]}") ]
        // target should only could be a struct
        | t -> failwith $"BUG: FieldSelect codegen on invalid target type: %O{t}"

    // Put everything together: compile the target, access the field
    selTargetCode.ResetAccCode()
    ++ m.AddCode(
        C [ Comment "Start of field select" ]
        @ fieldAccessCode
        @ C [ Comment "End of field select" ]
    )
```

Figure 4.38: Code generation of struct field access

4.3.6.13 Strings

Strings are statically allocated at compile time and are written to the allocated space in memory when the module is instantiated. This is accomplished by utilizing data section entries with *data strings* as described in 4.3.5.1. Strings are parsed by reference.

Figure 4.39 shows how these are represented in the WAT module. Strings are interpreted as the *Unicode* standard `UTF-8`. This allows this implementation of Hygge to represent the entire Unicode character set. An earlier version of the implementation strings was interpreted as the *Unicode* `UTF-16` format. See appendix F for more details.

```
(data (i32.const 0) "\0c\00\00\00\12\00\00\00\12\00\00\00")
(data (i32.const 12) "hygge println test")
(data (i32.const 30) "\2a\00\00\00\10\00\00\00\10\00\00\00")
(data (i32.const 42) "hygge print test")
(data (i32.const 101) "\71\00\00\00\03\00\00\00\01\00\00\00")
(data (i32.const 113) "0x2705") ;; Unicode Character "U+2705"
```

Figure 4.39: Hexadecimal data strings in WebAssembly WAT module

When the module has been instantiated, the data will be placed in memory as a structure containing the tuple (d, s, l), where *d* is the address of the string, *s* is the size of the string in bytes and *l* is the number of characters in the string. How the tuple (d, s, l) is stored in memory is shown in Figure 4.40. The arrow pointing to the data address field is the pointer left on the stack when evaluating the *StringVal*-node.

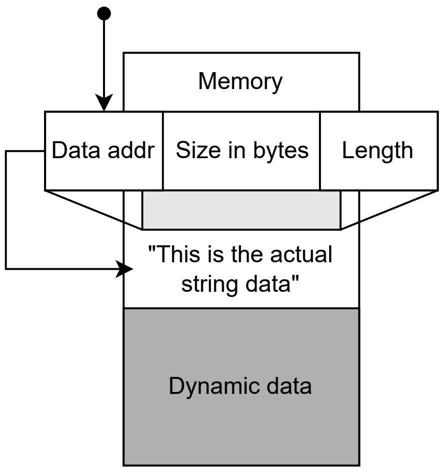
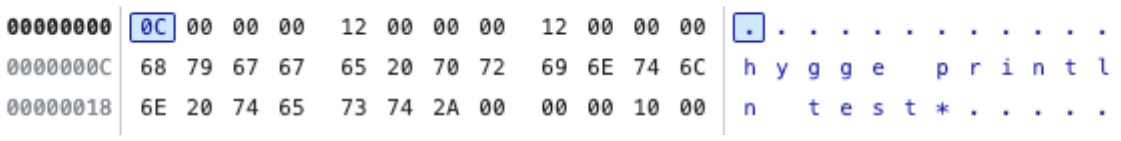


Figure 4.40: Strings in memory

As part of its development tools, Google Chrome has a memory inspector. fig. 4.41 shows an example of this. In this case, the length and the size in bytes of the string is 18, meaning that it is 12 in hexadecimal. This value will be the same when only American Standard Code for Information Interchange (ASCII) characters are used.



```
00000000  0C 00 00 00  12 00 00 00  12 00 00 00  . . . . . . . . . . . .
0000000C  68 79 67 67  65 20 70 72  69 6E 74 6C  h y g g e   p r i n t l
00000018  6E 20 74 65  73 74 2A 00  00 00 10 00  n   t e s t * . . . . .
```

Figure 4.41: String in Arraybuffer shown in Chrome dev-tools

The intrinsic operation *stringLength(s)* is used for obtaining the length of a string.

```
| StringLength e ->
    let m' = doCodegen env e m

    m'
        .ResetAccCode()
        .AddCode([ (I32Load_(None, Some(8), m'.GetAccCode()), "load string
    length") ])
```

Figure 4.42: Code generation of StringLength function

Consider code snippet 4.42. Foremost, the node *e* should evaluate to an address pointing to the structure containing the tuple (d, s, l), pushing the address to the *operand stack*. This address is then used to load the string length by applying an offset of 8 bytes.

4.3.6.14 Arrays

This section will describe the code generation of arrays. Arrays are dynamically allocated in *linear memory* and parsed by reference. This encompasses an array constructor to create array structures in memory and common array operations. Arrays in Hygge are zero-indexed and can only hold one data type for each array.

The implemented functionality is listed below and matches what is found in the specification of *HyggeWasm* found in appendix A.1.

- Create and initialize an array
- Retrieve the length of an array
- Read from a specific index in an array
- Write to a specific position in an array
- Slice array into a sub-array

A table of the operations and their syntax in *Hygge* can be seen in table 2.6.

Array constructor The array constructor will dynamically allocate memory during runtime in the fashion described in section 3.3.4.1, followed by populating the designated memory space with a specified initialization value. An example of an Array in Hygge can be seen in 4.43. The constructor takes two arguments: the length of the array and an initial value.

```
let size: int = 8;
let initValue: int = 0;
let arr: array {int} = array(size, initValue)
```

Figure 4.43: Use of the Array constructor

Hygge does not allow arrays of length zero. Therefore, the length is checked at runtime to be $length > 0$. If this is not the case, the program's execution is terminated. An array instance is implemented as a *struct* containing a pointer to the actual data in memory and the array's length. See Figure 4.44 for an illustration of this.

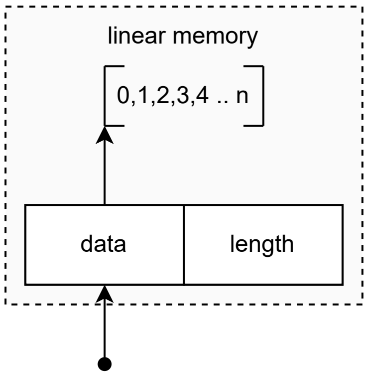


Figure 4.44: Structure in memory defining array

The memory segment designated for the array instance is initialized at runtime by iterating through memory in 4-byte chunks and storing the initial value provided in the constructor at each position.

WebAssembly contains instructions for bulk memory operations such as `memory.fill`, but this instruction stores per byte [57], and it was therefore impossible to align with 4 bytes.

Array slice Array slices work similarly to slices in *JavaScript* [58], producing a shallow copy of the original array [59], thus referencing the same actual data in memory as the original array. This is illustrated in fig. 4.46, which shows an array of 12 elements that has been sliced into a subarray of size 4. If the sliced array changes its value at index zero, the original array will contain the new value at index 4. A test program can be seen in code snippet 4.45.

An array can be sliced unlimited times and can overlap each other. Since the sliced array instance has the same structure as an ordinary array instance, all array operations can also be performed on it, including slice.

```
// create the two arrays
let original: array {int} = array(12, 0);
let sliced: array {int} = arraySlice(original, 4, 8);
// fill the first array with numbers
let mutable i: int = 0;
while (i < arrayLength(original)) do {
    arrayElem(original, i) <- i;
    i <- i + 1
};
// assert that the second array contains the correct numbers
assert(arrayElem(sliced, 0) = 4);
assert(arrayElem(sliced, 1) = 5);
assert(arrayElem(sliced, 2) = 6);
assert(arrayElem(sliced, 3) = 7)
```

Figure 4.45: Slicing array in Hygge

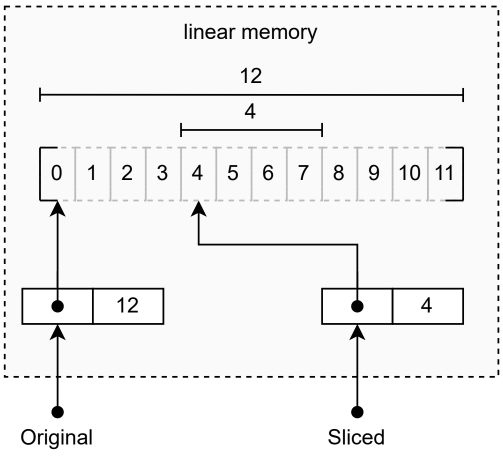


Figure 4.46: Example of array instance and sliced array instance in memory

Ensuring data integrity when working with arrays When reading and writing to an array element, there is an index to an element supplied. To ensure that these array operations only operate on the defined array's memory space, the index should be validated at runtime to be within the array's bounds.

The bounds of the array index are defined as:

- Index is greater or equal to 0
- Index is smaller than the array length

The bounds are implemented as guards with inverted conditions of the above mentioned bounds. If a guard condition is computed to a truthy value, the `exit_code` is set to *42* and then a trap is triggered with an `unreachable` instruction. An example of the code generated for checking the bounds can be seen in 4.47.

```
(if
(i32.lt_s ;; check if (index < 0)
    (i32.const 0) ;; push index to stack
    (i32.const 0) ;; push zero to stack
)
(then
  (global.set $exit_code ;; set exit code
    (i32.const 42) ;; error exit code pushed to stack
  )
  (unreachable) ;; exit program
)
)
(if
  (i32.ge_s ;; check if (index > length)
    (i32.const 0) ;; push index to stack
    (i32.load offset=4
      (global.get $arr_var) ;; get array length
    )
  )
(then
  (global.set $exit_code ;; set exit code
    (i32.const 42) ;; error exit code push to stack
  )
  (unreachable) ;; exit program
)
)
```

Figure 4.47: Checking bound of array

4.3.6.15 Discriminated union-type constructor

The discriminated union-type constructor produces a union-type instance. A union-type instance consists of a label and some data. Notice that the data can be a *unit*, effectively representing a null value.

Consider the code snippet 4.48 to elucidate the concept further. Code snippet 4.48 show cases of the use of the union-type constructor in Hygge. The constructor is utilized in lines 7 and 8. In line 7, the constructor is used with the label ”Some” and the value of 42, and in line 8, the constructor is used with the label ”None”, encapsulating a unit value.

A union-type instance can be used with pattern matching. On line 10 the first union-type instance stored in variable *x* is used with pattern matching. This will match the instance against the patterns inside the match block. In this scenario, the first pattern will match the instance, leading to printing 42.

```
type t = union {
    Some: int;
    None: unit
};

// union-type constructor
let x: t = Some{42};
let n: t = None{()};

match x with {
    Some{v} -> println(v);
    None{_} -> println("None")
}
```

Figure 4.48: Use of the Union constructor

WebAssembly does not directly support union types. Consequently, a method for representing instances of union types must be designed.

The approach by HyggeWasm to represent the union-type instance, is with a *struct* containing two fields. Therefore, union-type instances are dynamically allocated into *linear memory*. The *struct* will have the size of $2 * 4$ bytes since every field is 32 bit.

To accomplish this, the expression is rewritten to a *struct* to reuse the code generation for structures. The first field contains the label, and the second is the data. Code snippet 4.49 shows how this is implemented.

Comparing the label's string values is hard to do in WebAssembly. Thus, the labels are not stored as string values. An unique integer identifier is assigned to each label to resolve this issue. This allows for comparisons of labels by comparing the distinct integer label identifiers. The identifiers are generated using *Util.genSymbolId*. This functionality was part of the original *RISC-V* back-end.

In the case of a unit value, the expression is substituted for an integer value, simply because the structure expects a value that can be stored in memory. This value is never read. It could be a further work improvement to let structs handle unit values.

```
| UnionCons(label, expr) ->
    // compute label id
    let id = env.SymbolController.genSymbolId label
    // create node for label
    let idNode = { node with Expr = IntVal id }
    // in case there is no data, aka a unit - we need to add a zero
    let data =
        match (expandType expr.Env expr.Type) with
        | TUnit -> { node with Expr = IntVal 0 } // unitvalue
        | _ -> expr
    // rewrite as struct
    let structNode =
        { node with
            Expr = Struct([ ("id", idNode); ("data", data) ]) }
    // codegen structNode
    C [ Comment "Start of union contructor" ]
    ++ (doCodegen env structNode m).AddCode([ (Comment "End of union
    contructor") ])
```

Figure 4.49: Code generation for the discriminated union-type constructor

4.3.6.16 Pattern matching

Implementing pattern matching in WAT involves encapsulating all cases within a *block* structure. Every case becomes an if-then block where the condition checks that the case label identifier equals the label of the matched union-type instance. The *block* allows the cases to branch out seamlessly, ensuring that no additional conditions need evaluation during execution. Although this might not be strictly necessary due to the uniqueness of labels and their corresponding label IDs, Hygge's type system inherently prohibits duplicate case labels in a match expression. This design eliminates the need for redundant evaluations when a case has been executed.

The pattern matching may be evaluated to a value; therefore, the *block* instruction is annotated with a result type to pass WebAssembly's type checking. Code snippet 4.50 contains a full example of how the pattern matching is represented in WAT.

```
(block $match_end (result i32) ;; <-- result type of the block
;; case for id: $1, label: Some
(if
    (i32.eq ;; check if index is equal to target
      (i32.load ;; load label
        (global.get $var_x) ;; get local var: var_x, have been promoted
      )
      (i32.const 1) ;; put label id 1 on stack
    )
  (then
    (global.set $match_var_x ;; set local var, have been promoted
      (i32.load offset=4
        (global.get $var_x) ;; get local var: var_x, have been promoted
      )
    )
    (global.set $var_i ;; set local var, have been promoted
      (i32.add
        (global.get $match_var_x) ;; get local var: match_var_x, have been
    promoted
        (i32.const 1) ;; push 1 on stack
      )
    )
    (global.get $var_i) ;; set local var, have been promoted
    (br $match_end) ;; break out of match
  )
)
;; case for id: $2, label: None
(if
    (i32.eq ;; check if index is equal to target
      (i32.load ;; load label
        (global.get $var_x) ;; get local var: var_x, have been promoted
      )
      (i32.const 2) ;; put label id 2 on stack
    )
  (then
    (global.set $match_var__ ;; set local var, have been promoted
      (i32.load offset=4
        (global.get $var_x) ;; get local var: var_x, have been promoted
      )
    )
    (i32.const 0) ;; push 0 on stack
    (br $match_end) ;; break out of match
  )
)
;; no case was matched, therefore return exit error code
(global.set $exit_code ;; set exit code
  (i32.const 42) ;; error exit code push to stack
)
(unreachable) ;; exit program
)
```

Figure 4.50: Anatomy of pattern matching

4.3.6.17 Loops

WebAssembly has some constructs that are not traditionally associated with assembly languages. Some constructs are the ones used for structured control flow. These are defined with the structured control instructions described in 2.2.10.

One of these instructions is the *loop*. An example of the loop instruction is shown in code snippet 4.51 in folded form.

```
(loop $label
    nop
)
```

Figure 4.51: Structured control with loop

The *loop* structure does not *loop* on its own. A branch instruction must be placed inside the loops block to achieve the looping behavior. Unlike the *block* structure, a branch inside a *loop* will jump to the beginning of its scope[27][p. 37].

While-Loop *block* and *loop* control structures described above are combined to achieve the behavior of a While-loop. Consider the code example 4.52.

The *block* serves as the outermost structure, defining an end-label that can be utilized for *loop* termination. Within the *block*, a nested *loop* structure is placed. This facilitates a mechanism for returning to the beginning of the *loop*.

Upon entering the loop, the condition of the while loop is evaluated as the initial step in the sequence. The condition determines if the loop should continue or end. The condition is inverted so a truthy value will result in a jump to the end of the block. If the jump is not made, it will continue by executing the instructions in the loop body. If the loop body leaves any value on the stack, a drop is inserted. This is done because control structures in WebAssembly can return a result, but Hygge does not allow any loop construct to resolve to a value. At the end, an unconditional jump is performed to reach the beginning of the loop block in order to reset the loop for the next iteration.

```
(block $loop_exit
  (loop $loop_begin
    (br_if $loop_exit ;; if false break
      (i32.eqz ;; evaluate loop condition
        ;; the condition itself
      )
    )
    ;; the loop body
    (br $loop_begin) ;; jump to the beginning of the loop
  )
)
```

Figure 4.52: Anatomy of a while loop in WAT

Do-While-Loop The implementation of do-while loops begins with the initial generation of code for the body, guaranteeing the initial execution of the body. The body should not push a value onto the stack. However, if such an occurrence arises, a drop operation is performed accordingly.

After the initial loop body, the do-while-loop operation is equivalent to the while-loop. Therefore, the do-while components are rewritten to a while loop.

```
| DoWhile(cond, body) ->

    let body' = (doCodegen env body m)
    // insert drop if body is not unit
    let mayDrop =
        if (expandType body.Env body.Type) = TUnit then
            body'.GetAccCode()
        else
            [ (Drop(body'.GetAccCode()), "drop value of the body") ]

    body'.ResetAccCode().AddCode(mayDrop)
    ++ (doCodegen env { node with Expr = While(cond, body) } m)
```

Figure 4.53: Code generation of do-while loops

For-Loop Code example 4.54 shows how code generation for a for-loop is performed. The first step is generating code for the init node. The init-node is typically used to initialize the counter variable. Variables can not be declared in the initiation scope of the for-loop due to limitations in the *parser*. If the init-node produces a value on the stack, this value is dropped. The initialization instructions generated should be executed as the first action and are consequently positioned directly before the loop block.

After these preliminary steps, the for-loop, like the do-while loop, can be rewritten as a regular while loop. The loop body is adjusted to incorporate the original loop body and the expression that typically updates the counter variable, in that specific sequence. This ensures that the update instructions are only evaluated after the body has been executed.

```
| For(init, cond, update, body) ->
    let init' = (doCodegen env init m) ;; compile init
    let mayDrop = ;; drop if init is not unit
        if (expandType init.Env init.Type) = TUnit then
            init'.GetAccCode()
        else
            [ (Drop(init'.GetAccCode()), "drop value of init node") ]

    init'.ResetAccCode().AddCode(mayDrop)
    ++ (doCodegen
        env
        {node with Expr=While(cond,{node with Expr = Seq([body; update ])})}
        m)
```

Figure 4.54: Code generation of for-loop

4.3.6.18 Functions

This Section will describe how functions and function pointers are implemented in *Hygge-Wasm*. First, all functions except the *_start* are given an internal name with the prefix ”fun_” and are ensured unique names as described in Section 3.3.9. All arguments are treated as local variables since this is how WebAssembly operates. This means they are added to the code generation environment as local variables by name. Local variables that are arguments have the prefix ’arg_’ in front of the name, making it easier to distinguish between the two.

Consider code snippet 4.55. Please note that the code snippet 4.55 omits details related to the *heap* memory mode for clarity in the description. While Hygge can have nested functions, WebAssembly can not. Therefore, the nesting is flattened so all functions are on the top level of the WAT module. The function *compileFunction* does this reorganization of functions. An example can be seen in appendix L.

All lambda terms are compiled using the *compileFunction* function. The *compileFunction* collects all information about a function. This includes a type declaration corresponding to the function signature and the compiled function body.

When compiling the function body, the code generation environment's *VarStorage* contains the arguments as local variables. All the instructions produced by compiling the body are placed in a module within a temporary code accumulator. The temporary code accumulator contains all function instructions when it returns to *compileFunction*. At this point, the body's code is not directly associated with a function name. To do this, all instructions are moved from the temporary code accumulator into a named function instance inside the *WGF* representation.

```
and internal compileFunction
    (name: string)
    (args: List<string * Type.Type>)
    (body: TypedAST)
    (env: CodegenEnv)
    (m: Module)
    (captured: string list): Module =
    // map args to their types
    // and add cenv as argument
    let argTypes': Local list = (Some("cenv"), I32) :: (argsToLocals env args)
    // create function signature
    let signature: FunctionSignature = (argTypes', mapType body.Type)
    // compile function body
    let m': Module = doCodegen { env with CurrFunc = name } body m
    // create function instance
    let f: Commented<FunctionInstance> =
        ({ locals = m'.GetLocals() // add locals
           signature = signature // add signature
           body = m'.GetAccCode() // add all instructions of the function
           name = Some(Label(name)) }, // add name to function
         $"function {name}")
    m'
        .AddFunction(name, f, true) // add function and type declaration to module
        .ResetAccCode() // reset accumulated code
        .ResetLocals() // reset locals
```

Figure 4.55: Compile function

Recursive functions For a function to be recursive in Hygge, the recursive let-binder has to be used. The function's body is compiled with a *VarStorage* that includes the function itself. This ensures that the recursive calls can be resolved within its own body.

Function pointers All imported functions are directly called by name using the *call* instruction. This can be done because it is static, meaning that one *call* instruction will always only call one particular function. Another approach has to be used when the function can be a value determined at runtime. The following sections describe how this more flexible approach is designed and implemented.

The overall idea is to use indirection in the form of a look-up table that can translate an index (integer value) to a function reference.

One of the necessary features is the ability to resolve function index functions by name. To do this, a global variable is added to the module when compiling a lambda. It has to be global since the function may be called from any other function within the module.

The name of the global variable is added to the *VarStorage* in the code generation environment. *VarStorage* will map from the variable name or function name to the pointer name. This is the only use of global variables before promoting local variables. The global variable is named ”<label of function>*ptr” and is initialized with a memory address. The address points to a chunk of memory that contains a table index of the function. The table index is put into memory as a data element. An illustration can be seen in figure 4.56. This allows the code to treat the function as a simple function that does not capture a closure environment similar to the ones that do capture. Closures are described in Section 4.3.6.19.

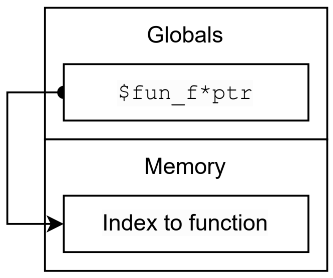


Figure 4.56: Function pointer

The function table To enable Hygge programs to parse function references, a look-up table is created when the first function reference is added. In the WAT module, the table is named *func_table*. From here on, this table is referred to as the function table.

Indirect calls via function table To call functions that reside in the function table, the *call_indirect* instruction is used. *call_indirect* will, based on the index, look up in the function table. This will, in the binary module, map to a function signature in the function section that will also contain the true address of the instructions of the function. All the instructions of the function are in a separate code section in the binary format. This is illustrated in figure 4.57.

This allows the program to determine what function to call at runtime. This approach is also used to implement function pointers in *C*/*C*++ and virtual functions in *C*++[27, p. 59]. By extension, it is also the mechanism for handling functions as first-class values in languages such as *JavaScript*.

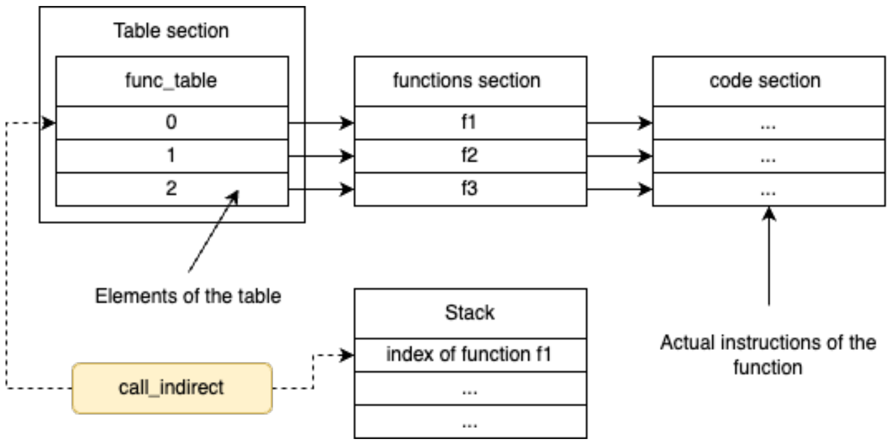


Figure 4.57: Use of function tables with *call_indirect*

The *call_indirect* instruction is also given a function type. This type is used during the validation phase, described in section 2.2.5. The validation phase ensures that the module is well-formed. The context of the *call_indirect* instruction involves checking that the function signature matches the expected function type. Thus, the value that is pushed (return value) to the stack after execution matches the instruction flow for the rest of the program.

Application When an application is compiled the necessary code for resolving the function reference and the closure environment is produced.

```
| Application(expr, args: List<Node<TypingEnv, Type>>) ->
    /// compile arguments
    let argm = List.fold (fun m arg -> m + doCodegen env arg (m.ResetAccCode()
    )) m args

    /// generate code for the expression for the function to be applied
    let exprm: Module = (doCodegen env expr m)

    // type to function signature
    let typeId = GenFuncTypeID(typeToFuncSiganture env (expandType expr.Env
    expr.Type))

    (argm)
        .ResetAccCode()
        .AddCode([ Comment "Load expression to be applied as a function" ])
        .AddCode(
            [ (CallIndirect(
                    Named(typeId),
                    [ (I32Load_(None, Some(4), exprm.GetAccCode()), "load
    closure environment pointer") ]
                    @ argm.GetAccCode() // load the rest of the arguments
                    // load function pointer
                    @ [ (I32Load(exprm.GetAccCode()), "load table index") ]
                ),
                "call function") ]
        )
```

Figure 4.58: Code generation of an application

The function's provided arguments are first evaluated, followed by the specified expression for application. A type identifier is generated, aligning with a function type declaration in the type section. Lastly, all generated code is encapsulated within the *call_indirect* instruction for consumption. Since the index and pointer to the closure environment are in memory, load instructions are inserted to retrieve the data.

Anonymous functions Anonymous functions are, by definition, not given a name by the programmer. In the AST, an anonymous function is a lambda term that is not encapsulated by a *Let* or *LetRec* node. An anonymous function is given an internal name by the compiler that is also visible in the produced WAT code. To make it easier to identify what anonymous function the name refers to, the name has been constructed to indicate the relationship between functions. The name *$fun_f/anonymous* will refer to an anonymous function declared inside the function *f*. Subsequently the innermost anonymous function of fig. 4.59 will be named *$fun_sum/anonymous/anonymous*.

```
fun sum(a: int): (int) -> (int) -> int = { // <-- Named "$fun_sum"
    fun (b: int) -> {        // <-- Named "$fun_sum/anonymous"
        fun (c: int) -> {    // <-- Named "$fun_sum/anonymous/anonymous"
            a + b + c
        }
    }
}
```

Figure 4.59: Anonymous function naming convention

Global scope and Variable promotion Let-binders in the global scope of the Hygge program, that do not contain a lambda term, become a global variable through a promotion process. The code snippet 4.60 shows a Hygge program with the variable *x* in the program's global scope that is later used inside a function *f*. *x* will be encapsulated inside the *_start* in the WAT module. Since *x* is also read inside the function *f*, *x* must be a global variable so all functions can access it.

```
let x: int = 40; // <-- Variable x becomes global value
let f: () -> int = fun() -> 40 + x; // <-- Accsess value of x

assert(f() = 80)
```

Figure 4.60: Test case 031-top-level.hyg

The function *promoteLocals* is responsible for implementing the promotion process. It takes a list of local variable names as input and generates a new module where all variables in the list are converted to global variables. Additionally, all instructions that are related to getting or setting one of the local variables in the list are converted to instructions that operate on global variables. This transformation is performed by the *localSubst* function, which recursively propagates the AST and substitutes all local get and set instructions of the specified variable. A section of the *localSubst* function is provided in code snippet 4.61.

```
let rec localSubst (code: Commented<WGF.Instr.Wasm> list) (var: string) :
    Commented<WGF.Instr.Wasm> list =
    match code with
    | [] -> code // end of code
    | (LocalGet(Named(n)), c) :: rest when n = var ->
        [ (GlobalGet(Named(n)), c + ", have been promoted") ] @ localSubst
    rest var
    | (LocalSet(Named(n), instrs), c) :: rest when n = var ->
        [ (GlobalSet(Named(n), localSubst instrs var), c + ", have been
    promoted") ]
        @ localSubst rest var
    | (LocalTee(Named(n), instrs), c) :: rest when n = var ->
        [ (GlobalSet(Named(n), localSubst instrs var), c + ", have been
    promoted")
          (GlobalGet(Named(n)), c + ", have been promoted") ]
        @ localSubst rest var
    // block instructions
    | (Block(l, vt, instrs), c) :: rest -> [ (Block(l, vt, (localSubst instrs
    var)), c) ] @ localSubst rest var
    | (Loop(l, vt, instrs), c) :: rest -> [ (Loop(l, vt, (localSubst instrs
    var)), c) ] @ localSubst rest var
    // more cases ...
```

Figure 4.61: Variable promotion

4.3.6.19 Closures

When code is generated for a lambda term, it captures and encapsulates the lexical environment within a *struct* stored in memory. From this point, this *struct* is denoted as the closure environment. The closure environment encloses all the data of the captured variables[60, section: Captured Variables and Closures] as they existed at the moment of capture.

After introducing closures some pointers have an additional segment with the address to the closure environment. Figure 4.62 illustrates the change to function pointers. Given that the pointer consistently references the index, an application can manage them uniformly. When applying a function an address to a closure environment is passed to the function, in case there does not exist a closure environment the address will be meaningless and will never be used.

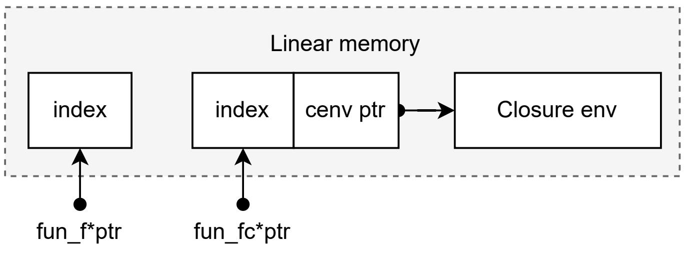


Figure 4.62: Function pointer with and without closure

To let the functions access the closure environment a closure conversion is done. Functions are rewritten by adding an argument named *cenv*, this argument points to a closure environment if one exists for the function. This alters the function signature in the WAT code. The function $(int) \rightarrow int$ becomes $(int, int) \rightarrow int$. This directly corresponds to the signature in figure 4.63. The *cenv* argument will always be the first argument in the signature, meaning that it will have a local variable index of zero. The code snippet 4.55 shows on line 12 how *cenv* is added to the function signature.

Variables inside the closure environment are resolved via *VarStorage* and the storage type *offset* is used to resolve captured variables.

```
(func $fun_h (;0;) (param $cenv i32) (param $arg_k i32) (result i32)
```

Figure 4.63: Function signature after rewrite

Closures with shared mutable variables Mutable variables are rewritten to be encapsulated inside a *struct* with one field named *value*. Code snippet 4.64 shows this rewrite of the mutable variables if captured. Moreover, all variable access in the scope is rewritten to a `FieldSelect` so it will access the field *value*.

```
let fieldName = "value"

// create struct with one field called value
let structNode =
    { node with
        Expr = Struct([ (fieldName, init) ])
        Type = TStruct([ (fieldName, init.Type) ]) }

// var node with struct pointer var
let varNode =
    { node with
        Expr = Var(name)
        Type = TStruct([ (fieldName, init.Type) ]) }

// select field value from struct every time var is used
let selectNode =
    { node with
        Expr = FieldSelect(varNode, fieldName)
        Type = init.Type }

// replace every occurrence of the var in scope with the struct FieldSelect
let scope' = ASTUtil.subst scope name selectNode

// node with sequence of let and scope
let n =
    { node with
        Expr = Let(name, tpe, structNode, scope', export) }

let m' = (doCodegen env n m)
```

Figure 4.64: Rewriting mutable variables

Before this modification, closures with mutable variables did not work in complex cases with shared variables, as shown in the code snippet 4.65. Both the *increment* and *decrement* functions capture an environment where the mutable variable *count* is shared between them. Both environments point to the *struct* with the *value* field and will, therefore, both operate on the same shared value. This is illustrated in Figure 4.66.

```
type Counters = struct {increment: () -> int; decrement: () -> int};

// Return a structure with two functions that share a counter.
// The 'count' is initialized to 0.
// The 'count' can be either incremented or decremented
fun makeCounters(): Counters = {
    let mutable count: int = 0; // The mutable variable 'count'

    // The lambda terms below capture 'count' twice
    struct { increment = fun () -> { count <- count + 1 };
             decrement = fun () -> { count <- count - 1 } } : Counters
};
// create a counter
let c1: Counters = makeCounters();
assert(c1.increment() = 1);
assert(c1.increment() = 2);
// create a counter more
let c2: Counters = makeCounters();
assert(c2.increment() = 1); // Output: 1 (independent of c1)
assert(c2.increment() = 2);
assert(c2.decrement() = 1);
assert(c2.decrement() = 0)
```

Figure 4.65: Advanced example with closures using mutable variable

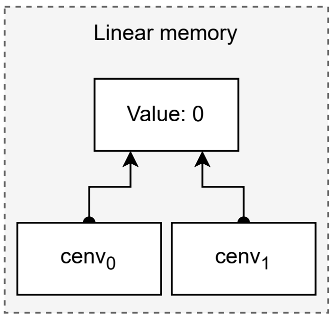


Figure 4.66: Mutable variable shared between closure environments

4.4 Summary

The main focus of this chapter is on the implementation of hyggeWasm. It describes how code was generated for each feature in the Hygge programming language. Examples of notable features implemented include composite datatypes, e.g. structs and arrays, pattern matching, functions, and closures.

Implementation of the Learning and Development tool and the WGF, is also described in this chapter. The Learning and Development tool has been implemented to load and execute binary WebAssembly modules with the extended capabilities of the hyggeWasm runtime. The runtime facilitates I/O and memory allocation by the embedder.

WAT Generation Framework (WGF) has been implemented to represent the WAT module and can convert the IR to a WAT module.

5 Bringing garbage collection to HyggeWasm

This chapter will explore how the *WasmGC* standard can bring garbage collection (GC) to *HyggeWasm*. This was added relatively late in the project period and is here treated separately from the rest of the implementation. This is not a full implementation of all features in *HyggeWasm*, but rather an exploration of how the features of *WasmGC* can be adopted by *HyggeWasm*.

5.1 Memory layout

WasmGC introduces a new way of storing *structs* and *arrays* in a *store* maintained by a module instance running within the VM[44][61]. When utilizing the *heap* memory mode of *HyggeWasm*, all dynamically allocated structures, such as *structs* and *arrays*, will reside in the *store* of the module instance. This is until the Garbage Collector (GC) deems that the structure can be removed. The static data continues to be stored in *linear memory*.

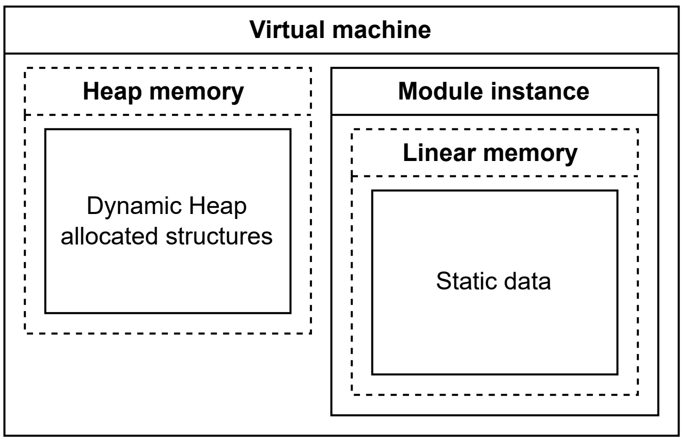


Figure 5.1: Memory split in VM

5.2 Introducing heap types

The *WasmGC* proposal enables garbage collection (GC) through the definition of *struct* and *array* types.

For the GC to reclaim memory chunks, it needs information about the structure of the elements stored in memory. This includes details about the data type within the structures and references to other objects residing in memory.

Types are added to the type section alongside function types. They need to be defined before use. Types can refer to other types as long as they are defined before use within the module. Thus, the types have the same order as they are defined in the Hygge program.

5.3 New reference instructions

To enable working with new data objects on the *heap*, a set of new *reference instructions* have been added to Wasm. The new instructions facilitate instances, reading and writing data, and casting types. A list of the instructions used by *HyggeWasm* can be seen in table 5.1.

Instruction	Description
`struct.new`	Create a new struct
`struct.get`	Access a field in a struct
`struct.set`	Set a field in a struct
`array.new`	Create a new array
`array.set`	Set a value in an array
`array.get`	Get a value from an array
`array.len`	Retrieve the length of an array
`ref.cast`	Cast a reference

Table 5.1: WasmGC Reference Instructions

5.4 Code generation

This section ensures the report's overall brevity and maintains focus on essential content. Hence, not every implemented feature supporting the *heap* memory mode will be comprehensively covered. Instead, it provides examples of the produced code and general code generation examples.

5.4.1 Arrays

Arrays will be examined by exploring the compiled code of the Hygge program in code snippet 5.2, compiled with the *heap* memory mode. The program initializes an array, retrieves its length, reads a value, modifies an element, and reads it again.

```
let arr: array {int} = array(2 + 2, 40 + 2); // create array
let len: int = arrayLength(arr); // get the length
let val: int = arrayElem(arr, 1); // get value at index 1

assert(len = 4); // asset length
assert(val = 42); // assert initial value of index 1

arrayElem(arr, 1) <- 200; // write value to index 1

let val2: int = arrayElem(arr, 1); // get value at index 1

assert(val2 = 200) // assert new value of index 1
```

Figure 5.2: Working with arrays in Hygge

The type section now contains the heap type definitions. The type *$arr_i32* defines a mutable `i32` array corresponding to the array created in the Hygge program. Type names have been designed for reuse. Moreover, the global variable to which the array is assigned has changed, transforming it from an `i32` value to a nullable reference of the type *$arr_i32*. The type declarations can be seen in 5.3. Note that the type section needs to be before the global section for the type reference to work.

```
(type $arr_i32 (;0;) (array (mut i32))) ;; mutable i32 array type
;; global variable declaration with nullable reference to the type arr_i32
;; is initialized to null
(global $var_arr (;2;) (mut (ref null $arr_i32)) (ref.null $arr_i32))
```

Figure 5.3: Type and global declarations

The code sample 5.4 shows selected parts of the compiled program that have changed due to the new instructions. The code snippet shows how the instructions related to arrays are used within the compiled program. Using the new *reference instructions* reduces the number of instructions needed. As opposed to the implamentaion in section 4.3.6.14, the index bounds are automatically checked to ensure they are within bounds when using heap types. Furthermore, the debugging experience is improved with appropriate error messages in cases of out-of-bounds access.

```
;; Create new Array
(array.new $arr_i32 ;; create new array
  (i32.add ;; add (40 + 2) <-- the array is inilized with 42 in all positions
    (i32.const 40) ;; push 40 on stack
    (i32.const 2) ;; push 2 on stack
  )
  (i32.add ;; add (2 + 2) <-- the size of the array will be 4
    (i32.const 2) ;; push 2 on stack
    (i32.const 2) ;; push 2 on stack
  )
)

;; Get length of Array
(array.len ;; get length of array
  (global.get $var_arr) ;; get the reference of 'arr'
)

;; Write value to array
(array.set $arr_i32 ;; write 200 to index 1 in array 'arr'
  (global.get $var_arr) ;; get the reference of 'arr'
  (i32.const 1) ;; write at index 1
  (i32.const 200) ;; write the value 200
)

;; Access array element
(array.get $arr_i32 ;; get data from array 'arr' at index 1
  (global.get $var_arr) ;; get reference of 'arr'
  (i32.const 1) ;; read at index 1
)
```

Figure 5.4: Use of the array reference instructions

All array operations are implemented in the *heap* memory mode, except for the *slice* functionality.

5.4.2 Structs

Structs will be examined by exploring the compiled code of the Hygge program in 5.5, compiled with the *Heap* memory mode. The program creates a *struct* with three fields, modifies one field, and asserts the content of the *struct*.

```
let s: struct {i: int; a: float; b: int} = struct {i = 42; a = 93.2f; b = 90};
s.b <- 100;
assert(s.b = 100);
assert(s.i = 42);
assert(s.a = 93.2f)
```

Figure 5.5: Use of structs

A heap type matching the *struct* is placed in the module. See code snippet 5.6. The *struct* type name reflects the fields and types of the *struct* itself. This is because fields are referenced by name in contrast to the other memory modes where they are referenced by index. This means the field names must be part of the type identifier to be unique. The global variable *var_s* uses the type and initializes the variable to a null reference of the same type.

```
(type $s|i-i32|a-f32|b-i32 (;0;) (struct
        (field $i (mut i32))
        (field $a (mut f32))
        (field $b (mut i32))))

(global $var_s (;2;) (mut
        (ref null $s|i-i32|a-f32|b-i32))
        (ref.null $s|i-i32|a-f32|b-i32))
```

Figure 5.6: Type definitions of structs

The code sample 5.4 shows selected parts of the compiled program that use the new *struct* reference instructions.

```
;; Create struct
(struct.new $s|i-i32|a-f32|b-i32
  (i32.const 42) ;; push 42 on stack
  (f32.const 93.199997) ;; push 93.199997 on stack
  (i32.const 90) ;; push 90 on stack
)
;; Set value of field 'b'
(struct.set $s|i-i32|a-f32|b-i32 $b ;; set field: b
  (global.get $var_s) ;; get local var: var_s, have been promoted
  (i32.const 100) ;; push 100 on stack
)
;; Get value of field 'b'
(struct.get $s|i-i32|a-f32|b-i32 $b ;; get field: b
  (global.get $var_s) ;; get local var: var_s, have been promoted
)
```

Figure 5.7: Use of the struct reference instructions

5.5 Example of closure

The Hygge program shown in snippet 5.8 compiles correctly with the new heap constructs. The anonymous inner function of *makeCounter* captures *x* and *y*.

```
fun makeCounter(y: int): () -> int = {
    let mutable x: int = 2;
    fun () -> {
        x <- x * y // x is captured from the surrounding scope
    }
};
let c1: () -> int = makeCounter(2);
assert(c1() = 4)
```

Figure 5.8: Closure example

```
(func $fun_makeCounter/anonymous (param $cenv (ref null eq)) (result i32)
 ;; local variables declaration:
 (local $clos (ref $clos_fun_makeCounter/anonymous))
 ;; downcast to the closure env type of 'fun_makeCounter/anonymous'
 (local.set $clos
   (ref.cast (ref $clos_fun_makeCounter/anonymous)
     (local.get 0) ;; get cenv
   )
 )
 ;; local var 'clos' is used to access the closure env from here.
)
```

Figure 5.9: Cast of closure environments

The environments are parsed around to enable the closures as the *eq* type. The abstract type, *eq*, is a subtype of *any* that encompasses all types, including references. Then, before use, it is cast to the appropriate type. This can be seen in code sample 5.9. The closure type is created when compiling the lambda term.

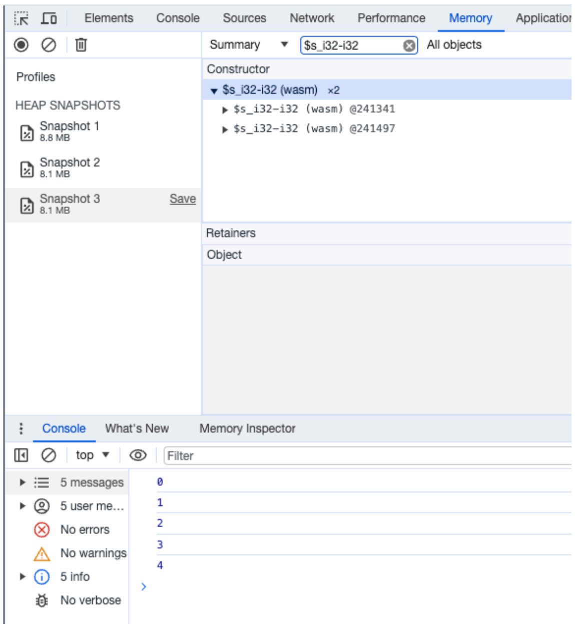
5.6 Experimenting with the garbage collector

A test was conducted to evaluate whether the structures placed on the heap were correctly garbage collected. The program shown in code example 5.10 from */tests/codegen/pass/mem-loop.hyg* places a new struct on the heap for each iteration. The structure is only referenced inside the loop body and can be collected right after each iteration. Furthermore, the contents of the newly allocated struct were checked as expected.

```
let mutable x: int = 0;
let stop: int = 100000;

while (x < stop) do {
    let s1: struct {f: int; i: int} = struct {f = 42 + x; i = 42};
    assert(s1.i = 42);
    assert(s1.f = 42 + x);
    print(x);
    x++
}
```

Figure 5.10: Test case for WasmGC heap structures

For garbage collection to work correctly, structures placed on the heap must be collected to free up memory during runtime. To inspect this, the Google Chrome memory profiling feature is used. This is part of Chrome's development tools. The test program 5.10 is executed with a breakpoint inside the loop body. After five iterations, memory profiling is used to create a heap snapshot. When expecting the heap snapshot, it can be seen that only two of the structures created are still in memory, proving that the garbage collector is working as intended. A screenshot of this can be seen in fig. 5.11.



```
▾ $s_i32-i32 (wasm)  ×2
  ▸ $s_i32-i32 (wasm) @241341
  ▸ $s_i32-i32 (wasm) @241497
```

```
0
1
2
3
4
```

Figure 5.11: Function pointer with and without closure

5.7 Limitations of the heap memory mode implementation

Programs compiled in the *heap* memory mode have some serious issues that have not been resolved in the implementation. The following conditions will either lead to compiler time errors or to compiled programs that do not run correctly.

Programs that use subtyping sometimes produce inconsistent identifier labels, resulting in a faulty program. Structural subtyping does not work since the needed upcasts are not included in the code generation. This means a program where the type and the actual struct completely match, running correctly. See code example 5.5 for a program that runs correctly. In contrast, a program like the one shown in 5.12 will fail since the type declaration identifier will match $f : int$ and therefore not match $f : int, g : bool$.

```
let s: struct {f: int} = struct {f = 1 + 2; g = true};
assert(s.f = 3)
```

Figure 5.12: Program using structural subtyping

The discriminated union-type constructor and pattern matching do not work because they were not implemented to take advantage of WasmGC constructions yet. Array slices have not been implemented; using array slices will produce programs that either cannot run or will run incorrectly. The *heap* support simple closures but will fail in cases where multiple functions access the same mutable variable due to the missing types.

5.8 Capabilities Overview

Even though the implementation of the *heap* mode has limitations, the limited part does provide enough to implement classic algorithms like bubble sort, binary search, merge sort, and selection sort. Furthermore, closures work in most cases.

Below is a selection of noteworthy test cases showcasing the capabilities of the *heap* mode.

- `/tests/codegen/pass/bubblesort.hyg`
- `/tests/codegen/pass/binarySearch.hyg`
- `/tests/codegen/pass/mergeSort.hyg`
- `/tests/codegen/pass/selectionSort.hyg`
- `/tests/codegen/pass/000-lambda-3.hyg`
- `/tests/codegen/pass/000-lambda-8.hyg`
- `/tests/codegen/pass/000-lambda-7.hyg`

5.9 Summary

This chapter introduces the new memory architecture using the store of the VM. A new set of heap types is presented to inform the VM of the data structures on the heap. These structures can then be manipulated with the reference instructions provided by *WasmGC*. Then, examples of how the new heap types and instructions are generated to provide the language features of Hygge. Experiments are conducted to ensure correct garbage collection operation, and the implementation's limitations are described.

6 Optimizations

This chapter describes how optimization techniques have been utilized to produce more effective WebAssembly code. The optimizations have been implemented as late-stage optimizations, known as peephole optimizations[1, p. 349]. Each optimization targets a specific pattern of instructions that can be substituted or removed. These optimizations may run multiple parses over the instructions and will continue until a stable output is reached.

6.1 Optimizing the reading and writing of local variables

A prevalent practice in handling local variables involves writing to local variables, followed promptly by their subsequent retrieval. This is because local variables often are used as *virtual registers*, acting as simple temporary variables. When generating code, in the initial phase, the ordering of the instructions cannot always be anticipated due to an absent context. This can lead to the *local.set* and *local.get* being used adjacently to each other. This is a common pattern. Thus, WebAssembly has included the *local.tee* instruction to optimize this pattern. The *local.tee* instruction is equivalent to a *local.set* followed by a *local.get*.

To take advantage of this, the optimization algorithm searches for the pattern of two adjacent *local.set* and *local.get* with the same label. When found, the two instructions are substituted with a single *local.tee*. This code that implements this optimization can be seen in code snippet 6.1.

```
| (LocalSet(l1, instrs), c1) :: (LocalGet(l2), c2) :: rest when l1 = l2 ->
    (LocalTee(l1, instrs), c1 + c2) :: optimizeInstr rest
```

Figure 6.1: Substituting set and get for a tee

6.2 Dead-code Elimination

Dead-code elimination strives to remove code that does not influence the program's result. The tree structure that the WAT instructions form is traversed, searching for a drop-node. An example of this tree structure's appearance can be seen in code snippet 6.2. The goal of this optimization is to prune the branches that have a drop node as its root.

A drop-node represents the WebAssembly instruction `drop` and encapsulates the instructions that push an unused value to the *operand stack*. The drop instructions are inserted during the initial code generation process to accommodate WebAssembly's validation rules as described in section 4.3.5.

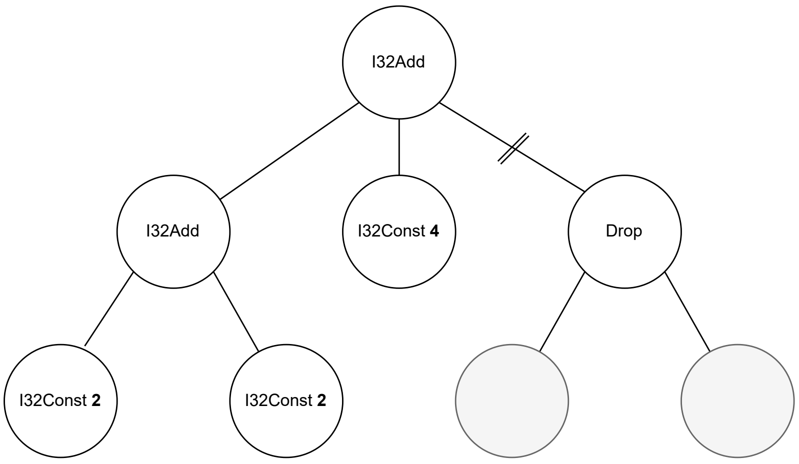


Figure 6.2: Drop subtree

Before pruning the branch with the drop-node as the root, we have to ensure that no instructions that can cause side effects are present in the sub-tree. All instructions that write to variables or memory are considered to cause side effects[7][p. 247]. If the sub-tree is found to include at least one side effect, the tree is not altered.

```
(drop
  local.tee $var
)
```

Figure 6.3: Example of a sub-tree with a side effect

One example of a code segment that would cause this problem can be seen in 6.3. In this case, it is the `local.tee` instruction that writes to a variable. This variable is defined outside the sub-tree and may be used elsewhere. Therefore, the sub-tree cannot be pruned without altering the program's intended execution.

6.3 Special case for dead-code elimination

The specific scenario shown in code snippet 6.3 has its own optimization. The code pattern of a `drop` with a `local.tee` as the last instruction inside that `drop` can be rewritten to a `local.set` without the `drop`, thus bringing down the instruction count.

```
fun f1(x: int): int = x--
```

Figure 6.4: Hygge code that can be optimized

A concrete example is the *Hygge* code snippet in 6.4. Compiling this without optimizations will produce the output seen in code snippet 6.5a. Correspondingly, the code snippet in 6.5b is the compiled output with this optimization applied.

```
(func $fun_f1 (param $cenv i32) (
    param $arg_x$1 i32) (result i32
    )
    (local.get $arg_x$1)
    (drop
      (local.tee $arg_x$1
        (i32.sub
          (local.get $arg_x$1)
          (i32.const 1)
        )
      )
    )
)
```

(a) Before optimisation

```
(func $fun_f1 (param $cenv i32) (
    param $arg_x$1 i32) (result i32
    )
(local.get $arg_x$1)
(local.set $arg_x$1
  (i32.sub
    (local.get $arg_x$1)
    (i32.const 1)
  )
)
)
```

(b) After optimisation

Figure 6.5: Before and after optimization

6.4 Constant folding

Constant folding involves identifying and evaluating constant expressions during the compilation stage instead of their computation at runtime. Constant folding has been implemented on integer and floating point arithmetic operations and relational operations. In code example 6.6, the case that handles folding of the *i32.add* instruction is seen. Note that every arithmetic and relational operation has a similar case.

This case matches when a *i32.add* instruction that contains two constants as its arguments are present. The function *isConstConst* only returns true if and only if two *i32.const* or two *f32.const* are present in *instrs*. Since *i32.add* is an integer instruction, it can safely be assumed to hold integers.

```
| (I32Add(instrs), c1) :: rest when isConstConst instrs ->
    let (v1, v2) = (getI32ConstConst instrs)
    (I32Const(v1 + v2), c1) :: optimizeInstr rest
```

Figure 6.6: Constant folding of i32.add node with two constants

The function *getI32ConstConst* retrieves the two integer values of the two *i32.const* instructions. The *i32.add* instruction is substituted with a single *i32.const* that holds the sum of the two values. Thereby, the optimizer calculates the result statically so it does not have to be done during the runtime of the program.

Considering the example 6.7, compiling this code without optimizations yields the output shown in 6.8.

```
assert(4 + 5 + 3 = 12)
```

Figure 6.7: Hygge program to be optimized

In line 7 of code snippet 6.8 is an *i32.add* instruction that contains two constant values, which is the exact case this optimization targets. Applying the rule once yields the result of `i32.const 9`. This means that the outer *i32.add* starting on line 8 contains two constants, and the rule can be applied again. As with any peephole optimization, they are applied until a stable output is reached, meaning that the optimizations can be applied multiple times to reduce the needed instructions.

```
1  (func $_start (;0;)
2  ;; execution start here:
3  (if
4      (i32.eqz ;; invert assertion
5        (i32.eq ;; equality check
6          (i32.add
7            (i32.add ;; <-- become 'i32.const 9'
8              (i32.const 4) ;; push 4 on stack
9              (i32.const 5) ;; push 5 on stack
10           )
11           (i32.const 3) ;; push 3 on stack
12         )
13         (i32.const 12) ;; push 12 on stack
14       )
15     )
16   (then
17     (global.set $exit_code ;; set exit code
18       (i32.const 42) ;; error exit code push to stack
19     )
20     (unreachable) ;; exit program
21   )
22 )
23 ;; if execution reaches here, the program is successful
24 )
```

Figure 6.8: Before constant fold

In this case, the constant folding of the instructions *i32.add*, *i32.eq*, and *i32.eqz* can be used to reduce the expressions to a constant value. The result of the of these reductions can be seen in 6.9.

```
(func $_start (;0;)
;; execution start here:
(if
  (i32.const 0) ;; condition
  (then
    (global.set $exit_code ;; set exit code
      (i32.const 42) ;; error exit code push to stack
    )
    (unreachable) ;; exit program
  )
)
;; if execution reaches here, the program is successful
)
```

Figure 6.9: After constant fold

6.5 Branch-level tree shaking

The code shown in 6.9 can be reduced even further since the condition of the *if* instruction is now a constant value. This means that we can statically determine, whether the *if* or the *else* block will be executed. In this case, there is no *else* block, and the condition is a false value; therefore, the entire *if* instruction can be removed, resulting in an empty module. The programming language *AssemblyScript* employs a comparable technique, denoted as *Branch-level tree shaking*[62].

6.6 Summary

This chapter describes how optimization techniques can be utilized to make the generated code more effective, four techniques are laid out; namely local variable read-and-write optimization, dead-code elimination, constant folding, and branch-level tree shaking, showing examples of how they affect the code and reduce the instruction count.

7 Evaluation

This section assesses the generated code, the impact of the optimization phase, assembler support, and the testing site of the compiler.

7.1 Internal vs external memory mode

As described in Section 3.3.4, programs compiled with the *Internal* memory mode include more logic, producing larger modules. This section will try to quantify the size difference.

Consider the test program from the test suite *000-lambda-10-shadow.hyg*. When compiling the program with the *folded* writing style and the memory mode *external*, the WAT file consists of *403* lines. When we then only change the memory mode to *internal*, the file consists of *634* lines instead, an increase of 57.32%. When both compilations are performed in the *linear* writing style, this percentage becomes 50%.

The difference in the size of the WebAssembly modules will depend on the language features used. Due to this variability, no further effort has been made to investigate it. However, it can be inferred that the program's extensive utilization of dynamically allocated data structures will result in considerably larger WebAssembly modules.

7.2 Assembler Support

Three different assemblers have been tested on the code produced by *HyggeWasm*.

In the memory modes *internal* and *external*, the *Wat2Wasm*[43] and *wasm-tools*[63] tools can produce valid functional binary modules for all test programs provided by the project, and do so in both writing styles.

Wat2Wasm does not support *WasmGC* constructs at the time of writing and can, therefore, not be used for this mode. In this case, *wasm-tools* should be used.

wasm-as[18] can assemble *64* of the *186* programs corresponding to 34.41% of the test suite. Only the folded writing style can be processed by *wasm-as*.

7.3 Testing

The section presents a comprehensive analysis of the testing procedures conducted for this thesis. The test suite consists of `482` test programs with varying complexity. The majority of test programs test very specific scenarios related to a feature.

All tests are in the */tests* folder. Tests are separated into folders based on the phase that they are meant to test. The folders are:

- codegen
- lexer
- parser
- typechecker

Please note that the tests for the lexer, parser, and type checker are untouched and only included for completeness.

The original test suite for *HyggeC*, the Hygge compiler that produced *RISC-V* assembly consisted of 350 test programs, where some of the tests were supplied at the start of the Compiler Construction course, and others were added as part of the assignments throughout the course. Most tests have been reused and are part of the new test suite of 491 test programs.

The tests that this project is primarily concerned with are the test programs that target code generation. There are 212 test programs in this category, found in the folder */tests/codegen*. The rest of this section will only deal with tests for code generation. To give an impression of the size and scope of the code generation tests, see figure 7.1, which shows a histogram of the number of instructions in the compiled test programs in increments of 50. It can be observed that most tests in the test suite are fairly small, with under 100 instructions.

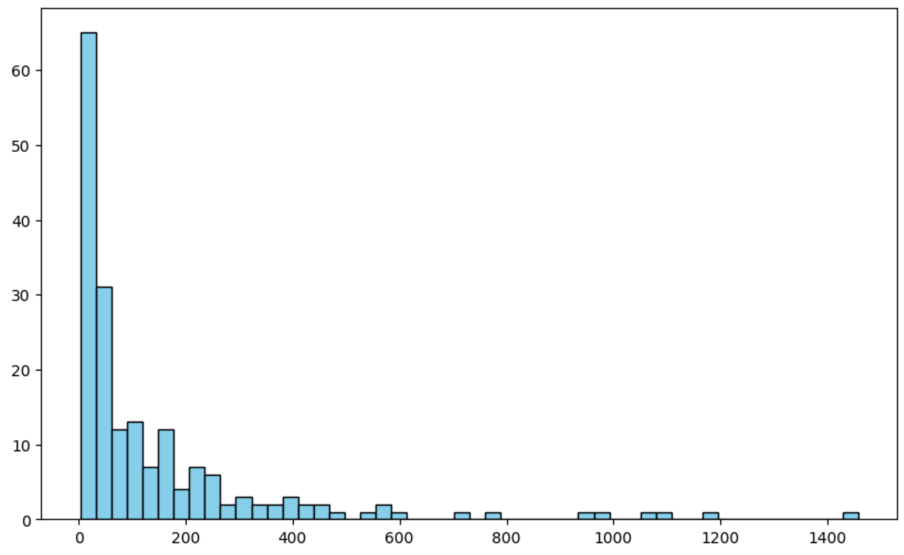


Figure 7.1: Histogram of instructions count

To ensure that all the different modes work correctly, all tests targeting the code generation are run in multiple configurations; this includes memory modes, *internal* and *external* as well as *writing style*, *folded* and *linear*. All test programs are conducted with and without peephole optimizations applied to ensure that the program execution is not altered. All the configurations mean that when the entire test suite is executed, **1.203** distinct tests are evaluated to ensure correct operation. Code generation tests constitute 1040 of these tests. The *heap* mode is not part of these tests and has been tested manually through the Learning and Development tool.

7.4 Optimizations

This section will assess the peephole optimizations that have been incorporated. The methodology for evaluating these optimizations involves quantifying the number of instructions within each function in the module. This quantification is conducted before the peephole optimizations are applied and after their application, allowing for a comparative analysis of the two.

All data is collected into one set in appendix H. A sample of the data is shown in 7.1. The data consists of the file name of the test program, the instruction count before and after optimizations are applied, the difference between the two, and a reduction in percentage.

name of file	instr count	instr count after op	diff	% reduction
000-negate-float	16	0	16	100.00
hygge-union-structs	210	206	4	1.90
hygge-union	105	104	1	0.95
if-return	774	468	306	39.53
insertionSort	936	732	204	21.79
letClousure	244	237	7	2.87
linearSearch	536	423	113	21.08
memory-grow	142	142	0	0.00
memory-static	3	3	0	0.00
mergeSort	1458	1247	211	14.47
multiplier	82	68	14	17.07
quickSort	1094	889	205	18.74
selectionSort	977	770	207	21.19

Table 7.1: Exsample of optimization data set

The analysis has been done in *Python* and can be found in the file *DataAnalysis/notebook.ipynb*.

7.4.1 The impact of optimization

The mean of the percentage reduction of the instruction count for the test suit is 14.62%. 109 of the test programs have no reduction. 18 test programs have a 100% reduction.

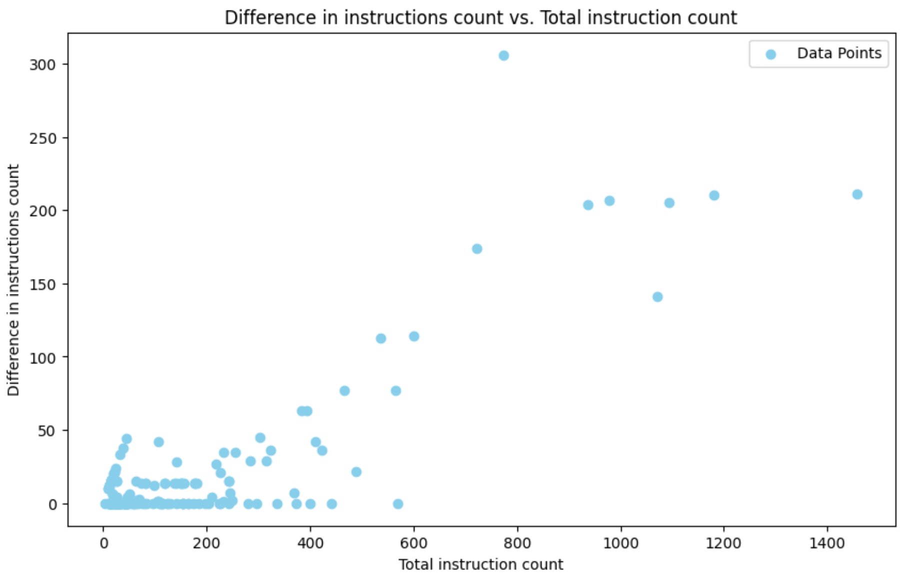


Figure 7.2: Difference in instructions count vs. Total instruction count

Figure 7.2 illustrates a somewhat noticeable trend indicating a linear relationship between the module size and the corresponding reduction in the number of instructions.

When applied separately, the dead-code elimination reduces the instruction count mean by 2.32%.

When applying only the constant folding technique, the mean of the reduction is 12.36%. This means constant folding is the most effective technique used. If this is applied without the branch-level tree shaking, the mean drops to 5.06%.

When applied separately from the other techniques, the local read-and-write optimization results in a mean reduction of only 0.05%. Thus, the least effective optimization that has been implemented.

The mean of 14.62% reduction in instruction count across all tests, when every optimization is applied is a satisfying result and shows that the techniques used can be successfully applied to WebAssembly.

7.5 Summary

This chapter starts by evaluating the impact of internal and external memory modes in terms of program size. Then, the support of assemblers was evaluated, concluding that *Wat2Wasm* and *wasm-tools* work on the entire test suite. Then, the test suite itself is and its scope evaluated. This leads to using the test suite to evaluate the impact of the optimizations described in chapter 6.

8 Future work

This chapter will address missing features, general improvements, and additional optimizations of the compiler.

Improve dead code elimination The current implementation of dead code elimination, as described in Section 6.2, does not account for the fact that temporary variables may only be needed in the sub-tree that is removed, leaving their declarations behind in either the local section of a function or the global section of the module. Therefore, it would be natural to extend this function to check if a variable is used elsewhere; otherwise, remove the declaration, thereby producing a cleaner module.

Allow programs to have zero memory All modules produced by *HyggeWasm* have at least one page of memory available as described in Section 3.3.4.2. For programs that do not have static or dynamic data, this is unnecessary and should be avoided. A solution could be to check if static memory is utilized and then examine whether any memory instructions are used within the module to ensure no dynamic allocation is needed. If none are found, then it is acceptable to set the memory to zero pages.

Reintroduce exported functions Before closures were introduced into Hygge, the feature of the exporting functions with the *export* keyword was implemented. To allow this to work correctly with closures a calling convention has to be established.

Full support for WasmGC A natural extension of the project would be to continue the implementation of *Heap* memory mode that relies on the *WasmGC* constructs. The Limitations of the current implementation are described in Section 5.7.

Explicitly defined main function Add the possibility for the Hygge programmer to define an explicit entry point as described in Section 3.3.7. This would also mean that no code can exist outside a function, in the global scope.

Enhance support of *wasm-as* assembler Several things can be done to enhance the support of the *wasm-as*. Most of the errors encountered when passing the WebAssembly programs produced by *HyggeWasm* were related to not following the strict structure of the tree that *wasm-as* enforces. This means that an instruction like *i32.add* must only have two child nodes and can not have a three node even though it does not amount to a value.

One way to do this is to change the IR to enforce this structure. Code sample 8.1 shows how this could be done.

```
| I32Add of Wasm Commented list ;; current implementation
| I32Add of Wasm Commented * Wasm Commented ;; new implamentation
```

Figure 8.1: Enforce structure in IR

Another change will be that it seems like *wasm-as* expect result type declarations in some places that do not match what *wat2wasm* do. Therefore there will have to be added a new compile flag that speedily added when in ”*wasm-as*-mode.”

Combine memory allocator into one WebAssembly module to run the same code in all environments At the moment, the memory allocator is implemented in both *TypeScript* and *C#*, leaning to less maintainable code. A better way would be to compile the memory allocator to WebAssembly and run that one version.

Provide better debugging experience One way to provide a better debug experience would be to produce *Source maps* as part of the compilation process to map between the original Hygge program and the produced WebAssembly code. This would require producing the binary format instead of WAT.

Tail recursion optimization Even though a Hygge function is written to be tail-recursive, the generated WebAssembly code does not discard the caller frames. This can lead to a call stack overflow when doing heavy recursion. To mitigate this problem, the tail call extension[64] of WebAssembly can be used. For this to work in HyggeWasm, there needs to be a way to recognize when a function is tail-recursive and then substitute the `call_indirect` instruction for the `return_call_indirect` instruction.

9 Conclusion

The main purpose of the thesis has been to design and implement a WebAssembly compiler back-end for the existing programming language *Hygge*.

WebAssembly was thoroughly reviewed to understand how the ISA could translate to the high-level features of Hygge. Furthermore, the method used to compile each feature was assessed for reliability and validity by reviewing other compilers and literature.

The *Hygge* high-level features outlined in appendix A.1 have been successfully implemented. Extensive testing has been carried out to validate and ensure correct operation, thus fulfilling the problem **P1** in the Problem statement.

While *HyggeWasm* cannot be compared with mature languages like *Rust*, it does bring non-trivial features like closures. A feature that not even an established WebAssembly language like *AssemblyScript* does support[65] at the time of writing. Furthermore, *HyggeWasm* incorporates the new features of *WasmGC* that are at the absolute forefront of the WebAssembly features. It is one of the few languages using these new features at the moment.

This thesis explores different solutions to design problems, including memory modes, system interfaces, and writing styles of the textual format, showcasing a wide range of problem-solving approaches. Thereby fulfilling **P2** in the Problem statement.

As the project progressed, it became clear that additional utilities were needed. The Learning and Development tool was necessary to ensure easy execution and debugging of the WebAssembly programs. In addition, the HyggeWasm runtime was implemented in both *C#* and *TypeScript*. This allows the produced WebAssembly modules to easily interact with I/O and entrust memory allocation to the host system.

The memory modes allow Hygge to manage and allocate memory with three distinct techniques. The Hygge system interface (HyggeSI) brings a simplified interface that nicely integrates with the Learning and Development tool. WASI can be used to produce universal binaries with limited capability.

The WebAssembly code was also optimized using various techniques to improve the resulting programs. Moreover, the optimizations were evaluated to form a consensus on their effectiveness to fulfill **P3** in the Problem statement.

Incorporating a new IR into the project required significant effort. However, this step was instrumental in facilitating the comments for WAT programs and offered the much-needed flexibility to exploit new *WasmGC* instructions.

For the reason described in this section, it can be concluded that the project has been successful in designing and implementing the new compiler back-end. One may, however, consider further development of *HyggeWasm*, on the basis of thoughts found in Chapter 8.

Bibliography

[1] Henri E Bal Ceriel JH Jacobs Koen Langendoen Dick Grune Kees van Reeuwijk. *Modern Compiler Design*. Springer, 2012. ISBN: 9781461446996.

[2] *RISC-V*. Online; accessed 15 of November 2023. Nov. 2023. URL: https://riscv.org/.

[3] *A Comparison between WebAssembly and RISC-V*. Online; accessed 9 of Januar 2024. Jan. 2024. URL: https://medium.com/@losfair/a-comparison-between-webassembly-and-risc-v-e8fb9d37e6cc.

[4] *WebAssembly (Wasm) and Its Impact: How Wasm is Changing the Landscape of Web Development*. Online; accessed 9 of Januar 2024. Jan. 2024. URL: https://www.linkedin.com/pulse/webassembly-wasm-its-impact-how-changing-landscape-web-development/?trk=public_post_main-feed-card_feed-article-content.

[5] *The Future of Web Development: Exploring WebAssembly and Its Revolutionary Impact*. Online; accessed 9 of Januar 2024. Jan. 2024. URL: https://medium.com/@rohitsharma_49878/the-future-of-web-development-exploring-webassembly-and-its-revolutionary-impact-f42d042279a5.

[6] *What is fsharp*. Online; accessed 15 of November 2023. Nov. 2023. URL: https://learn.microsoft.com/en-us/dotnet/fsharp/what-is-fsharp.

[7] Peter Sestoft. *Programming Language Concepts - Second Edition*. Springer, 2017. ISBN: 9783319607894.

[8] Alceste Scalas. *Module 2: The Hygge0 Language Specification*. Online; accessed 16 of August 2023. DTU, Aug. 2023. URL: https://courses.compute.dtu.dk/02247/f23/hygge0-spec.html.

[9] *Type Compatibility*. Online; accessed 14 of November 2023. Nov. 2023. URL: https://www.typescriptlang.org/docs/handbook/type-compatibility.html.

[10] Gerard Gallant. *The State of WebAssembly – 2022 and 2023*. Online; accessed 4th of September 2023. Uno platform, Jan. 2023. URL: https://platform.uno/blog/the-state-of-webassembly-2022-and-2023/.

[11] Andreas Haas et al. "Bringing the web up to speed with WebAssembly". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2017). URL: https://api.semanticscholar.org/CorpusID:5899227.

[12] webassembly.org. *Roadmap*. Online; accessed 14 of November 2023. webassembly.org, Oct. 2023. URL: https://webassembly.org/roadmap/.

[13] WebAssembly Community Group. *WebAssembly Specification*. Online; accessed 10 of August 2023. WebAssembly Community Group, Aug. 2023. URL: https://webassembly.github.io/spec/core/index.html.

[14] *WebAssembly*. Online; accessed 14 of November 2023. Nov. 2023. URL: https://developer.mozilla.org/en-US/docs/WebAssembly.

[15] webassembly.org. *WebAssembly High-Level Goals*. Online; accessed 13 of October 2023. webassembly.org, Oct. 2023. URL: https://webassembly.org/docs/high-level-goals/.

[16] webassembly.org. *Use Cases*. Online; accessed 13 of October 2023. webassembly.org, Oct. 2023. URL: https://webassembly.org/docs/use-cases/.

[17] *About Emscripten*. Online; accessed 5 of Januar 2024. Jan. 2024. URL: https://emscripten.org/docs/introducing_emscripten/about_emscripten.html.

[18] *binaryen toolschain*. Online; accessed 15 of Januar 2024. Jan. 2024. URL: https://github.com/WebAssembly/binaryen.

[19] *Grain: A strongly-typed functional programming language for the modern web.* Online; accessed 4 of February 2024. Feb. 2024. URL: https://grain-lang.org/.

[20] The AssemblyScript Authors. *AssemblyScript*. Online; accessed 17 of October 2023. AssemblyScript.org, Oct. 2023. URL: https://www.assemblyscript.org/.

[21] W3C. *WebAssembly specification - Types*. Online; accessed 2 of Oct 2023. W3C, Aug. 2023. URL: https://webassembly.github.io/spec/core/bikeshed/#types.

[22] *Overview*. Online; accessed 4 of February 2024. Feb. 2024. URL: https://webassembly.github.io/spec/core/intro/overview.html?highlight=linear+memory.

[23] *WebAssembly.Memory() constructor*. Online; accessed 14 of November 2023. Nov. 2023. URL: https://developer.mozilla.org/en-US/docs/WebAssembly/JavaScript_interface/Memory/Memory.

[24] Mozilla foundation. *Understanding WebAssembly text format*. Online; accessed 10 of August 2023. Mozilla foundation, Aug. 2023. URL: https://developer.mozilla.org/en-US/docs/WebAssembly/Understanding_the_text_format.

[25] Daniel Holden. *S-Expressions*. Online; accessed 10 of August 2023. Build Your Own Lisp, Aug. 2023. URL: https://buildyourownlisp.com/chapter9_s_expressions.

[26] WebAssembly Community Group. *Instructions*. Online; accessed 11 of August 2023. WebAssembly Community Group, Aug. 2023. URL: https://webassembly.github.io/spec/core/text/instructions.html.

[27] Rick Battagline. *THE ART OF WEBASSEMBLY*. William Pollock, 2021. ISBN: 9781718501447.

[28] *Wasm ABIs*. Online; accessed 13 of December 2023. Dec. 2023. URL: https://www.webassembly.guide/webassembly-guide/webassembly/wasm-abis.

[29] *WebAssembly System Interface*. Online; accessed 18 of Januar 2024. Jan. 2024. URL: https://github.com/WebAssembly/WASI/blob/main/README.md.

[30] Shangtong Cao Yixuan Zhang and et al. Haoyu Wang. “Characterizing and Detecting WebAssembly Runtime Bugs”. In: (2023).

[31] *WasmerSharp*. Online; accessed 15 of November 2023. Nov. 2023. URL: https://migueldeicaza.github.io/WasmerSharp/.

[32] *WasmerSharp*. Online; accessed 15 of November 2023. Nov. 2023. URL: https://www.nuget.org/packages/WasmerSharp/.

[33] *Wasmtime*. Online; accessed 15 of November 2023. Nov. 2023. URL: https://www.nuget.org/packages/Wasmtime.

[34] *Four Approaches to Debugging Server-side WebAssembly*. Online; accessed 1 of Januar 2024. Jan. 2024. URL: https://shopify.engineering/debugging-server-side-webassembly.

[35] *WebAssembly Garbage Collection (WasmGC) now enabled by default in Chrome*. Online; accessed 27 of Januar 2024. Jan. 2024. URL: https://developer.chrome.com/blog/wasmgc.

[36] *Apple silicon*. Online; accessed 4 of February 2024. Feb. 2024. URL: https://en.wikipedia.org/wiki/Apple_silicon.

[37] *The lldb TUI (text user interface)*. Online; accessed 1 of Januar 2024. Jan. 2024. URL: https://peeterjoot.com/2019/08/26/the-lldb-tui-text-user-interface/.

[38] *Python Scripting*. Online; accessed 15 of Januar 2024. Jan. 2024. URL: https://lldb.llvm.org/use/python.html.

[39] *WebAssembly and Dynamic Memory*. Online; accessed 30 of Januar 2024. Jan. 2024. URL: https://frehberg.wordpress.com/webassembly-and-dynamic-memory/.

[40] *binaryen.js*. Online; accessed 7 of Januar 2024. Jan. 2024. URL: https://github.com/AssemblyScript/binaryen.js.

[41] *Binaryen.js - Creating WebAssembly from JavaScript*. Online; accessed 7 of Januar 2024. Jan. 2024. URL: https://kripken.github.io/slides/binaryen.js.html#/.

[42] *binaryen-net*. Online; accessed 7 of Januar 2024. Jan. 2024. URL: https://github.com/erandis-vol/binaryen-net.

[43] W3C Working Group. *wabt*. Online; accessed 9 of August 2023. W3C Working Group, Aug. 2023. URL: https://github.com/WebAssembly/wabt.

[44] *WebAssembly Specification - Release 2.0 + tail calls + function references + gc*. Online; accessed 16 of Januar 2024. Jan. 2024. URL: https://webassembly.github.io/gc/core/.

[45] *Allocator Designs*. Online; accessed 19 of Januar 2024. Jan. 2024. URL: https://os.phil-opp.com/allocator-designs/.

[46] *React*. Online; accessed 11 of Januar 2024. Jan. 2024. URL: https://react.dev/.

[47] *NPM*. Online; accessed 11 of Januar 2024. Jan. 2024. URL: https://www.npmjs.com/.

[48] *NPM - use-file-picker*. Online; accessed 11 of Januar 2024. Jan. 2024. URL: https://www.npmjs.com/package/use-file-picker.

[49] *React hooks*. Online; accessed 11 of Januar 2024. Jan. 2024. URL: https://www.w3schools.com/react/react_hooks.asp.

[50] *Using the WebAssembly JavaScript API*. Online; accessed 12 of Januar 2024. Jan. 2024. URL: https://developer.mozilla.org/en-US/docs/WebAssembly/Using_the_JavaScript_API.

[51] *NPM - @wasmer/wasi*. Online; accessed 12 of Januar 2024. Jan. 2024. URL: https://www.npmjs.com/package/@wasmer/wasi.

[52] *Window prompt()*. Online; accessed 18 of Januar 2024. Jan. 2024. URL: https://www.w3schools.com/jsref/met_win_prompt.asp.

[53] *Window: localStorage property*. Online; accessed 19 of Januar 2024. Jan. 2024. URL: https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage.

[54] *Validation Algorithm*. Online; accessed 20 of November 2023. Nov. 2023. URL: https://webassembly.github.io/spec/core/appendix/algorithm.html?highlight=validation.

[55] *Unary negation*. Online; accessed 20 of November 2023. Nov. 2023. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Unary_negation.

[56] *Logical AND*. Online; accessed 17 of November 2023. Nov. 2023. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_AND.

[57] *Memory Instructions*. Online; accessed 15 of December 2023. Dec. 2023. URL: https://webassembly.github.io/gc/core/bikeshed/#syntax-instr-memory.

[58] *Array slice*. Online; accessed 15 of December 2023. Dec. 2023. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/slice.

[59] *Shallow copy*. Online; accessed 15 of December 2023. Dec. 2023. URL: https://developer.mozilla.org/en-US/docs/Glossary/Shallow_copy.

[60] Alceste Scalas. *Module 9: Closures*. Online; accessed 1 of November 2023. DTU, Nov. 2023. URL: https://courses.compute.dtu.dk/02247/f23/closures.html#closure-conversion-of-a-lambda-term.

[61] *A new way to bring garbage collected programming languages efficiently to WebAssembly*. Online; accessed 27 of Januar 2024. Jan. 2024. URL: https://v8.dev/blog/wasm-gc-porting.

[62] *Branch-level tree-shaking*. Online; accessed 16 of Januar 2024. Jan. 2024. URL: https://www.assemblyscript.org/concepts.html#branch-level-tree-shaking.

[63] *wasm-tools*. Online; accessed 15 of Januar 2024. Jan. 2024. URL: https://github.com/bytecodealliance/wasm-tools.

[64] *WebAssembly tail calls*. Online; accessed 8 of February 2024. Feb. 2024. URL: https://v8.dev/blog/wasm-tail-call.

[65] *Assemblyscript - No closures*. Online; accessed 29 of Januar 2024. Jan. 2024. URL: https://www.assemblyscript.org/status.html#on-closures.

10 Acronyms

DTU	Technical University of Denmark
OS	Operating System
WASI	The WebAssembly System Interface
VM	Virtual Machine
WAT	WebAssembly Text Format
I/O	Input/Output
IR	Intermediate Representation
ISA	Instruction Set Architecture
LIFO	Last In First Out
JVM	Java Virtual Machine
WORA	Write once, run anywhere
MoSCoW	Must Have, Should Have, Could Have, Won’t Have this time
MVP	Minimum Viable Product
CFG	Context-Free Grammar
AST	Abstract Syntax Tree
SIMD	single instruction multiple data
POSIX	Portable Operating System Interface
PC	Program Counter
CLI	Command-Line Interface
ABI	Application Binary Interface
ASCII	American Standard Code for Information Interchange
DWARF	Debugging With Attributed Record Formats
JIT	just-in-time
AOT	ahead-of-time
API	Application Programming Interface
npm	Node Package Manager
PC	Program Counter
GUI	Graphical User Interface
TUI	Text-based User Interfaces
UI	User Interfaces
IDE	Integrated Development Environment

WGF	WAT Generation Freamework
GC	Garbage Collector
TDD	Test-Driven Development

A Requirements specifications

This specification lists all requirements of *HyggeWasm* and The learning/development tools where *HyggeWasm* is the version of Hygge that can be compiled into WebAssembly.

An initial specification for both specifications was created at the start of the project. During the project, the specifications were modified but largely remained the same.

A.1 Code generation and Language features - Requirements

This specification lists all language features that should be part of *HyggeWasm* in the Must Have, Should Have, Could Have, Won't Have this time (MoSCoW) format.

Must have

1. The generated code must be a valid WAT module.
2. The developer must be able to add a comment to every instruction.
3. The generated code must be formatted in a reasonable and readable way.
4. The generated code must have only one entry point of execution (One main function) that will be implicit and therefore invisible for the Hygge programmer.
5. The backend must be able to produce functional WAT pertaining to the subsequent language features and operators:
   (a) Arithmetic operators
      i. Subtraction
      ii. Addition
      iii. Remainer division (Modulo)
      iv. Division
      v. Square root
      vi. Maximum and minimum between two numbers
   (b) Relational operators
      i. Equality
      ii. Less than
      iii. Less than or equal
      iv. Greater then
      v. Greater than or equal
   (c) Variables
      i. Immutable Variables
      ii. Mutable Variables
      iii. Pre- and post-increment and decrement operators $(++var, var--)$

iv. Addition assignment ($var_1 + = 1$)

v. Subtraction assignment ($var_2 - = 1$)

vi. Multiplication assignment ($var_1 * = 1$)

vii. Division assignment ($var_1 / = 1$)

viii. Remainer division assignment (var_1% $= 1$)

(d) Logical operators

i. Exclusive or (Xor)

ii. Or

iii. And

iv. Short-circuiting:

A. And (&&)

B. Or (||)

(e) Control flow

i. Conditional statements (if-then-else)

ii. While-loops

iii. For-Loops

iv. Do-while-loops

(f) Data structures and operations

i. Structs - Constructor, Access field. Assign field value.

ii. Tuples - Constructor, Access field. Assign field value.

iii. Arrays - Constructor, Access element, Assign element value, Slice Array.

(g) Functions

i. Functions as first-class citizens

ii. Recursive functions declarations

iii. Recursive functions calls

(h) Data types

i. Integer values

ii. Floating-point values

iii. Strings

A. String length - Can obtain the length of a string with a pointer to the string data.

(i) I/O

i. Read integer or float values into module.

ii. Write integer, float, and string values value to output stream

(j) Test specific features
This feature is primarily included to enable easy tests of the hygge test programs.

    i. Assert operator that will end the program or continue execution based on a boolean expression.

Should have

6. The developer should be able to add line comments without a companion instruction.
7. The backend should be able to produce functional WAT pertaining to the subsequent language features and operators:
    (a) Typing and Pattern matching
        i. Discriminated Union Types
        ii. Pattern matching
    (b) Closures

Could have

8. The compiler could use optimization techniques such as peephole optimization.
9. The generated code could generate code in both a flat and nested (Folded) writing style based on input arguments.
10. The hygge language could be extended to let the hygge programmer explicitly add exports of functions.

Won't have (this time)

11. Be able to throw and handle exceptions
12. Have explicit main function declaration

A.2 Developer and learning tools - Requirements

This specification lists all requirements of the development and learning tool application that will aid the exploration, debugging, and execution of *HyggeWasm* programs.

Must have

1. The tool must be able to show the compiled code WAT.
2. The tool must be able to run the WAT code.
3. The tool must be able to display the output of the WAT programs.
4. The tool should be able to accept inputs for the WAT programs to use.
5. The tool must be able to run on both Windows and MacOS.
6. The tool must have eave the same host interface used for testing the compiler.

Should have

7. The tool should be able to place breakpoints in the code.
8. The tool should be able to step through every code instruction.
9. The tool should be able to display the linear memory.

Could have

10. The tool could display the Hygge programs corresponding AST.
11. The tool should be able to display both original and unpacked[1] WAT programs.

Won't have (this time)

12. Be able to show comments in the debugging tool
13. Store debug information in the .wasm files that map to the *.wat* program.
14. Store debug information in the .wasm files that map to the *.hyg* program.
15. Syntax high-lighting of hygge code.

[1]Unpacked form of the code is wherever optional folded instructions have been written in the linear style.

B User interface - Early wireframe

Concept drawings in the form of wireframes.

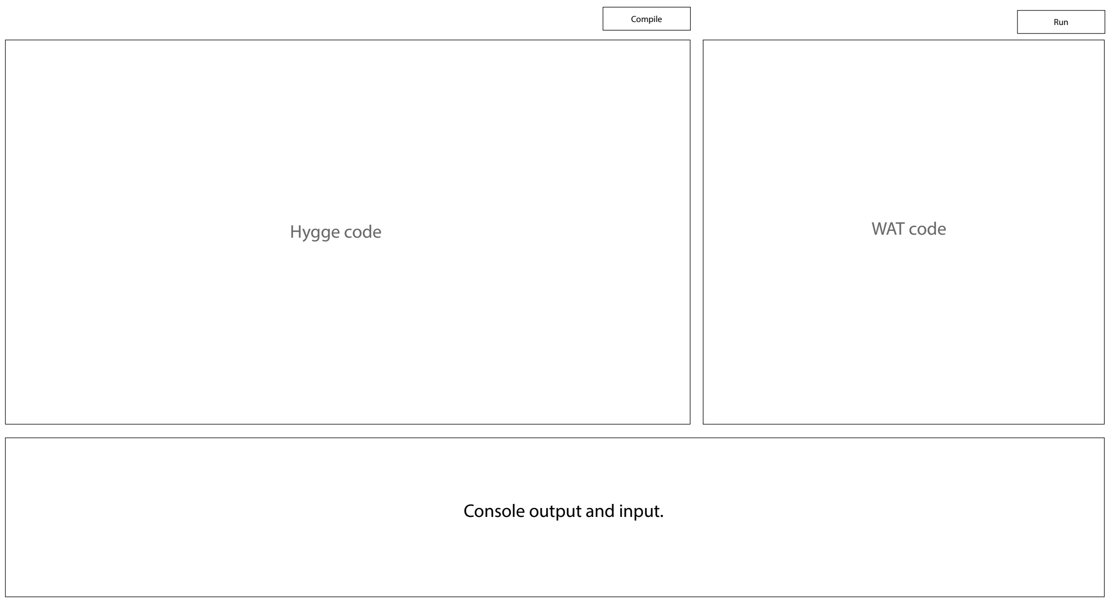


Figure B.1: Wireframe

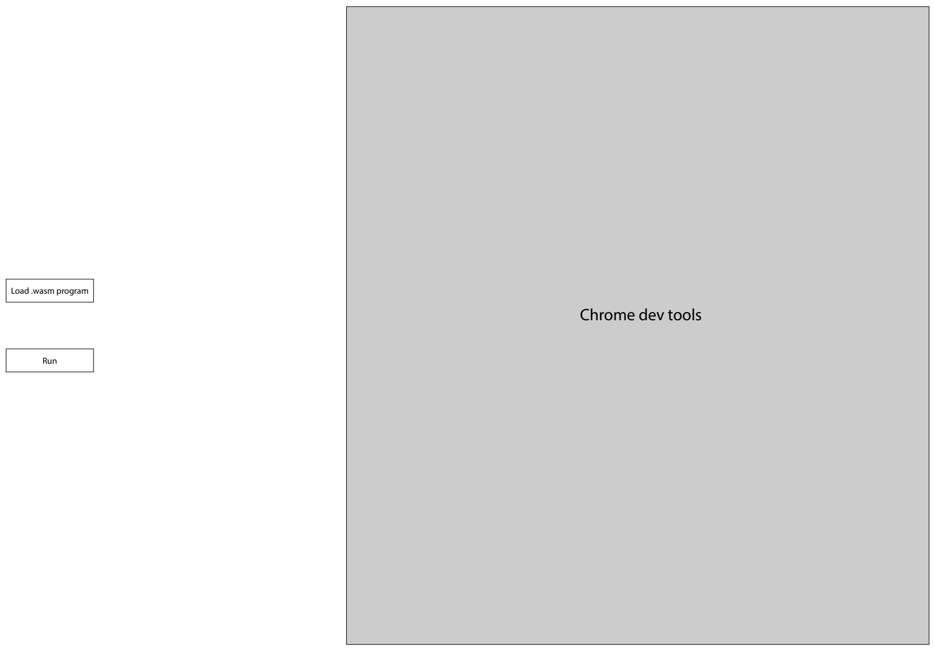


Figure B.2: Wireframe with dev-tools

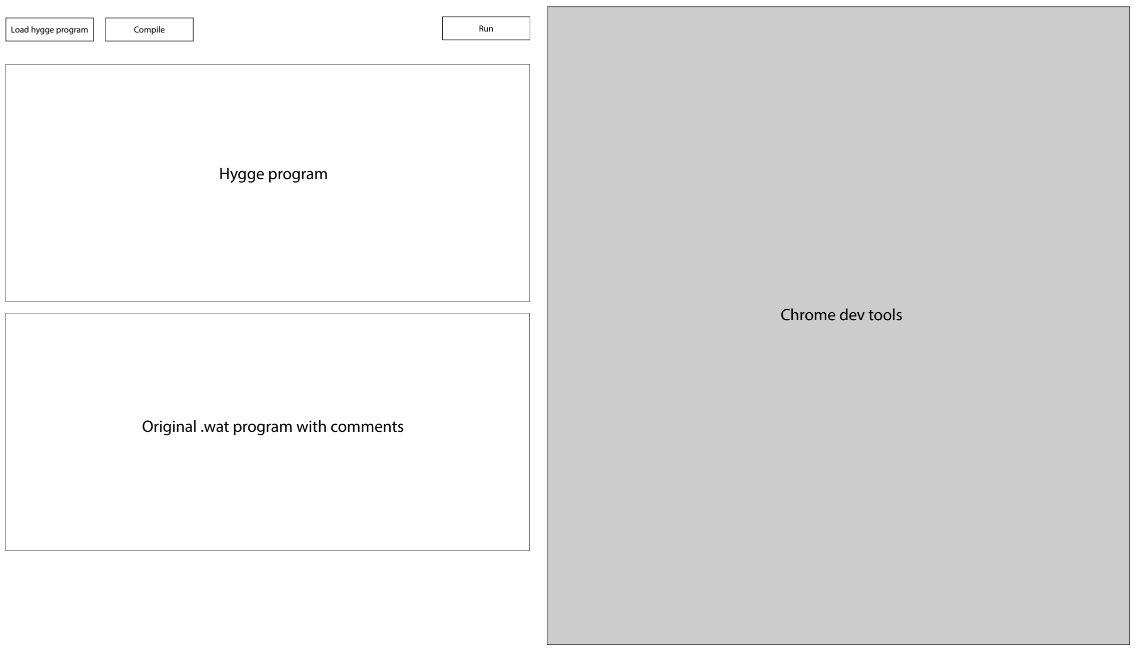
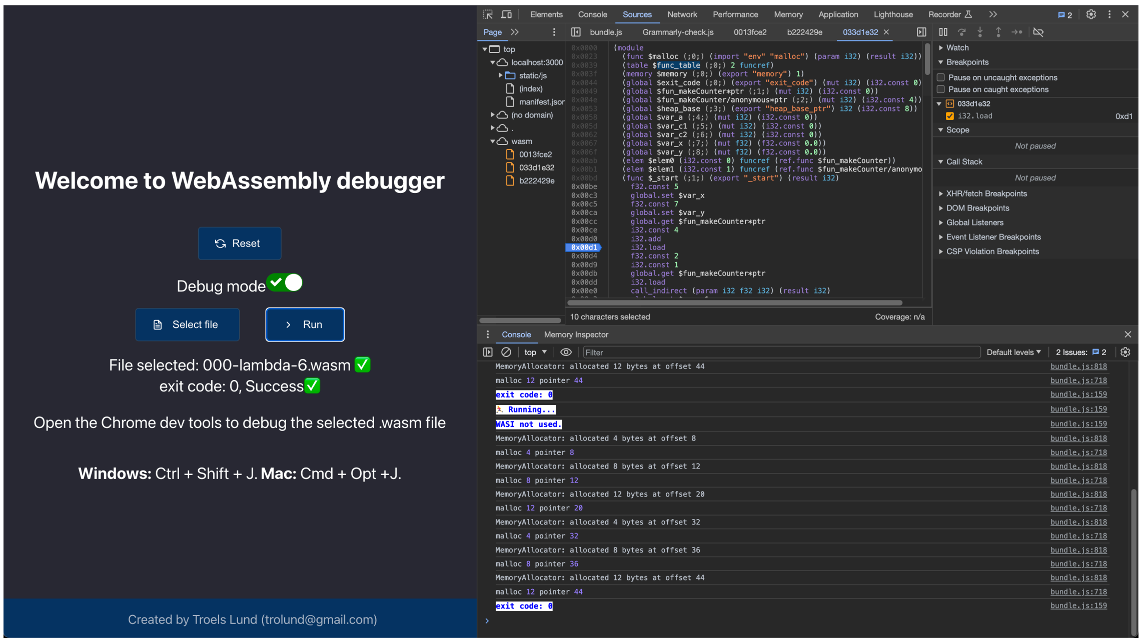


Figure B.3: Wireframe with dev-tools and hygge functionality

C User interface



Welcome to WebAssembly debugger

Debug mode ✅

File selected: 000-lambda-6.wasm ✅
exit code: 0, Success ✅

Open the Chrome dev tools to debug the selected .wasm file

Windows: Ctrl + Shift + J. **Mac:** Cmd + Opt +J.

Created by Troels Lund (trolund@gmail.com)

```
0x0000  (module
0x0023    (func $malloc (;0;) (import "env" "malloc") (param i32) (result i32))
0x0039    (table $func_table (;0;) 2 funcref)
0x003f    (memory $memory (;0;) (export "memory") 1)
0x0044    (global $exit_code (;0;) (export "exit_code") (mut i32) (i32.const 0)
0x0049    (global $fun_makeCounter*ptr (;1;) (mut i32) (i32.const 0))
0x004e    (global $fun_makeCounter/anonymous*ptr (;2;) (mut i32) (i32.const 4))
0x0053    (global $heap_base (;3;) (export "heap_base_ptr") i32 (i32.const 8))
0x0058    (global $var_a (;4;) (mut i32) (i32.const 0))
0x005d    (global $var_c1 (;5;) (mut i32) (i32.const 0))
0x0062    (global $var_c2 (;6;) (mut i32) (i32.const 0))
0x0067    (global $var_x (;7;) (mut f32) (f32.const 0.0))
0x006f    (global $var_y (;8;) (mut f32) (f32.const 0.0))
0x00ab    (elem $elem0 (i32.const 0) funcref (ref.func $fun_makeCounter))
0x00b1    (elem $elem1 (i32.const 1) funcref (ref.func $fun_makeCounter/anonymo
0x00bd    (func $_start (;1;) (export "_start") (result i32)
0x00be      f32.const 5
0x00c3      global.set $var_x
0x00c5      f32.const 7
0x00ca      global.set $var_y
0x00cc      global.get $fun_makeCounter*ptr
0x00ce      i32.const 4
0x00d0      i32.add
0x00d1      i32.load
0x00d4      f32.const 2
0x00d9      i32.const 1
0x00db      global.get $fun_makeCounter*ptr
0x00dd      i32.load
0x00e0      call_indirect (param i32 f32 i32) (result i32)
```

10 characters selected   Coverage: n/a

```
MemoryAllocator: allocated 12 bytes at offset 44        bundle.js:818
malloc 12 pointer 44                                    bundle.js:718
exit code: 0                                            bundle.js:159
🏃 Running...                                           bundle.js:159
WASI not used.                                          bundle.js:159
MemoryAllocator: allocated 4 bytes at offset 8          bundle.js:818
malloc 4 pointer 8                                      bundle.js:718
MemoryAllocator: allocated 8 bytes at offset 12         bundle.js:818
malloc 8 pointer 12                                     bundle.js:718
MemoryAllocator: allocated 12 bytes at offset 20        bundle.js:818
malloc 12 pointer 20                                    bundle.js:718
MemoryAllocator: allocated 4 bytes at offset 32         bundle.js:818
malloc 4 pointer 32                                     bundle.js:718
MemoryAllocator: allocated 8 bytes at offset 36         bundle.js:818
malloc 8 pointer 36                                     bundle.js:718
MemoryAllocator: allocated 12 bytes at offset 44        bundle.js:818
malloc 12 pointer 44                                    bundle.js:718
exit code: 0                                            bundle.js:159
>
```

Figure C.1: UI after Sprint 2

D Manual for learning and development tool

This document contains a user guide for using the learning and development tool.

D.1 Software Version Requirements

First, make sure that your browser is updated. The tool works with:

- Google Chrome: Version 119 or later
- Mozilla Firefox: Version 120 or later

The tool may work perfectly with other browsers, but this has not been tested. Furthermore, it is required to use Google Chrome since this is the best-tested browser for this app.

D.2 Run the learning and development tool locally

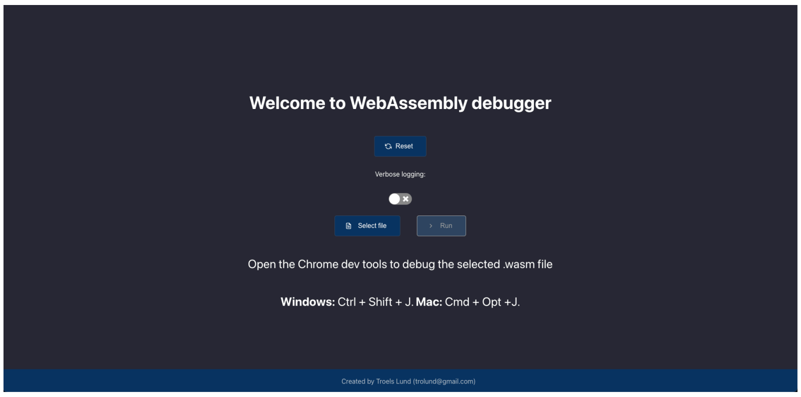
To run the web application locally, go to the solution's root folder, *wasm-debugger*. Then, run the commands shown in D.1. This installs all dependencies of the project and runs the application. The web application will, by default, be served on port `3000`.

```
npm i
npm run start
```

Figure D.1: Run the web application

D.3 Use the learning and development tool

If you have successfully started the web application, open your browser and go to `http://localhost:3000/`. This should give you a view similar to D.2.



Welcome to WebAssembly debugger

Open the Chrome dev tools to debug the selected .wasm file

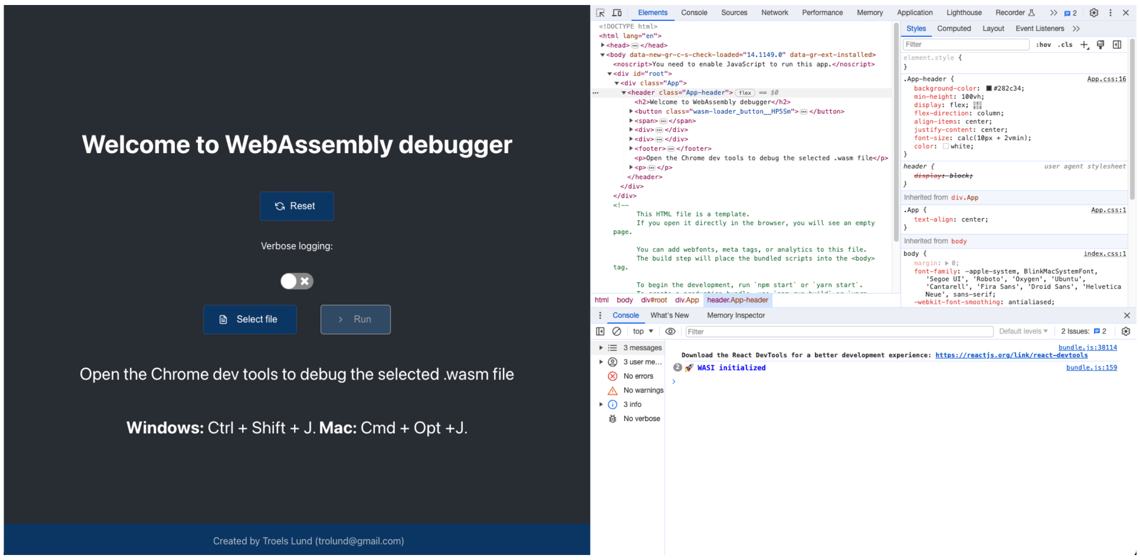
Windows: Ctrl + Shift + J. **Mac:** Cmd + Opt +J.

Created by Troels Lund (trolund@gmail.com)

Figure D.2: Initial state of tool

Our test program will use the Hygge program in D.3.

This application uses a development tool built into the browser. Therefore, to get the full experience, open the development tool by pressing `Ctrl + Shift + J` on a Program Counter (PC) or `Cmd + Opt +J` on a Mac. This should look like D.4.



Welcome to WebAssembly debugger

Verbose logging:

Open the Chrome dev tools to debug the selected .wasm file

Windows: Ctrl + Shift + J. **Mac:** Cmd + Opt +J.

Created by Troels Lund (trolund@gmail.com)

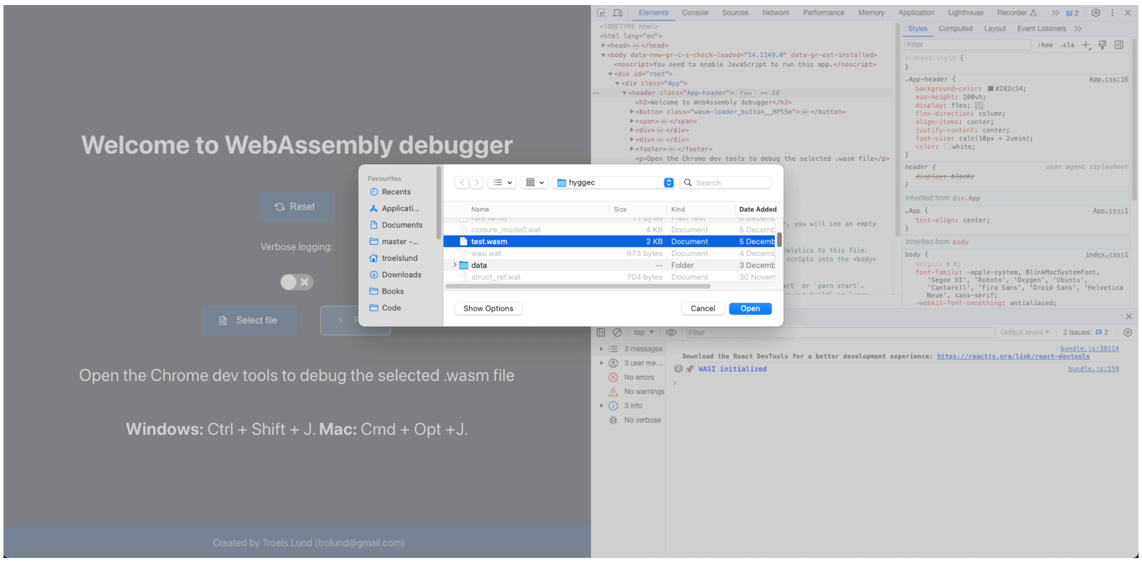
```
<!DOCTYPE html>
<html lang="en">
 ▸<head> ⋯ </head>
 ▾<body data-new-gr-c-s-check-loaded="14.1149.0" data-gr-ext-installed>
    <noscript>You need to enable JavaScript to run this app.</noscript>
   ▾<div id="root">
     ▾<div class="App">
       ▾<header class="App-header"> flex == $0
          <h2>Welcome to WebAssembly debugger</h2>
         ▸<button class="wasm-loader_button__HP5Sm"> ⋯ </button>
         ▸<span> ⋯ </span>
         ▸<div> ⋯ </div>
         ▸<div> ⋯ </div>
         ▸<footer> ⋯ </footer>
          <p>Open the Chrome dev tools to debug the selected .wasm file</p>
         ▸<p> ⋯ </p>
        </header>
      </div>
    </div>
    <!--
          This HTML file is a template.
          If you open it directly in the browser, you will see an empty
    page.

          You can add webfonts, meta tags, or analytics to this file.
          The build step will place the bundled scripts into the <body>
    tag.

          To begin the development, run `npm start` or `yarn start`.
```

```
element.style {
}
.App-header {                                         App.css:16
    background-color: #282c34;
    min-height: 100vh;
    display: flex;
    flex-direction: column;
    align-items: center;
    justify-content: center;
    font-size: calc(10px + 2vmin);
    color: white;
}
header {                                   user agent stylesheet
    ~~display: block;~~
}
Inherited from div.App
.App {                                                 App.css:1
    text-align: center;
}
Inherited from body
body {                                               index.css:1
    margin: ▸ 0;
    font-family: -apple-system, BlinkMacSystemFont,
        'Segoe UI', 'Roboto', 'Oxygen', 'Ubuntu',
        'Cantarell', 'Fira Sans', 'Droid Sans', 'Helvetica
        Neue', sans-serif;
    -webkit-font-smoothing: antialiased;
```

```
3 messages      Download the React DevTools for a better development experience: https://reactjs.org/link/react-devtools   bundle.js:38114
3 user me...    2 🚀 WASI initialized                                                                                    bundle.js:159
No errors
No warnings
3 info
No verbose
```

Figure D.4: Dev tool opened

Then, the program can be loaded by clicking the *Select file* button. This will open a dialog where you can select one binary WebAssembly module file (.wasm). This is shown in D.5.



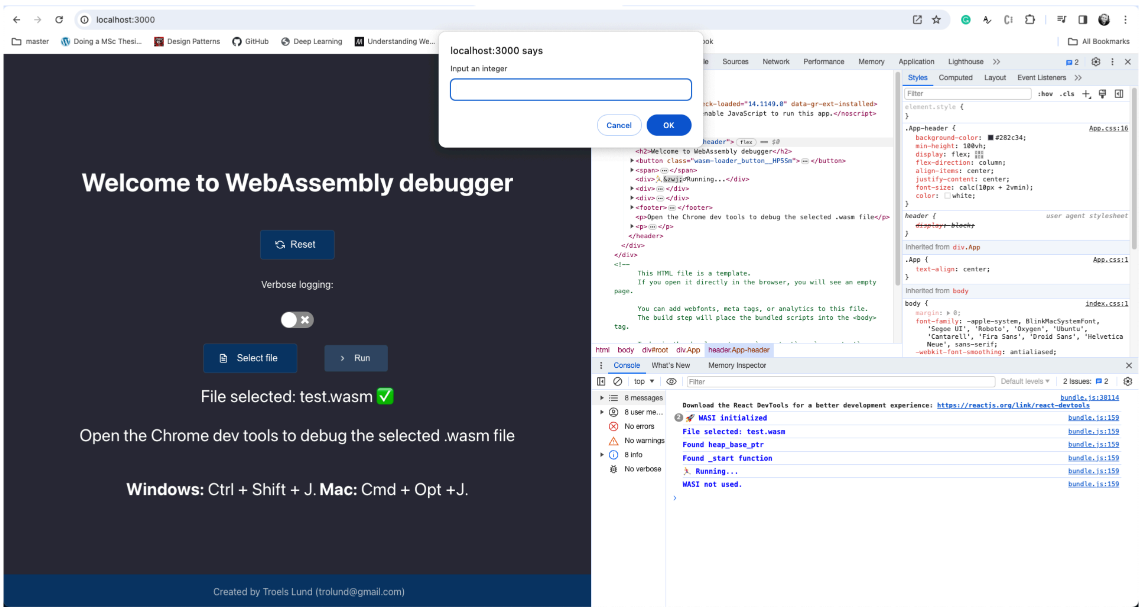
Name	Size	Kind	Date Added
closure_mode0.wat	4 KB	Document	5 Decemb
test.wasm	2 KB	Document	5 Decemb
wasi.wat	873 bytes	Document	4 Decemb
data	--	Folder	3 Decemb
struct_ref.wat	704 bytes	Document	30 Novem

Figure D.5: Open file

Then, the button *Run* can be clicked to start the program's execution. Since the program reads a user input with $readInt()$, will the execution be held and wait for input by showing an alert dialog with an input text field. In this case, asking for an integer value; for this example, we input 10. See D.6.

```
let mutable x: int = readInt(); // read input
println("--------------------------");

let arr: array {int} = array(x, 0); // array constructor

fun f(arr: array {int}, i: int): array {int} = { // recursive function
    if (i < arrayLength(arr)) then { // read length of array as part of
    condition
        arrayElem(arr, i) <- i + 1; // assign value to array element
        f(arr, i + 1) // recursive function call
    }
    else {
        arr // return modified array
    }
};

f(arr, x / 2); // modified array returned

x <- 0; // reset x

do { // do-while to print array data
    println(arrayElem(arr, x)); // read array element
    x <- x + 1 // increment
} while (x < arrayLength(arr)); // read length of array as part of condition

println("--------------------------");

let sliced: array {int} = arraySlice(arr, x / 2, arrayLength(arr)); // slice
    array in half

x <- 0;

do { // do-while to print array data
    println(arrayElem(sliced, x)); // read array element
    x <- x + 1 // increment
} while (x < arrayLength(sliced)); // read length of array as part of
    condition

println("--------------------------");

type OptionalArray = union { // read length of array as part of condition
    Some: array {int};
    None: unit
};

let o: OptionalArray = Some{sliced}; // union constructor

match o with { // match pattern of 'o'
    Some{v} -> println(arrayLength(v));
    None{_} -> println("None")
}
```

Figure D.3: Working with Arrays in Hygge



```
localhost:3000 says
Input an integer
```

Welcome to WebAssembly debugger

File selected: test.wasm ✅

Open the Chrome dev tools to debug the selected .wasm file

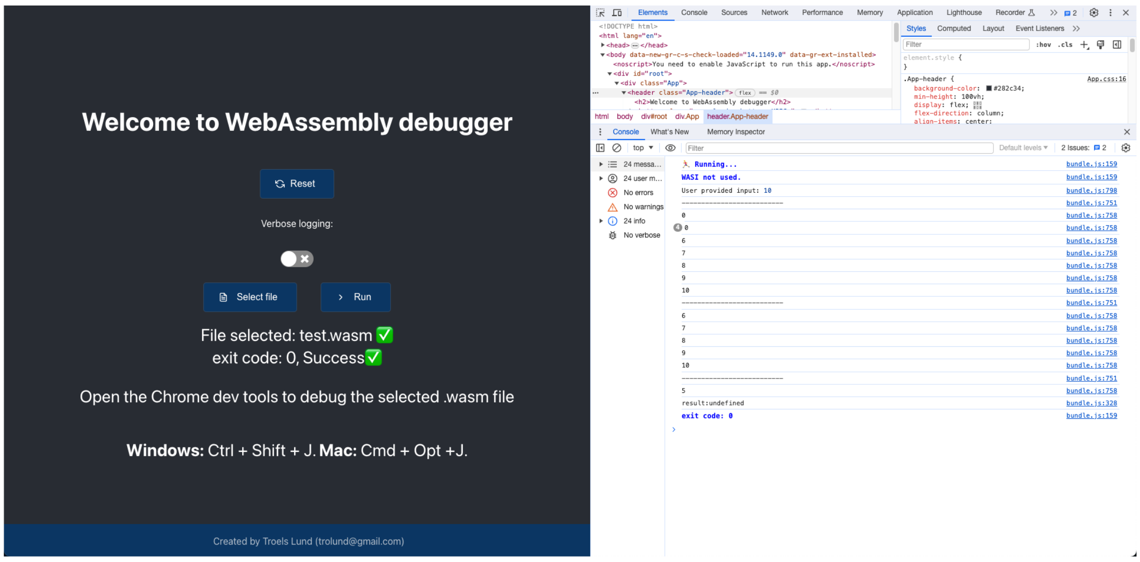
Windows: Ctrl + Shift + J. **Mac:** Cmd + Opt +J.

Created by Troels Lund (trolund@gmail.com)

```
Download the React DevTools for a better development experience: https://reactjs.org/link/react-devtools   bundle.js:38114
🚀 WASI initialized                                              bundle.js:159
File selected: test.wasm                                         bundle.js:159
Found heap_base_ptr                                              bundle.js:159
Found _start function                                            bundle.js:159
🏃 Running...                                                    bundle.js:159
WASI not used.                                                   bundle.js:159
```

Figure D.6: Integer input

After submitting the value, the execution will continue. The output of the program will be shown in the browser console.



Welcome to WebAssembly debugger

File selected: test.wasm ✅
exit code: 0, Success✅

Open the Chrome dev tools to debug the selected .wasm file

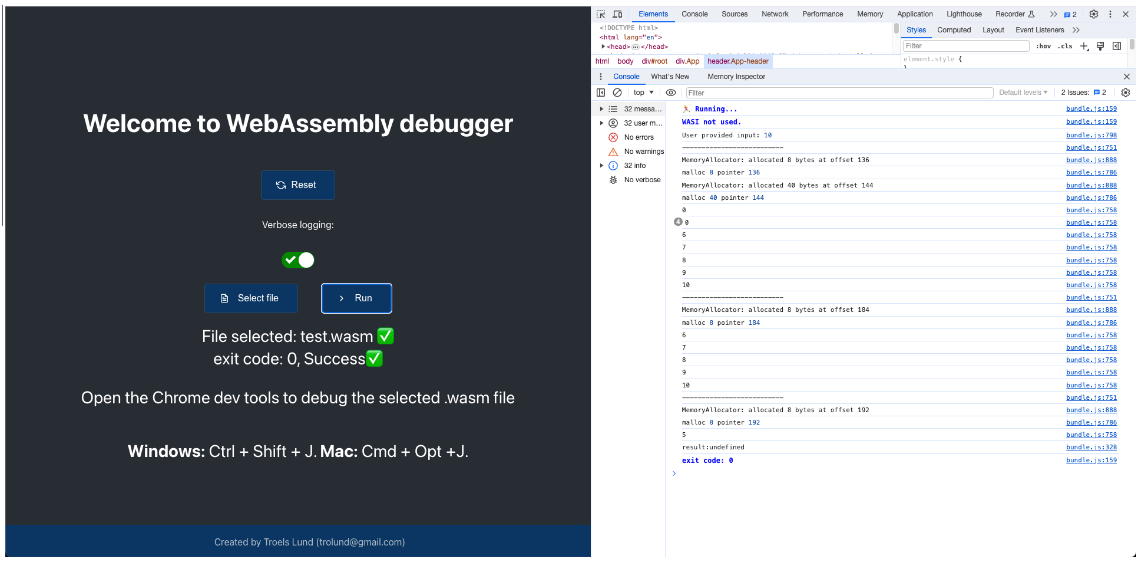
Windows: Ctrl + Shift + J. **Mac:** Cmd + Opt +J.

Created by Troels Lund (trolund@gmail.com)

```
🏃 Running...                     bundle.js:159
WASI not used.                    bundle.js:159
User provided input: 10           bundle.js:798
-------------------------         bundle.js:751
0                                 bundle.js:758
0                                 bundle.js:758
6                                 bundle.js:758
7                                 bundle.js:758
8                                 bundle.js:758
9                                 bundle.js:758
10                                bundle.js:758
-------------------------         bundle.js:751
6                                 bundle.js:758
7                                 bundle.js:758
8                                 bundle.js:758
9                                 bundle.js:758
10                                bundle.js:758
-------------------------         bundle.js:751
5                                 bundle.js:758
result:undefined                  bundle.js:328
exit code: 0                      bundle.js:159
```

Figure D.7: Output in console

If the verbose logging toggle is on, allocation information will be shown in the program compiled in the `External` memory mode. The output can be seen in D.8.



```
Running...                                          bundle.js:159
WASI not used.                                      bundle.js:159
User provided input: 10                             bundle.js:798
-------------------------                           bundle.js:751
MemoryAllocator: allocated 8 bytes at offset 136    bundle.js:888
malloc 8 pointer 136                                bundle.js:786
MemoryAllocator: allocated 40 bytes at offset 144   bundle.js:888
malloc 40 pointer 144                               bundle.js:786
0                                                   bundle.js:758
4 0                                                 bundle.js:758
6                                                   bundle.js:758
7                                                   bundle.js:758
8                                                   bundle.js:758
9                                                   bundle.js:758
10                                                  bundle.js:758
-------------------------                           bundle.js:751
MemoryAllocator: allocated 8 bytes at offset 184    bundle.js:888
malloc 8 pointer 184                                bundle.js:786
6                                                   bundle.js:758
7                                                   bundle.js:758
8                                                   bundle.js:758
9                                                   bundle.js:758
10                                                  bundle.js:758
-------------------------                           bundle.js:751
MemoryAllocator: allocated 8 bytes at offset 192    bundle.js:888
malloc 8 pointer 192                                bundle.js:786
5                                                   bundle.js:758
result:undefined                                    bundle.js:328
exit code: 0                                        bundle.js:159
```

Figure D.8: Vabose output in console

D.4 Debug program

First, load a program and open the development tool, as shown in the previous section.

To place break points in the WebAssebly code that has been loaded, do as follows (use figure D.9 as reference):

1. Click *sources* at the top of the development tool panel.

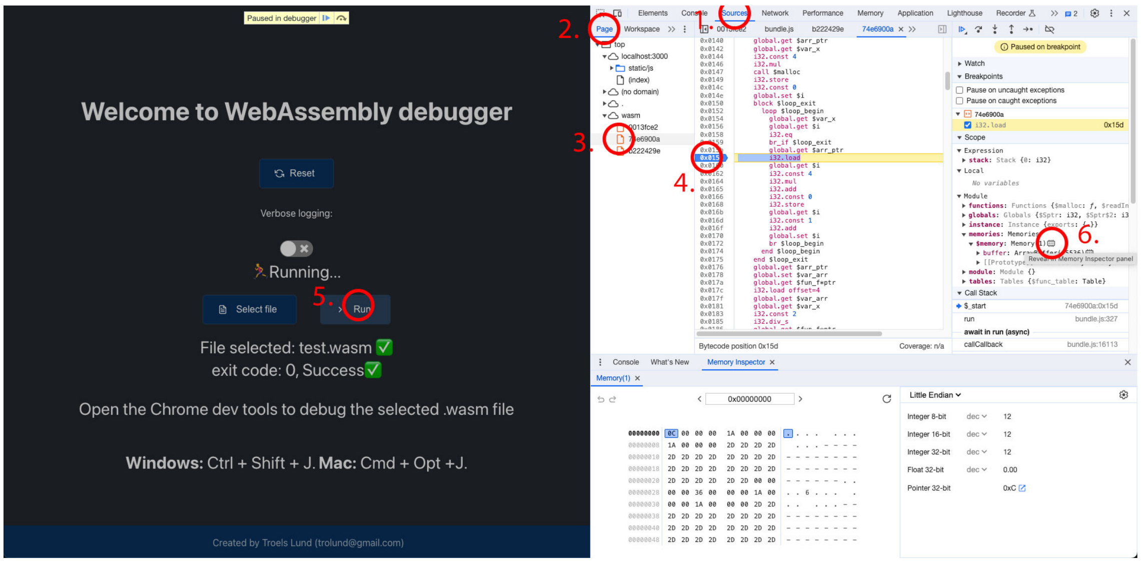
2. Ensure you can see the page's resources by clicking *page*.

3. Find the file you just loaded and click it. Unfortunately, Chrome renames the file for security reasons; look for the file that has the "_start" function.

4. Click on the line number you want to place a breakpoint on.

5. Click *run*. When the breakpoint is reached, the execution stops, and instructions can be stepped through.

6. Click the memory symbol to inspect the linear memory.



File selected: test.wasm ✅
exit code: 0, Success✅

Open the Chrome dev tools to debug the selected .wasm file

Windows: Ctrl + Shift + J. **Mac:** Cmd + Opt +J.

```
0x0140      global.get $arr_ptr
0x0142      global.get $var_x
0x0144      i32.const 4
0x0146      i32.mul
0x0147      call $malloc
0x0149      i32.store
0x014c      i32.const 0
0x014e      global.set $i
0x0150      block $loop_exit
0x0152        loop $loop_begin
0x0154          global.get $var_x
0x0156          global.get $i
0x0158          i32.eq
0x0159          br_if $loop_exit
0x015b          global.get $arr_ptr
0x015d          i32.load
0x0160          global.get $i
0x0162          i32.const 4
0x0164          i32.mul
0x0165          i32.add
0x0166          i32.const 0
0x0168          i32.store
0x016b          global.get $i
0x016d          i32.const 1
0x016f          i32.add
0x0170          global.set $i
0x0172          br $loop_begin
0x0174        end $loop_begin
0x0175      end $loop_exit
0x0176      global.get $arr_ptr
0x0178      global.set $var_arr
0x017a      global.get $fun_f*ptr
0x017c      i32.load offset=4
0x017f      global.get $var_arr
0x0181      global.get $var_x
0x0183      i32.const 2
0x0185      i32.div_s
```

Bytecode position 0x15d   Coverage: n/a

```
Paused on breakpoint
▸ Watch
▾ Breakpoints
  ☐ Pause on uncaught exceptions
  ☐ Pause on caught exceptions
▾ 74e6900a
  ☑ i32.load                                  0x15d
▾ Scope
▾ Expression
  ▸ stack: Stack {0: i32}
▾ Local
    No variables
▾ Module
  ▸ functions: Functions {$malloc: f, $readIn
  ▸ globals: Globals {$Sptr: i32, $Sptr$2: i3
  ▸ instance: Instance {exports: {…}}
  ▾ memories: Memories
    ▾ $memory: Memory(1)
      ▸ buffer: ArrayBuffer(65536)
      ▸ [[Prototype
  ▸ module: Module {}
  ▸ tables: Tables {$func_table: Table}
▾ Call Stack
  ➜ $_start                     74e6900a:0x15d
    run                         bundle.js:327
    await in run (async)
    callCallback                bundle.js:16113
```

Reveal in Memory Inspector panel

Console   What's New   Memory Inspector ×

Memory(1) ×

0x00000000

```
00000000  0C 00 00 00  1A 00 00 00  . . . .   . . .
00000008  1A 00 00 00  2D 2D 2D 2D    . . . - - - -
00000010  2D 2D 2D 2D  2D 2D 2D 2D  - - - - - - - -
00000018  2D 2D 2D 2D  2D 2D 2D 2D  - - - - - - - -
00000020  2D 2D 2D 2D  2D 2D 00 00  - - - - - - . .
00000028  00 00 36 00  00 00 1A 00  . . 6 . . .   .
00000030  00 00 1A 00  00 00 2D 2D  . .   . . . - -
00000038  2D 2D 2D 2D  2D 2D 2D 2D  - - - - - - - -
00000040  2D 2D 2D 2D  2D 2D 2D 2D  - - - - - - - -
00000048  2D 2D 2D 2D  2D 2D 2D 2D  - - - - - - - -
```

Little Endian

Integer 8-bit	dec	12
Integer 16-bit	dec	12
Integer 32-bit	dec	12
Float 32-bit	dec	0.00
Pointer 32-bit		0xC

Figure D.9: Debug and inspect linear memory

E Manual for using the CLI

Guide for running and compiling WebAssembly in the CLI

E.1 Run '.wat'-file

The CLI lets the user run the '.wat'-file with *Wasmtime* and the HyggeWasm runtime.

```
./hyggec wasm test.wat
```

E.2 Run test suite

The CLI lets the user run the test suite.

```
./hyggec test
```

E.3 Compile Hygge program

The CLI lets the user compile a Hygge program to a WAT module. The options are in Table E.1.

```
./hyggec <path to hygge program> -s l -o <path to wat output file> -e -i 0 -m 1
```

Flag	Description	Input's
_	Input	path to hygge program
-s	Writring style	linar ("l") or folded ("f")
-o	Output file	Path to wat output file
-i	System interface	0 - HyggeSI or 1 - WASI
-m	Memory mode	0 - External or 1 - Internal or 2 - Heap (WasmGC)
-e	Execute after compilation	_

Figure E.1: Compile options

F UTF-16 Strings

In an earlier version of the implementation, strings were interpreted as the *Unicode* `UTF-16` format. This had some disadvantages:

- **Advantages:**
  - The length does not have to be explicitly stored because it can be computed.
- **Disadvantages:**
  - Cannot represent the entire Unicode character set.
  - Each character uses more bytes when using common symbols found in ASCII.
  - The console output would not print the entire string when symbols that used more bits were used; see figure F.3.

This led to the abandonment of UTF-16 to instead use UTF-8, which can represent the entire Unicode character set and use less space for commonly used symbols commonly found in ASCII.

The implementation was very similar. When the module has been instantiated, the data will be placed in memory as a structure containing the pair (d, s), where d is the address of the string and s is the size of the string in bytes. How the pair (d, s) is stored in memory is shown in figure F.1. The arrow pointing to the data address field is the pointer left on the stack when evaluating the *StringVal* node.

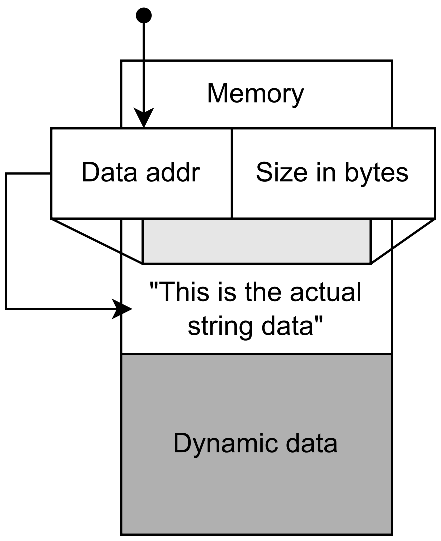
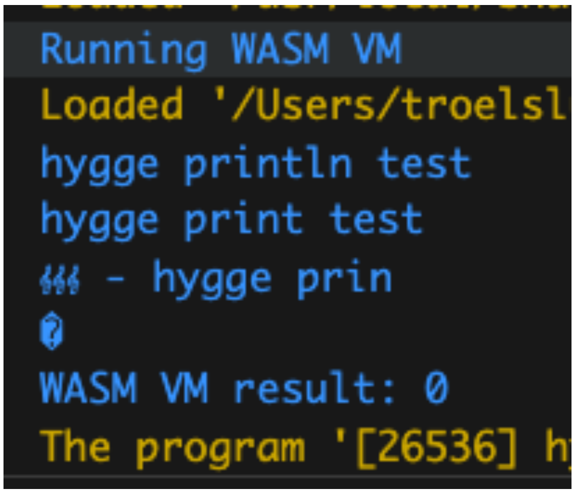


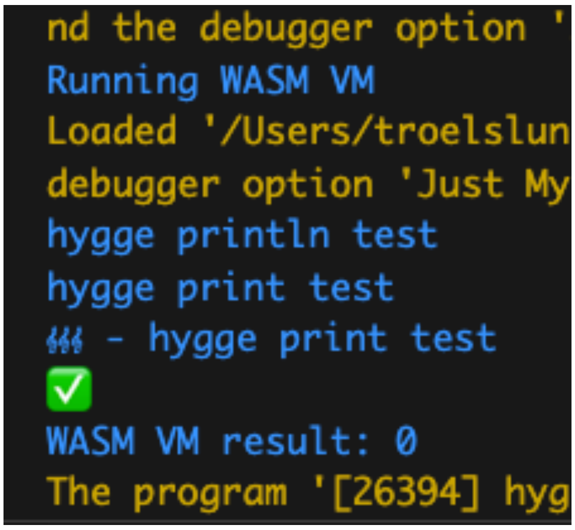
Figure F.1: Strings in memory

As part of its development tools, Google Chrome has a memory inspector that can show how this is stored in the *ArrayBuffer*, see figure F.2.



```
Running WASM VM
Loaded '/Users/troelsl
hygge println test
hygge print test
### - hygge prin
?
WASM VM result: 0
The program '[26536] h
```

(a) UTF-16



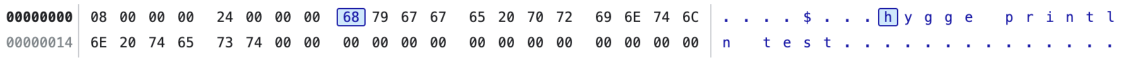
```
nd the debugger option '
Running WASM VM
Loaded '/Users/troelslun
debugger option 'Just My
hygge println test
hygge print test
### - hygge print test
✅
WASM VM result: 0
The program '[26394] hyg
```

(b) UTF-8

Figure F.3: Comparing the use of UTF-8 to UTF-16

```
| StringLength e ->
    let m' = doCodegen env e m

    let instrs =
        m'.GetAccCode()
        @ [ (I32Load_(None, Some(4)), "load string length")
            // divide by 2 to get the number of characters
            (I32Const 1, "push 1 on stack") // or i32.const 2
            (I32ShrS, "divide by 2") ] // or i32.div_s

    m'.ResetAccCode().AddCode(instrs)
```

Figure F.4: Evaluating StringLength node



```
00000000  08 00 00 00  24 00 00 00  68 79 67 67  65 20 70 72  69 6E 74 6C  . . . . $ . . . h y g g e   p r i n t l
00000014  6E 20 74 65  73 74 00 00  00 00 00 00  00 00 00 00  00 00 00 00  n   t e s t . . . . . . . . . . . . . .
```

Figure F.2: String in Arraybuffer show in Chrome dev-tools

One advantage of using UTF-16 is that the string length can be computed from the number of bytes because UTF-16 is a fixed-size format, where each character occupies a constant number of bytes. This means the Number of bytes$/2 =$ String length. Therefore, there was no need to store the length in an additional field as long as only ASCII characters were used.

The intrinsic operation `stringLength(s)` was implemented as shown in code example F.4.

G Hygge System interface specification

The specification defines the Hygge system interface, HyggeSI. The function described in the spec can be imported by a wasm module while running in a HyggeSI-compatible environment, meaning an environment that implements the Hygge runtime. This is similar to ABI used by other compiler toolchains[28].

G.1 Interface

- `malloc`:
  - Module: ”env”
  - Function: ”malloc”
  - Type: $(int) \rightarrow int$
  - Description: Used to allocate linear memory block
- `readInt`:
  - Module: ”env”
  - Function: ”readInt”
  - Type: $() \rightarrow int$
  - Description: Reads an integer from input
- `readFloat`:
  - Module: ”env”
  - Function: ”readFloat”
  - Type: $() \rightarrow float$
  - Description: Reads a floating-point number from input
- `writeInt`:
  - Module: ”env”
  - Function: ”writeInt”
  - Type: $(int) \rightarrow ()$
  - Description: Writes an integer to output
- `writeFloat`:
  - Module: ”env”
  - Function: ”writeFloat”
  - Type: $(float) \rightarrow ()$
  - Description: Writes a floating-point number to output
- `writeS`:
  - Module: ”env”

- Function: ”writeS”
- Type: $(int, int, int) \to ()$
- Description: Writes a string value from linear memory to output. The first argument is an address to the first character in memory. The second is the length in bytes.

  The last value indicates if a new line character should be inserted as the last character in the sequence. If this value is $v > 0$ a new line is inserted otherwise not.

H Optimisation data set

The optimization data set is in the file *stats.csv*. The file includes data about the number of instructions in the regular WebAssembly module and the optimized one, the difference in instruction count between the two, and a calculated percentage reduction. An example of the data can be seen in table H.1.

name of file	instr count	instr count after op	diff	% reduction
000-negate-float	16	0	16	100.00
hygge-union-structs	210	206	4	1.90
hygge-union	105	104	1	0.95
if-return	774	468	306	39.53
insertionSort	936	732	204	21.79
letClousure	244	237	7	2.87
linearSearch	536	423	113	21.08
memory-grow	142	142	0	0.00
memory-static	3	3	0	0.00
mergeSort	1458	1247	211	14.47
multiplier	82	68	14	17.07
quickSort	1094	889	205	18.74
selectionSort	977	770	207	21.19

Table H.1: Exsample of optimization data

I Visual studio code configurations

Tasks

```
{
    // See https://go.microsoft.com/fwlink/?LinkId=733558
    // for the documentation about the tasks.json format
    "version": "2.0.0",
    "tasks": [
        {
            "label": "build",
            "command": "dotnet build",
            "type": "shell",
            "group": "build",
            "presentation": {
                "reveal": "silent"
            },
            "problemMatcher": "$msCompile"
        },
        {
            "label": "towasm",
            "command": "wat2wasm", // Could be any other shell command
            "args": [
                "--debug-names", // keep names from wat file
                "test.wat",
                "-o",
                "test.wasm"
            ],
            "presentation": {
                "reveal": "silent"
            },
            "type": "shell"
        },
        {
            "label": "wasm-tools",
            "command": "wasm-tools", // Could be any other shell command
            "args": [ // wasm-tools parse struct_working.wat -o struct_working.wasm
                "parse",
                "test.wat",
                "-o",
                "test.wasm",
            ],
            "presentation": {
                "reveal": "silent"
            },
            "type": "shell"
        },
        {
            "label": "wasmas",
            "command": "wasm-as", // Could be any other shell
    command
            "args": [
                "--debuginfo", // keep names from wat file
                "--debug",
                "--enable-gc",
                "--enable-reference-types",
                "test.wat",
                "-o",
                "test.wasm"
            ],
            "presentation": {
                "reveal": "silent"
            },
            "type": "shell"
        },
        {
            "label": "run",
            "command": "./hyggec", // Could be any other shell
    command
            "args": [
                "wasm", // keep names from wat file
                "test.wat"
            ],
            "presentation": {
                "reveal": "always"
            },
            "type": "shell"
        }
    ]
}
```

Launch configurations - launch.json

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Compile hygge program and produce test.wasm
    file",
            "type": "coreclr",
            "request": "launch",
            "preLaunchTask": "build",
            "postDebugTask": "wasm-tools", // towasm or wasmas
    or wasm-tools
```

```
            "program": "${workspaceFolder}/bin/Debug/net8.0/hyggeWasm.dll",
            "args": [
                "${workspaceFolder}/examples/hygge/exam.hyg",
                "--style",
                "f", // folded (f) or linear (l)
                "-o",
                "test.wat",
                // "--optimize", "4", // produce WAT with optimizations
                "-e", // execute file with wasmtime
                "-i", "0", // 0 - hygge, 1 - wasi
                "-m", "0" // 0 - external, 1 - internal, 2 - heap (WasmGC)
            ],
            "cwd": "${workspaceFolder}",
            "stopAtEntry": false,
            "console": "internalConsole"
        },
        {
            "name": "Run tests",
            "type": "coreclr",
            "request": "launch",
            "preLaunchTask": "build",
            "program": "${workspaceFolder}/bin/Debug/net8.0/hyggeWasm.dll",
            "args": [
                "test",
                // "-f",
                // "wasm"
            ],
            "cwd": "${workspaceFolder}",
            "stopAtEntry": false,
            "console": "internalConsole"
        }
    ]
}
```

J Module API

Methods

```
member this.GetHostingList() : list<string>

member this.AddToHostingList(name: string) : Module

member this.GetAllFuncs() : list<string * Commented<FunctionInstance>>

member this.ReplaceFuncs(list: list<(string * FunctionInstance) * string>) :
    Module

member this.RemoveLocal(name: string) : Module

member this.IsFunction(name: string) : bool

member this.LookupFuncInFuncTable(name: string) : option<int *
    FunctionInstance * string>

member this.AddFuncRefElement(label: string, index: int) : Module

member this.AddCode(instrs: Instr.Wasm list) : Module

member this.AddCode(instrs: Commented<Instr.Wasm> list) : Module

member this.GetTypes() : List<TypeDef>

member this.GetAccCode() : list<Commented<Wasm>>

member this.ResetAccCode() : Module

member this.ResetLocals() : Module

member this.AddLocals(locals: list<Local>) : Module

member this.GetLocals() : Module

member this.AddLocals(name: string, locals: list<Local>) : Module

member this.AddInstrs(name: string, instrs: Instr.Wasm list) : Module

member this.AddInstrs(name: string, instrs: Commented<Instr.Wasm> list) :
    Module

member this.AddImport(i: Import) : Module

member this.AddTypedef(typedef: TypeDef) : Module

member this.AddFunction(name: string, f: Commented<FunctionInstance>) : Module

member this.AddFunction(name: string, f: Commented<FunctionInstance>,
    addTypedef: bool) : Module

member this.AddTable(t: Table) : Module

member this.AddMemory(m: Memory) : Module

member this.AddGlobal(g: Global) : Module

member this.AddGlobals(globals: list<Global>) : Module

member this.AddExport(e: Export) : Module

member this.AddData(d: Data) : Module

member this.Combine(m: Module) : Module
```

Static Members

```
static member (+)(wasm1: Module, wasm2: Module) : Module

static member (++)(wasm1: Module, wasm2: Module) : Module

static member (++)(instr: Commented<Instr.Wasm> list, wasm2: Module) : Module
```

K 02247 Compiler Construction - Project Report

This is a report that was created during the course 02247 Compiler Construction at DTU by the members of group 6:

Bogdan Turdasan (s222981)
Cristina Ailoaei (s213804)
Troels Lund (s161791)

It is attached as the file: *Compiler_Construction.pdf*

L Function restructuring and organization

This shows how functions are reorganized from the Hygge program in shown in code snippet L.1 to the WAT module in code snippet L.2.

```
let f_outer: (int, int) -> int = fun(x: int, y: int) -> {
    let f_inner: (int, int) -> int = fun(x: int, y: int) -> {
        x + y + 2
    };

    f_inner(x, y)
};

assert(f_outer(1, 2) = 5)
```

Figure L.1: Organisation of functions in Hygge

```
...
(func $_start  (result i32)
    ;; execution start here:
    ...
    i32.const 0 ;; exit code 0
    return ;; return the exit code
)
(func $fun_f_inner (param $cenv i32) (param $x i32) (param $y i32) (result i32
    ) ;; function fun_f_inner
    ...
)
(func $fun_f_outer (param $cenv i32) (param $x i32) (param $y i32) (result i32
    ) ;; function fun_f_outer
    ...
)
...
```

Figure L.2: Organisation of functions in WAT module

M Example of Internal bump allocation

This Appendix shows an example of the WAT code that grows *linear memory* at runtime.

```
(if
    (i32.ge_s ;; size need > current size
      (i32.add ;; find size need to allocate
        (global.get $heap_base) ;; get heap base
        (i32.const 8) ;; size of struct
      )
      (i32.mul ;; find current size
        (memory.size) ;; memory size
        (i32.const 65536) ;; page size
      )
    )
  (then
    (drop ;; drop new size
      (memory.grow ;; grow memory if needed
        (i32.div_s ;; grow memory!
          (i32.add ;; find size need to allocate
            (global.get $heap_base) ;; get heap base
            (i32.const 8) ;; size of struct
          )
          (i32.const 65536) ;; page size
        )
      )
    )
  )
)
(global.set $Sptr ;; set struct pointer var, have been hoisted
  (global.get $heap_base) ;; leave current heap base address
  (global.set $heap_base ;; set base pointer
    (i32.add ;; add size to heap base
      (global.get $heap_base) ;; get current heap base
      (i32.const 8) ;; size of struct
    )
  )
)
```

Figure M.1: Internal bump allocation

N Unrealized Endeavors

This section provides a synopsis of efforts not integrated into the project's development.

N.1 Code generation exclusively for the linear writing style of WAT

At the start of the project, the author of this thesis wanted to start generating code in the simplest possible way. Therefore, it was decided that the code generation would exclusively target the linear writing style of WAT. Later, the project's scope was changed to incorporate the folded writing style.

Changing the IR defined by WGF was necessary to accomplish this. The IR needed to capture the instruction's hierarchal relationship in order to produce the folded form.

This entailed an extensive rewrite of the code generation and all other functionality that work with the WGF IR, such as the peephole optimizations.

The following sections exemplify how this change impacted the project.

N.1.0.1 Rewrite of peephole optimizations

In many scenarios, the peephole optimization code for the non-nested IR was significantly simpler to work with and do pattern matching with. In code snippet N.1 is the implementation of the optimization described in section 6.3 before the IR was changed to preserve the hierarchal relationship.

This can be compared with the code snippet presented in N.2, which achieves an identical optimization. However, it is more intricate due to the fact that the pertinent instructions are no longer situated in direct proximity to each other.

```
| (LocalTee (x, instrs), c) :: (Drop, _) :: rest ->
    // should be a local.set
    (LocalSet (x, instrs), c) :: optimizeInstr rest
```

Figure N.1: Optimisation code after the IR was nested

```
| (Drop(subTree), _) :: rest when isLocalTee (List.last subTree) ->
    // match!, we can remove the drop
    // and replace the local.tee with a local.set
    let subTree' = List.take (List.length subTree - 1) subTree

    match List.last subTree with
    | (LocalTee (x, instrs), c) ->
        let localSet = (LocalSet (x, instrs), c)
        let rest' = optimizeInstr rest
        subTree' @ [localSet] @ rest'
    | _ -> failwith "should not happen"
```

Figure N.2: Optimisation code after the IR was nested

The nested IR was an advantage when implementing the drop branch pruning since additional information the nesting provided made it trivial to identify what instructions produced the value that had to be dropped. This process is described in 6.2.

O Source code

HyggeWasm The source code of HyggeWasm is attached as the file: *hyggeWasm-Thesis.zip*. This includes the *HyggeC*, WGF, and the *WasmTimeDriver* projects.

Development and Learning tool The source code of the Development and Learning tool is attached as the file: *Wasm-Debugger-Thesis.zip*.

Compiled test programs All test programs are compiled, both '.wat' files and binary '.wasm' files can be found in the folder. They can all be found in the folder *Compiled test programs*.