

Bachelor's Thesis

SmallJ to WASM Compiler

Sven Elvedi and Timo Strolz

Spring Term 2026

Advisor: Thomas Corbat
Co-Advisor: Olivier Lischer
Examiner: Guido Zraggen
Co-Examiner: Hannes Badertscher



Department of Computer Science
Eastern Switzerland University of Applied Sciences (OST)
Campus Rapperswil

Abstract

This thesis presents the design and implementation of a WebAssembly backend for the SmallJ compiler, a simplified Java-like language used in the Compiler Construction course at OST. The existing SmallJ toolchain compiles source programs to a custom stack-based bytecode that can be executed on a dedicated runtime built on top of the .NET platform. The new backend targets WebAssembly to provide portable, sandboxed execution and to remove the “virtual machine within a virtual machine” architecture of the current runtime. In addition, the backend serves as an educational example of how different compilation targets influence compiler design.

The backend performs ahead-of-time compilation by translating SmallJ bytecode to WebAssembly. Since WebAssembly requires structured control flow, an explicit lifting phase reconstructs high-level constructs from unstructured bytecode jumps by building control flow graphs and applying a stackifier-style algorithm. Object-oriented features such as classes, inheritance and dynamic dispatch are mapped to WebAssembly’s GC and reference-types proposals. Because WebAssembly provides no native string type, a custom GC-managed string representation is introduced to support string operations. The backend is fully integrated into the existing C#-based SmallJ compiler and validated through extensive code-generation and runtime tests.

The result of this thesis is a fully functional SmallJ to WebAssembly compiler backend together with a lightweight runtime embedding that supports the complete set of features offered by the original SmallJ backend and runtime. The solution satisfies the functional requirements of the SmallJ language and is suitable for educational use, providing practical insight into compiler backends, runtime-management, control flow analysis and WebAssembly code generation.

Contents

Abstract	i
Contents	ii
1 Introduction	1
1.1 Background	1
1.2 Task and Goal	1
1.3 Conditions	2
1.4 Preliminary Work and Limitations	2
1.5 Thesis Outline	3
1.6 Summary	3
2 Analysis	4
2.1 SmallJ	4
2.1.1 Language	4
2.1.2 Compiler	4
2.1.3 Bytecode	5
2.1.4 Runtime	5
2.2 WebAssembly (WASM)	5
2.2.1 The Basics of WebAssembly	5
2.2.2 Execution Environments	6
2.2.3 Syntax and Semantics	8
2.2.4 Functions	11
2.2.5 Function Tables	12
2.2.6 Memory	13
2.2.7 Strings	15
2.2.8 Proposals	18
2.3 An Overview of Existing WASM Compilers	19
2.3.1 .NET Blazor WASM	19
2.3.2 J2CL	20
2.3.3 TeaVM	20
2.3.4 CheerpJ	20
2.3.5 GraalVM	20
2.3.6 JWebAssembly	20
2.4 Direct vs. Indirect Compilation	21
2.4.1 Direct Compilation	21
2.4.2 Indirect Compilation	21
2.5 AOT Compilation vs. Compiling the Runtime	21

2.5.1	AOT Compilation	22
2.5.2	Compiling the Runtime	22
2.6	Using an Existing Backend	22
2.6.1	The Dragon: LLVM	23
2.6.2	An Alternative to LLVM: Cranelift	23
2.6.3	Almost WASM: Binaryen	23
2.7	Architectural Comparison	23
2.8	Control Flow Graphs	25
2.8.1	Dominance	26
2.8.2	Edges	26
2.8.3	Reducibility	27
2.8.4	Relevance of Reducibility	29
2.9	Lifting Algorithms	30
2.9.1	Relooper	30
2.9.2	Stackifier	30
2.9.3	Comparison	31
2.9.4	Example	31
2.10	Summary	33
3	Design	34
3.1	Architectural Decisions	34
3.1.1	Lifting Step	34
3.1.2	Produces Value	35
3.1.3	SmallJ to WASM Mapping of OO-Features	37
3.1.4	String Handling	37
3.2	Architectural Overview	39
3.3	Compiler Pipeline	42
3.4	WASM Embedding	44
3.5	Summary	44
4	Implementation	45
4.1	Control Flow Graph Construction	45
4.2	Stackifier Implementation	46
4.2.1	Preprocessing	46
4.2.2	The Algorithm	48
4.2.3	Summary	51
4.3	WASM Generation	51
4.3.1	Layouts	52
4.3.2	Builders	53
4.3.3	Emitters	54
4.4	Object-Oriented Features	55

4.4.1	Virtual Method Dispatch	56
4.5	Strings	57
4.6	Specific Builder-Layout-Emitter Implementations	59
4.6.1	Module	59
4.6.2	TypeSection	59
4.6.3	ImportSection	60
4.6.4	FuncRefTable	60
4.6.5	StringSection	60
4.6.6	FunctionSection	61
4.6.7	MainExport	62
4.7	Changes to the Bytecode Generator	62
4.8	WASM Runtime	63
4.9	Dependencies	63
4.9.1	Existing Codebase	63
4.9.2	C#/.NET	64
4.9.3	NUnit	64
4.9.4	Moq	64
4.9.5	Microsoft Automatic Graph Layout (MSAGL)	64
4.9.6	Wasmtime	64
4.10	Testing	65
4.10.1	Standard Tests	65
4.10.2	Runtime Tests	65
4.10.3	Coverage Report	65
4.11	Summary	66
5	Conclusion	67
5.1	Results	67
5.1.1	WASM Backend	67
5.1.2	Runtime Embedding	67
5.1.3	Control Flow Analysis	68
5.1.4	Current Limitations	68
5.2	Evaluation	68
5.2.1	Success	69
5.3	Outlook	69
	List of Abbreviations	70
	Bibliography	71
	List of Figures	76

List of Tables	77
List of Code Listings	78
Appendix	79
A Assignment	80
B SmallJ Language Report	84
C SmallJ Bytecode Reference	95

1 Introduction

This chapter is an introduction to the thesis and gives an overview of the structure of this document. It describes the starting point from which the project emerged, the problem addressed by the study, the conditions under which the research was conducted, the work completed prior to the study and finally an overview of the subsequent chapters.

1.1 Background

Since the idea for this project emerged from the Compiler Construction course at OST it is expected that the results of this project can be used in future iterations of the course. With the Compiler Construction course [1] OST offers students the opportunity to learn how to build a compiler. The program is structured in such a way that students learn the theoretical concepts of compiler and runtime design as well as how to implement them during the lectures and then apply the acquired knowledge in a project. During the project, students have to implement selected compiler and runtime features. All students start with a code base that provides an unfinished version of the compiler as a basic framework. The goal of the project is a simple but fully functional compiler for SmallJ, a simplified version of Java, developed by Luc Bläser as an educational language for teaching compiler and runtime design [2].

The SmallJ compiler is implemented in C# and compiles SmallJ to SmallJ bytecode, which is a simple stack-based bytecode format. Instead of implementing a runtime from scratch, the .NET runtime is used. However, students are still required to implement memory allocation and garbage collection on their own, instead of relying on the .NET runtime for these features.

As an optional task, students can extend the SmallJ runtime with a Just-In-Time (JIT) compiler for native execution. However, the current project framework only supports JIT compilation for x86-64 architectures. Since some students are using ARM-based devices and can therefore not use the provided framework for JIT compilation, the task is optional for all students. Further details about the existing SmallJ compiler are provided in section 2.1.

1.2 Task and Goal

The task of this thesis is to implement a second backend for the SmallJ compiler that compiles to WebAssembly (WASM) [3]. WASM, like SmallJ bytecode, uses a stack-based virtual machine model and is often employed as

an intermediate compilation target. Almost all modern web browsers support the execution of WASM code, which makes it an interesting target for the SmallJ compiler. But WASM is not limited to web browsers, it can also be executed in other environments, such as Node.js or standalone runtimes like Wasmtime [4, 5]. Further details about WASM are provided in section 2.2.

The final product of the thesis is a fully functional SmallJ-to-WASM compiler, including both the WASM backend and a custom runtime embedding for executing the generated WASM code. Together, the new compiler backend and the custom runtime embedding support the following features:

- Compilation of basic arithmetic and logical instructions
- Compilation of control flow constructs, such as loops and conditionals
- Mapping of object-oriented features (classes, objects and method calls)
- Dynamic dispatch for method calls
- Memory management (heap allocation and garbage collection)
- I/O operations such as reading and writing strings and integers

The goal of the thesis is for the final product to be suitable for usage in the Compiler Construction course at OST. This would give students insights into how a different target intermediate representation affects the design and implementation of a compiler and runtime. The original assignment is provided in appendix A.

1.3 Conditions

The WASM compiler backend has to be integrated into the existing SmallJ compiler code base. Therefore, the WASM backend needs to be implemented in C# (.NET).

1.4 Preliminary Work and Limitations

The compiler frontend (lexer, parser and semantic checker) and the SmallJ bytecode backend have already been implemented last semester as part of the Compiler Construction course. Hence, this thesis will not cover anything compiler frontend related, but will focus solely on the implementation of the WASM backend.

1.5 Thesis Outline

Chapter 2 provides an analysis of the system context, related technologies and existing WASM compiler architectures. It describes the SmallJ language, the existing compiler and runtime, the SmallJ bytecode intermediate representation, relevant WASM concepts and different architectures of existing WASM compilers. Furthermore, it introduces control flow graphs and lifting algorithms, which form the theoretical foundation for reconstructing structured control flow when compiling bytecode to WebAssembly.

Chapter 3 describes the design of the proposed solution. It presents the overall compiler architecture, the chosen compilation strategy and the integration into the existing SmallJ compiler. In addition, this chapter discusses key design decisions, explicitly compares alternative approaches that were considered and explains the constraints and trade-offs that influenced the final design.

Chapter 4 covers the implementation of the solution. It explains the realization of the compiler pipeline, the stackifier algorithm and the generation of WebAssembly code. Beyond describing the implementation itself, this chapter discusses key implementation aspects and the limitations of the current implementation. In addition, this chapter describes the employed dependencies and technologies as well as the testing workflow used to validate the implementation.

Chapter 5 summarizes the results of the thesis and evaluates them against the initial goals. It discusses the achieved functionality, remaining limitations and the overall suitability of the solution for its intended educational context. The chapter concludes with an outlook on possible future extensions and improvements.

1.6 Summary

This chapter introduced the thesis and outlined the structure of this document. It described the background and motivation for the project, the task and goals of the thesis, the conditions under which the research was conducted, the preliminary work that has already been completed and an outline of the subsequent chapters. The next chapter analyzes the system context, related technologies and existing WASM compiler architectures, which will form the basis for the design and implementation of the solution.

2 Analysis

This analysis covers the SmallJ language and bytecode as well as the existing compiler and runtime. Further, this chapter goes into detail about selected WASM concepts that are of relevance for this thesis. The analysis also features an overview and comparison of the architectures of existing WASM compilers for Java and C#. Finally, the chapter covers control flow graphs and lifting algorithms, which are relevant for the implementation of the WASM backend in case an indirect compilation approach is chosen.

2.1 SmallJ

This section is a more detailed analysis of the SmallJ language, its bytecode intermediate representation, the compiler and the SmallJ runtime.

2.1.1 Language

As a simplified version of Java, SmallJ features similar syntax and semantics as Java, but with some simplifications and restrictions. Those restrictions include the lack of constructors, interfaces, generics, annotations and lambda expressions. The simplifications mainly involve the omission of quality of life and syntactic sugar features, such as no for/for-each loops, no switch statements and no string interpolation. With no constructors, every field of an object is initialized to its default value upon creation. All variables are also default initialized at declaration and there is no way of assigning values to variables at declaration.

SmallJ is object-oriented and supports single inheritance. In contrast to Java, SmallJ has no root class, which means that there is no common super class for all classes. SmallJ does not support static members. The full SmallJ language report is located in appendix B.

2.1.2 Compiler

The current implementation of the compiler is written in C# and compiles SmallJ to SmallJ bytecode. The frontend consists of a lexer, a parser and a semantic checker which are all implemented manually without the use of parser or lexer generators. As soon as these three components have processed the input source code and produced a semantically valid syntax tree with symbol table, the backend takes over. The backend includes the code generator, which is implemented using a visitor pattern that traverses the syntax tree to generate bytecode instructions.

2.1.3 Bytecode

SmallJ bytecode is a simple stack-based format, similar to Java bytecode. The bytecode format consists of instructions, each of which takes a specific number of operands. Control statements, such as loops and conditionals, are implemented using jump instructions. The SmallJ bytecode reference is located in appendix C.

2.1.4 Runtime

The SmallJ runtime is also implemented in C# and relies on the .NET VM. The runtime includes a loader, an interpreter, a garbage collector and a JIT compiler for x86-64 architectures. The loader loads the bytecode into memory, the interpreter runs an interpreter loop that fetches and executes the bytecode instructions, the garbage collector implements a mark-and-sweep algorithm for memory management and the JIT compiler compiles bytecode hot spots to native x86-64 machine code for better performance.

2.2 WebAssembly (WASM)

This section is about WASM and its concepts that are relevant to the present thesis. It covers the basics of WASM, its execution environments, syntax and semantics, functions, function tables, strings, memory and WASM proposals.

2.2.1 The Basics of WebAssembly

Note (WebAssembly Text Format): WebAssembly knows two different formats. The binary format (files with .wasm extension) and the text format (files with .wat extension). The text format is an assembly-like human readable format, that can not be executed directly, but can be parsed to the binary format. The binary format is the only executable form of WASM, but is not human readable. Whenever this thesis refers to WASM code, it is referring to the text format, since it is more human readable and easier to work with for the purpose of this thesis. References to the binary format will always be made explicitly.

Since WASM follows a stack-based virtual machine model, like SmallJ bytecode, it does not differ significantly from it. A stack-based virtual machine is a virtual machine that uses a stack to hold intermediate values during the execution of a program. All instructions operate on the values on the stack,

which means that there are no registers that the instructions can directly access [6].

Figure 1 depicts an example of a stack-based virtual machine executing an `add` instruction. The `add` instruction pops the top two values from the stack, adds them together and pushes the result back onto the stack.

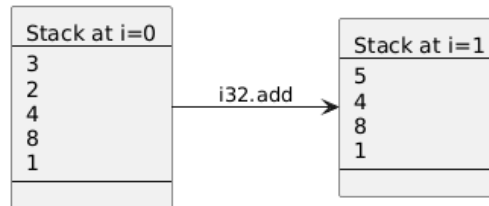


Figure 1: Example of an `add` instruction on a given stack. The left side depicts the stack before the `add` instruction is executed (at instruction index `i=0`). The right side depicts the stack after the `add` instruction is executed (at instruction index `i=1`).

The presence of such a stack, a range of instructions, including arithmetic, logical and control flow instructions, and the ability to manipulate linear memory, as will be covered in section 2.2.6, make WASM a turing complete language that can be used as a compilation target. What makes WASM particularly interesting are the following advantages that it provides [7]:

- **Portability:** WASM is supported by all major web browsers and can also be executed in standalone runtimes
- **Security:** WASM provides a sandboxed execution environment that limits the capabilities of the running code, enhancing security
- **Performance:** WASM’s compact binary format, low-level instruction set and restrictive type system allow for fast and efficient execution

Compared to the current SmallJ runtime, another advantage of compiling to WASM is that it eliminates the “virtual machine within a virtual machine” scenario, where the SmallJ bytecode interpreter is executed within the .NET runtime.

2.2.2 Execution Environments

As already mentioned, WASM can be executed in any modern web browser as well as in standalone runtimes like Wasmtime or Node.js [4, 5]. However, no matter where WASM is executed, it always runs as a sandbox that limits

the capabilities of the running code. One big limitation of this sandbox is that WASM does not have direct access to the underlying operating system, which means that it cannot perform input/output (I/O) operations, such as reading from the file system, making network requests or even printing to the standard output [7].

This closely resembles hardware assembly languages. For example, x86-64 assembly does not have direct access to the operating system either, but instead relies on system calls to perform I/O operations. WASM modules can be embedded in a similar way into a host environment that provides the necessary system calls and APIs for performing I/O operations.

Listing 1 shows an example of how a WASM module can be embedded into a Node.js environment. The JavaScript embedding defines an `env` object and passes it to the WASM instance at instantiation. This `env` object provides functions that can be called from inside the WASM module in a manner similar to system calls.

```
1 const fs = require('fs');
2 const bytes = fs.readFileSync('example.wasm');
3 const env = {
4   halt: () => process.exit(0),
5   writeInt: (n) => console.log(n),
6 };
7 WebAssembly.instantiate(bytes, {env}).then(({instance}) => {
8   instance.exports.main();
9 });
```

Listing 1: JavaScript WASM embedding example.

Inside a WASM module, the provided functions from the embedding environment can then be imported as shown in listing 2 and can be called like regular functions.

```
1 (import "env" "halt" (func $halt))
2 (import "env" "writeInt" (func $writeInt (param i32)))
```

Listing 2: WASM function import example.

The same process can be used to share linear memory between the WASM module and the embedding environment, which will be covered in more detail in section 2.2.6.

Certain WASM instructions might produce a trap under certain conditions. A trap is basically a runtime exception that immediately terminates the execution of the WASM module. WASM can not handle traps internally, but instead relies on the embedding environment to handle them. I.e. if embedded in Node.js, a trap throws a JavaScript exception that can be caught and handled. The classic examples for situations that produce traps are:

- Division by zero
- Out of bounds memory access
- Accessing or calling something that has never been defined
- Stack overflow or underflow

2.2.3 Syntax and Semantics

A WASM compilation unit takes the form of a module, which can contain type declarations, function definitions, globals, tables, memory instances, imports and exports. Imports must be provided by the embedding environment, while exports can be accessed by the embedding environment. Any definition can be exported. Usually, one function is exported as an entry point to the module, but there can be multiple exported definitions [6].

WASM supports two different syntactic styles, the s-expression style and the unpacked expression style. The unpacked expression style is more compact and closer to how a stack-machine would execute the code, while the s-expression style tries to mimic the syntactic structure of higher-level languages [9].

Listing 3 shows an example of the two different syntax styles for a code snippet that adds two numbers together and leaves the result on the stack.

```

1  ;; unpacked expression style
2  local.get $a
3  local.get $b
4  i32.add
5
6  ;; s-expression style
7  (i32.add
8    (local.get $a)
9    (local.get $b)
10 )

```

Listing 3: Comparison of WASM syntax styles.

While the s-expression style is arguably more human readable, the unpacked expression style is easier to generate for a compiler, since it requires less nesting, indentation and parentheses.

WASM also supports two different styles for referencing definitions inside a module, named references and indexed references. Named references use the name of the definition, while indexed references use the index of the definition. A named reference is introduced by a dollar sign followed by a name, for example `$myFunction`. The definition can then be referred to by its

name, while indexed references can only be referred to by their position in the module [6, 8].

The core specification of WASM defines four primitive types. Those are 32-bit and 64-bit integers (`i32` and `i64`) and 32-bit and 64-bit IEEE 754 floating point numbers (`f32` and `f64`). Boolean values are represented as integers, where 0 is false and any non-zero value is true. 32-bit integers are also used to represent addresses in the linear memory and indices in function tables. WASM does not distinguish between signed and unsigned integers, but instead provides different instructions for signed and unsigned operations. Such instructions are suffixed with either `_s` for signed operations or `_u` for unsigned operations, for example `i32.div_s` and `i32.div_u` [6, 7, 8, 10].

Local variables are declared in function headers and can be accessed by index or named reference using the `local.get` and `local.set` instructions. Locals are mutable and default initialized to zero. The indexing of locals includes both the function's parameters and its local variables, where the parameters are indexed first. Therefore, function parameters can be accessed the same way as locals and are also mutable [6].

A module can also declare or import typed global variables, which can be either mutable or immutable. Globals can be accessed from any function in the module by index (separate indexing from locals) or named references using the `global.get` and `global.set` instructions. WASM globals always require constant expressions for initialization [6].

The biggest difference between WASM and other assembly languages is that WASM does not support arbitrary jump instructions and `goto` statements, but instead relies on structured control flow constructs, similar to higher-level languages. One such construct is the `if/else` statement. Listing 4 shows an example of a simple `if` statement (without an `else` block) and an `if` statement with an `else` block. An `if` instruction pops the top value from the stack and checks if it is true (non-zero) or false (zero) in order to decide which block to execute.

```
1 (func $main
2   i32.const 0
3   i32.eqz
4   ;; if last value on stack is 1 (true),
5   ;; execute the if block, otherwise skip to 'end'
6   if
7     i32.const 1
8     call $writeInt
9   end
10
11  i32.const 1
12  i32.eqz
```

```

13  ;; if last value on stack is 1 (true), execute the if block
14  ;; and then skip to 'end', otherwise execute the else block
15  if
16      i32.const 1
17      call $writeInt
18  else
19      i32.const 0
20      call $writeInt
21  end
22 )

```

Listing 4: Example of if/else statement in WASM.

Other control flow constructs are the two related but slightly different `block` and `loop` constructs, which can be used to implement loops. The subtle difference between the two is that a `br` instruction that targets a `block` will jump to the end of the `block`, while a `br` instruction that targets a `loop` will jump to the beginning of the `loop`. The `br_if` instruction is essentially a conditional version of the `br` instruction, which only performs the jump if the top value on the stack is true (non-zero). Both `br` and `br_if` instructions can target any enclosing `block` or `loop` by using a relative index that corresponds to the nesting depth of the target `block` or `loop`. The `block` and `loop` constructs can also be labeled, in which case the `br` and `br_if` instructions can target labels instead of indices. Combining `block` and `loop` allows for the implementation of while loops.

Listing 5 shows an implementation of a complete WASM module that exports a function with a while loop that prints the numbers from 0 to 9. Such a module can be parsed to the binary WASM format by using a tool like Bytecode Alliance’s `wasm-tools` [11] and can then be executed in any embedding environment that provides the function `writeInt`. Using Bytecode Alliance’s `wasm-tools`, the `.wat` file containing the module can be parsed to a binary `.wasm` file by executing `wasm-tools parse example.wat -o example.wasm`. The Node.js code from listing 1 provides a working embedding environment for this example.

```

1  (module
2    (import "env" "writeInt" (func $writeInt (param i32)))
3
4    (func $main (local $x i32)
5      i32.const 0
6      local.set $x
7      block $block
8        loop $loop
9          local.get $x
10         i32.const 10
11         i32.lt_s

```

```

12     i32.eqz
13     ;; branch to the end of $block
14     ;; if $x is not less than 10
15     br_if $block
16     local.get $x
17     call $writeInt
18     local.get $x
19     i32.const 1
20     i32.add
21     local.set $x
22     ;; branch to the beginning of $loop
23     ;; for the next iteration
24     br $loop
25 end
26 end
27 )
28
29 (export "main" (func $main))
30 )

```

Listing 5: Example of a while loop in WASM.

2.2.4 Functions

Functions in WASM conceptually closely resemble functions in other programming languages. They consist of a function header and a function body. The function body contains the code that is executed when the function is called. In the case of WASM, a function body is a sequence of WASM instructions that manipulate the stack and perform computations. The function header is a bit more unique in WASM, since it contains not only the function's name, return type and parameters, but must also declare all local variables that are used in the function body [7].

Listing 6 shows an example of a function header in WASM that declares a return value, two parameters and two local variables. The two parameters are of type `i32` and are named `$a` and `$b`. `result i32` declares that the function returns a value of type `i32`. The two local variables are also of type `i32` and are named `$x` and `$y`. The dollar sign prefixed names can be omitted, in which case the parameters and local variables would have to be accessed by their definition order index. The `return` instruction returns the value on top of the stack as the function's return value, which, in this case, would be the value of the local variable `$x`.

```

1 (func $main (param $a i32) (param $b i32) (result i32)
2   (local $x i32) (local $y i32)
3   ;; function body ...
4   local.get $x

```

```

5   return
6 )

```

Listing 6: Example of a function header in WASM.

2.2.5 Function Tables

In the WASM core specification, “function tables” are just called “tables” but since they are generally only used to hold function references, they are often referred to as function tables. A function table is a contiguous array of function references that can be accessed by index. Function tables are used to implement indirect function calls, which are calls where the target function is not known at compile time, but is instead determined at runtime by an index into the function table. The `call_indirect` instruction is used to perform an indirect function call, where the index of the target function is popped from the stack and the function at that index in the function table is called [6, 10].

Listing 7 shows an example of a function table in WASM. The `table` declaration creates a function table with a size of 2 and a type of `funcref`, which means that it can hold references to functions. The `elem` declaration initializes the function table, starting at index 0, with two functions, `$test1` and `$test2`. The `main` function then performs a `call_indirect` to the function at index 1 in the function table, which would be `$test2`.

```

1 (module
2   (table 2 funcref)
3   (elem (i32.const 0) func $test1 $test2)
4
5   (func $main
6     i32.const 1
7     call_indirect
8     return
9   )
10
11  (func $test1
12    ;; function body ...
13  )
14
15  (func $test2
16    ;; function body ...
17  )
18
19  (export "main" (func $main))
20 )

```

Listing 7: Example of a function table in WASM.

2.2.6 Memory

WASM provides a linear memory model, where a module can declare a contiguous block of memory that can be accessed by all functions in the module. Such a block of memory can also be imported and exported, allowing it to be shared between the WASM module and the embedding environment. This consequently also allows for two independent WASM modules to share memory with each other via the embedding environment. The smallest amount of memory that can be allocated is one page, which the core specification strictly defines as 64KB. The maximum number of pages that can be allocated is 65536, which is a total of 4GB [7, 10].

Listing 8 extends the previous listing 1 by allocating a block of memory with an initial size of 1 page and a maximum size of 4 pages and sharing it with the WASM module. This block of memory can then be imported inside the WASM module with

```
(import "env" "mem" (memory 1 4)).  
  
1 const fs = require('fs');  
2 const bytes = fs.readFileSync('example.wasm');  
3 const memory = new WebAssembly.Memory({  
4   initial: 1,  
5   maximum: 4  
6 });  
7 const env = {  
8   halt: () => process.exit(0),  
9   writeInt: (n) => console.log(n),  
10  mem: memory  
11 };  
12 WebAssembly.instantiate(bytes, {env}).then(({instance}) => {  
13   instance.exports.main();  
14 });
```

Listing 8: Example of memory allocation by the embedding environment.

Inside a module, memory can be allocated with the `memory` declaration and accessed with the `load` and `store` instructions, which take an address as an operand and read or write a value at that address. The `memory.grow` instruction can be used to grow the memory by a specified number of pages up to the maximum size defined in the `memory` declaration. The `memory.size` instruction can be used to get the current size of the memory in pages.

Listing 9 shows an example of different memory operations in WASM. Line 3 creates a global variable named `$pointer` that is initialized to 0 and can be used as a pointer to the linear memory at address 0. Line 4 declares a block of memory with an initial size of 1 page and a maximum size of 4 pages. Line 10 stores the value 42 at the address pointed to by `$pointer`.

Line 14 loads the value at the address pointed to by `$pointer` and prints it using the imported `writeInt` function on line 15. Line 19 grows the memory by 1 page. The `memory.grow` instruction either leaves the value 1 on the stack if the memory was successfully grown or -1 if the memory could not be grown because it would exceed the maximum size. In this example, line 20 just calls `writeInt` on the result of the `memory.grow` instruction to get rid of it on the stack. Finally, line 23 gets the current size of the memory in pages.

```

1 (module
2   (import "env" "writeInt" (func $writeInt (param i32)))
3   (global $pointer i32 (i32.const 0))
4   (memory $mem 1 4)
5
6   (func $main
7     ;; store the value 42 at the location of $pointer
8     global.get $pointer
9     i32.const 42
10    i32.store
11
12    ;; read and print the value at the location of $pointer
13    global.get $pointer
14    i32.load
15    call $writeInt
16
17    ;; grow the memory by 1 page
18    i32.const 1
19    memory.grow
20    call $writeInt
21
22    ;; print the new memory size
23    memory.size
24    call $writeInt
25  )
26
27  (export "main" (func $main))
28 )

```

Listing 9: Example of memory operations.

Executing the module from listing 9 prints the output 42 1 2 with 42 being the value stored at the address pointed to by `$pointer`, 1 being the result of the `memory.grow` instruction and 2 being the new size of the memory in pages. The indexing of memory addresses is byte-based, which means that the address 0 points to the first byte of memory, the address 1 points to the second byte of memory and so on. How many bytes to read or write at a given address is determined by the instruction, for example `i32.load` reads 4 bytes from memory and `i64.store` writes 8 bytes to memory. This means that if the address 0 is used to store an `i32` value, the next available address

for storing another `i32` value would be address 4, since the first `i32` value occupies the bytes at addresses 0, 1, 2 and 3. Unaligned memory access, where the address is not a multiple of the value's byte size, is allowed in WASM but can lead to performance penalties [6, 8].

2.2.7 Strings

WASM does not have a built-in string type. Working with strings in WASM requires a custom implementation, which usually involves allocating memory for the string and storing the string data in that memory. This comes with two important implications that must be considered when working with strings [7].

First, encoding and decoding must be kept track of within the WASM module as well as between the WASM module and the embedding environment. For example, if the WASM module uses UTF-8 encoding, the embedding environment must also use UTF-8 encoding when passing strings to the WASM module and when interpreting strings returned from the WASM module. The module level declaration `data` can be used to initialize memory with data. If a string literal is used in a `data` declaration, it gets encoded as UTF-8 at compile time. As an example, the following declaration encodes the string "Hello, World!" as UTF-8 and initializes the memory starting at address 0 with the resulting bytes:

```
(data (i32.const 0) "Hello, World!").
```

But this only works for UTF-8. For other encodings the raw bytes of the string must be provided in the `data` declaration. For example, to store the string "hello" UTF-16 encoded, a declaration like the following would have to be used:

```
(data (i32.const 0) "\68\00\65\00\6C\00\6C\00\6F\00").
```

The `store` and `load` instructions can then be used to read and write the string data from memory, but for those instructions the encoding and decoding of the string data must always be handled manually, even for UTF-8.

Second, string lengths must be managed manually. This means that strings must either be stored as null-terminated strings or as length-prefixed strings. Null-terminated string data contains a trailing null byte to indicate the end of the string data, while length-prefixed string data contains a leading length field that indicates the length of the string data [7].

Listing 10 shows an example of a WASM module handling strings. The module imports two functions, one for printing null-terminated strings and one for printing length-prefixed strings. The null-terminated string "hello" is stored using a `data` declaration and the length-prefixed string "hi" is stored manually using `store` instructions in the main function. Finally, the main

function calls the imported functions to print both strings.

```
1 (module
2   (import "env" "writeNullTermString"
3     (func $writeNullTermString (param i32))
4   )
5   (import "env" "writeLenPreString"
6     (func $writeLenPreString (param i32))
7   )
8   (import "env" "mem" (memory $mem 1 4))
9
10  ;; store null-terminated string "hello" at address 0
11  (data (i32.const 0) "hello\00")
12
13  (func $main
14    ;; store length-prefix at address 32
15    i32.const 32
16    i32.const 2
17    i32.store8
18
19    ;; store "h" at address 33
20    i32.const 33
21    i32.const 104
22    i32.store8
23
24    ;; store "i" at address 34
25    i32.const 34
26    i32.const 105
27    i32.store8
28
29    ;; print null-terminated string at address 0
30    i32.const 0
31    call $writeNullTermString
32
33    ;; print length-prefixed string at address 32
34    i32.const 32
35    call $writeLenPreString
36  )
37
38  (export "memory" (memory $mem))
39  (export "main" (func $main))
40 )
```

Listing 10: Example of string handling.

Listing 11 extends the previous listing 8 with two functions for reading null-terminated and length-prefixed strings from the WASM module's linear memory and printing them to the console.

```
1 const fs = require('fs');
```

```

2  const bytes = fs.readFileSync('example.wasm');
3  const memory = new WebAssembly.Memory({
4    initial: 1,
5    maximum: 4
6  });
7  const decoder = new TextDecoder('utf-8');
8
9  function readNullTermString(ptr) {
10   const view = new Uint8Array(memory.buffer);
11   let end = ptr;
12
13   while (view[end] !== 0) {
14     end++;
15   }
16
17   return decoder.decode(view.subarray(ptr, end));
18 }
19
20 function readLenPreString(ptr) {
21   const view = new Uint8Array(memory.buffer);
22
23   const len = view[ptr];
24   const start = ptr + 1;
25
26   return decoder.decode(view.subarray(start, start + len));
27 }
28
29 const env = {
30   halt: () => process.exit(0),
31   writeInt: (n) => console.log(n),
32   writeNullTermString: (ptr) => {
33     console.log(readNullTermString(ptr));
34   },
35   writeLenPreString: (ptr) => {
36     console.log(readLenPreString(ptr));
37   },
38   mem: memory
39 };
40
41 WebAssembly.instantiate(bytes, {env}).then(({instance}) => {
42   instance.exports.main();
43 });

```

Listing 11: Example of string handling in embedding environment.

2.2.8 Proposals

By now it has become clear that the WASM core specification is quite minimalistic. It is limited to four numeric types, a linear memory model and the most basic instructions with control flow constructs. This is by design, since the specification is meant to be a minimalistic and stable foundation, while more complex features can be added on top of it through proposals.

WASM proposals are extensions to the specification. They emerge from the open and informal WASM community group, while standardizations of proposals are handled by the formal W3C WASM working group. Each proposal must go through a 6-phase standardization process before it can be added to the specification [12, 13, 14]. Those phases are:

- **Phase 0:** Pre-Proposal – Early discussions within the community group
- **Phase 1:** Feature Proposal – A formal proposal by the community group
- **Phase 2:** Proposed Spec Text Available – Community and working group elaborate on the proposal and a proposed specification text is made available
- **Phase 3:** Implementation Phase – Community and working group work on multiple implementations of the proposal to test it
- **Phase 4:** Standardize the Feature – The working group works on the final standardization of the proposal
- **Phase 5:** The Feature is Standardized – The working group has fully standardized the proposal

Note that for a proposal to be in phase 5 only means that it has been standardized. It does not mean that the proposal is part of the specification. Usually, “Phase 5 Proposals” become part of the specification in the next major release, which moves them from “Phase 5 Proposals” to “Finished Proposals”. “Phase 5 Proposals” are recommended to be supported by all WASM implementations, tools and runtimes, but it is still up to them to decide when to support a given proposal. Sometimes, proposals get tagged as “Phase 4+ Proposals”, which means that they are not fully standardized yet, but still enough to be supported. Some WASM implementations, runtimes and tools already support some “Phase 4+ Proposals” [15].

One of the most starred proposals on GitHub is the GC proposal, which adds native garbage collection support to WASM. Originally, the GC proposal also introduced support for reference types, typed function references and type imports, but those features have since been split into separate proposals. The combination of these proposals now enables basic struct and array instructions without manually having to deal with memory management. Such instructions include `struct.new`, `array.new`, `struct.get`, `array.get`, `struct.set` and `array.set` for allocating and accessing structs and arrays. Further, the proposals also provide declarations like `ref`, `ref null`, `ref null eq` and instructions like `ref.cast` to work with reference types. The proposals introduction of type declarations, like the ones in listing 12, even allow for subtyping [16, 17, 18, 19].

```
1 (type $Base (sub (struct)))
2 (type $Derived (sub $Base (struct)))
```

Listing 12: Example of type declarations.

2.3 An Overview of Existing WASM Compilers

Since WASM has become a popular compilation target for many languages, there is a wide landscape of WASM compilers for different source languages. Some of these compilers follow different strategies and design decisions for compiling the source language to WASM. This section covers different WASM compilers and analyzes their compilation strategies and design decisions. Since SmallJ is a language with a managed runtime, the focus is on WASM compilers for languages with managed runtimes, such as Java and C# and not on languages like C, C++ or Rust.

2.3.1 .NET Blazor WASM

.NET Blazor WASM features two different modes. Interpreted mode and AOT mode.

Interpreted Mode In Interpreted Mode, the .NET Mono runtime is compiled to WASM, but the .NET assembly files are not. The browser can then load and execute the Mono runtime, which can then load and interpret the .NET assembly files [20, 21, 22, 23].

AOT Mode AOT Mode, on the other hand, compiles the .NET assembly files ahead-of-time to WASM, which can be executed in the browser natively. This allows for a more lightweight runtime. Some low-level components of

the runtime, such as the garbage collector, are still necessary. This mode uses .NET Native AOT to compile to WASM. There is also an experimental .NET Native AOT fork that uses LLVM [20, 24, 25].

2.3.2 J2CL

J2CL’s WASM backend compiles Java directly to WASM without going through bytecode first. However, this is most likely due to the fact that J2CL was originally designed to be a Java to JavaScript transpiler and the WASM backend was added later as an experimental feature [26].

2.3.3 TeaVM

TeaVM is an AOT compiler that compiles JVM bytecode to WASM. It relies on other compilers to compile Java, Kotlin or Scala to bytecode first. Originally, TeaVM was designed to decompile Java bytecode to JavaScript and the WASM backend was added later. Due to the lack of documentation regarding the WASM backend, it seems like it is still experimental. According to the WASM backend release notes, it uses the native WASM garbage collector [27, 28, 29].

2.3.4 CheerpJ

CheerpJ follows an interesting and very unique approach in order to compile Java to WASM. It provides an entire JRE, that is written in C++ and compiled to WASM. The JRE includes a JIT compiler that transpiles Java bytecode to JavaScript. The generated JavaScript can then be called from the WASM module [30, 31].

2.3.5 GraalVM

GraalVM or GraalWasm is an embeddable WASM runtime for the JVM. The documentation emphasizes that it is still in the early stages of its development, which is probably the reason for the lack of detailed documentation. According to the documentation, it appears that GraalVM ships an entire WASM JVM implementation including the G1 garbage collector [32, 33].

2.3.6 JWebAssembly

JWebAssembly is a Java bytecode to WASM compiler that does not rely on a JRE. An interesting aspect of JWebAssembly is that it uses the JavaScript

garbage collector. Unfortunately, from looking at the repository, it seems that the project is no longer maintained actively [34].

2.4 Direct vs. Indirect Compilation

Section 2.3 shows that almost all existing WASM compilers follow an indirect compilation approach, where the source language is first compiled to an intermediate language and then from there to WASM. The only exception is J2CL, which compiles Java directly to WASM.

2.4.1 Direct Compilation

The direct compilation approach generates WASM straight from the syntax tree, omitting any intermediate representation (IR). This simplifies and shortens the compilation pipeline and avoids a separate lifting step, as will be described in section 2.9. Its main drawback is reduced modularity. Without an IR, optimizations become more difficult (optimizing directly on the syntax tree is often error-prone and cumbersome) and reuse across languages or backends becomes impractical [35].

2.4.2 Indirect Compilation

Most production compilers use an intermediate representation and for good reason. An IR separates frontend concerns from backend concerns, enabling a more divide-and-conquer approach. Optimizations can be implemented once on the IR to benefit multiple backends and language constructs can be desugared to reduce duplication (for example, ‘for’ and ‘while’ could map to the same IR pattern, although SmallJ has no for-loop currently). Further, the indirect approach leads to a cleaner and more maintainable architecture. Since the backend only needs to conform to the IR, it does not need to be modified on SmallJ language revisions, but only on IR revisions. This makes it easier to evolve the backend to target other platforms. The trade-off is an overall longer pipeline and the need for a lifting step to restore structured control flow. Although the lifting step is complex, it is a well-studied problem that is described in section 2.9.

2.5 AOT Compilation vs. Compiling the Runtime

The analysis of existing WASM compilers in section 2.3 shows that there is no consensus on whether to compile source code ahead-of-time to WASM or to compile an entire runtime to WASM.

2.5.1 AOT Compilation

This approach enables structured control flow to be enforced by WASM itself, offers better performance, simplifies debugging and leads to a cleaner architecture. By emitting native WASM ahead of time, the runtime avoids an interpreter or JIT layer, avoiding startup overhead and an abstraction level during execution. The generated code can be validated statically by the WASM verifier, which improves safety and makes control flow errors easier to detect early. Debugging benefits from a more direct mapping between source constructs and WASM blocks, rather than tracing through an interpreter loop. The main tradeoff is higher compilation time and reduced flexibility at runtime (e.g. no hot reloading), but for the current project this is acceptable.

2.5.2 Compiling the Runtime

Compiling the runtime negates many of the advantages provided by the WASM sandbox. WASM enforces a structured execution model (structured control flow and validated indirect calls) that, for example, forbids arbitrary jumps. Compiling the interpreter/runtime to WASM ensures that the interpreter’s code is validated by the WASM verifier, but the verifier cannot reason about the correctness or safety of the SmallJ bytecode that is executed within it. Conversely, compilation time per program is significantly reduced, as only the interpreter needs to be compiled to WASM, while user programs are executed as bytecode, enabling faster edit-execute iteration. Furthermore, the runtime-based approach does not significantly reduce overall complexity. Instead, complexity is shifted from compile-time to runtime, introducing additional challenges like debugging across multiple abstraction layers. Additionally, this would reintroduce a “virtual machine within a virtual machine” scenario, an issue present in the current implementation that the project explicitly aims to eliminate.

2.6 Using an Existing Backend

Another possibility would be to use an existing compiler backend that compiles to WASM. This would mean compiling SmallJ to an external backend’s IR and then use the external backend to compile to WASM. This would allow using already tried-and-tested infrastructure as well as easily targeting other platforms in the future. The drawbacks of this approach are that it would introduce large external dependencies and would require becoming familiar with the backend’s IR and constraints.

2.6.1 The Dragon: LLVM

LLVM is a modular compiler infrastructure that supports a wide range of platforms and targets, including WASM. It is widely used by languages and compilers such as Clang, Rust and Swift. LLVM features some of the most powerful and mature optimization capabilities. On the other hand, LLVM is a large and complex industry-grade project with a steep learning curve and a slow compilation time. It would also require to get familiar with the LLVM-IR and compiling to it, which is not trivial since it has constraints like SSA form [36].

2.6.2 An Alternative to LLVM: Cranelift

Cranelift is a more lightweight and simpler alternative to LLVM that also has a WASM backend, the learning curve and complexity is significantly lower than LLVM's. The optimization capabilities of Cranelift are not as mature and powerful as LLVM's, but performance is not a critical aspect of this project. Cranelift-IR shares many of LLVM-IR's constraints, most notably SSA form, so targeting it would need similar effort. However, Cranelift simplifies SSA-based representations by using block parameters (bb-parameters) instead of explicit ϕ -nodes and even provides utilities to construct it. Additionally, fewer invariants are enforced by the Cranelift-IR. Overall, Cranelift is the better choice for smaller compiler frontends [37].

2.6.3 Almost WASM: Binaryen

Binaryen is a C++ compiler toolkit and optimization library for WebAssembly only. It provides a compact IR that intentionally maps very closely to WebAssembly and offers a rich set of WASM-specific optimization and transformation passes. Because Binaryen's IR is almost identical to WASM, it is excellent for generating, optimizing and post-processing WASM modules, but it offers little additional abstraction for frontend authors. In other words: Binaryen does not substantially simplify the task of lowering high-level language constructs into WASM and it only supports WASM as a target [38].

2.7 Architectural Comparison

The analysis of the existing compilers and their different architectures shows that there is no universal architecture. The feasible architectures that have emerged from the analysis are:

- Direct AOT compilation: Compiling directly from SmallJ to WASM, without going through bytecode first

- Indirect AOT compilation: Compiling from SmallJ to WASM via the existing bytecode intermediate representation
- Compiling the runtime: Compiling an entire runtime, including an interpreter for the existing bytecode, to WASM and executing user programs as bytecode within the runtime
- Using an external backend: Compiling from SmallJ to an existing backend's IR and then using the external backend (i.e. LLVM or Cranelift) to compile to WASM

Figure 2 provides a visual overview of the different architectures and their compilation pipelines.

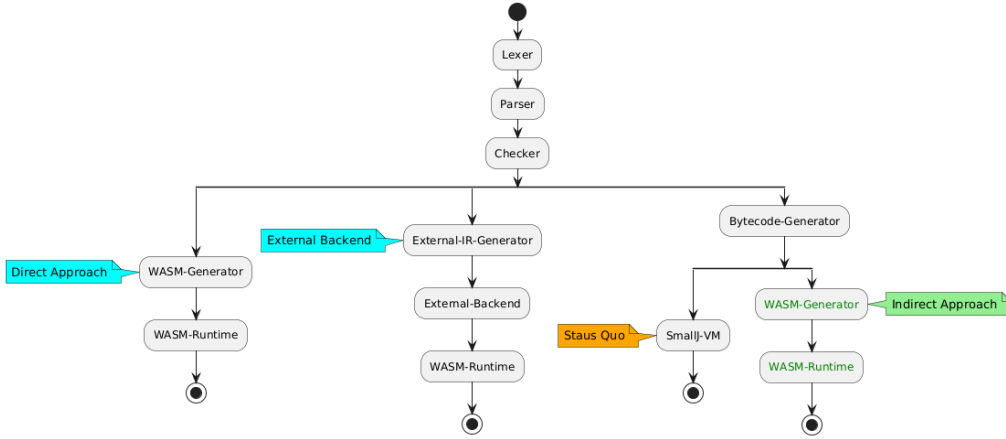


Figure 2: Visualization of different compiler architectures as flow chart.

Table 1 provides a detailed comparison of these architectural approaches across various criteria. Note that the table is not based on empirical data, but rather on theoretical considerations and assumptions that emerged from this analysis. The different architectures are awarded a grade from 1 to 5 for each criterion, with 1 being the best grade and 5 being the worst.

<i>Criteria</i>	<i>Direct AOT</i>	<i>Indirect AOT</i>	<i>Compiled runtime</i>	<i>External backend</i>
<i>Compiler complexity</i>	2	4	2	3
<i>Runtime complexity</i>	3	3	4	3
<i>Execution performance</i>	2	2	4	1
<i>Startup latency</i>	2	2	4	2
<i>Binary size</i>	2	2	4	2
<i>Reuse of existing code</i>	4	3	2	4
<i>Maintainability</i>	3	2	2	3
<i>Optimization potential</i>	3	2	4	1
<i>External dependencies</i>	None (1)	None (1)	None (1)	Backend (5)
<i>Total</i>	22	21	25	24
<i>Advantages</i>	Less pipeline steps, no complex lifting step required, easiest to implement	Cleaner architecture, industry standard	Fast compilation per program, faster edit-execute iteration, minimal changes to existing runtime	Proven and tested infrastructure, easy multi-platform targeting, mature optimization capabilities, active community support
<i>Disadvantages</i>	Less elegant architecture, makes optimization more difficult, WASM-backend tightly coupled to syntax tree	Complex lifting step required, more pipeline steps	Negates WASM sandbox advantages, no bytecode safety verification, VM within VM scenario, complex debugging, shifts complexity to runtime	External dependencies, steep learning curve, must learn backend IR, constraints like SSA form

Table 1: Comparison of compiler architectures. Each architecture is awarded a grade from 1 to 5 per criterion with 1 being the best and 5 the worst grade.

2.8 Control Flow Graphs

In the case of an indirect compilation, a lifting step is required in order to reconstruct the structured control flow that was lost during the compilation to bytecode. Generally, there are two main types of algorithms that can be used for this, relooper and stackifier, which are both described in more detail in section 2.9. Both of these algorithms operate on control flow graphs.

A control flow graph (CFG) is a representation of all paths that might be traversed through a program during its execution. CFGs can be used in compilers to analyze and optimize the control flow of a program.

In essence, a CFG is a directed graph with nodes representing basic blocks of code and edges representing control flow between those blocks. A block of code is a sequence of instructions with no branches, except at the end and no entry points, except at the beginning. Each branch instruction splits a block into multiple blocks, one for each possible branch target [39, 40].

An example of a simple CFG can be seen in figure 3.

Simple CFG

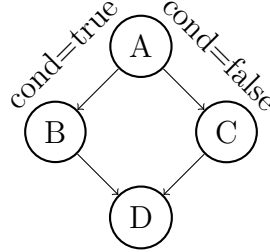


Figure 3: Example of a simple control flow graph for a program with an if-else instruction. Block A contains the program up until the if-else statement. Block B contains the instructions of the true branch (then block). Block C contains the instructions of the false branch (else block). Block D contains the program after the if-else statement.

2.8.1 Dominance

Dominances are relationships between different nodes, or blocks, in a CFG.

Definition 2.8.1 (Dominance). A block M is said to dominate another block N if every entry point to N must go through M first.

The entry block of a CFG, which is the block where execution starts, dominates all other blocks in the CFG. The stackifier algorithm relies on dominance relationships to determine how to structure the control flow, as it uses dominators to determine how to order basic blocks and introduce nested block scopes [40, p. 656].

2.8.2 Edges

Edges in a CFG represent the control flow between blocks. All edges can be classified as different types. In the context of control flow reconstruction, the important types are forward edges, retreating edges and back edges. Figure 4 depicts an example CFG with different edge types [40, p. 702].

Note (DFS-based ancestor/descendant): Ancestor/Descendant relationships within flow graphs are defined with depth-first search (DFS) ordering. If a node A comes before node B in the DFS order, A is considered an ancestor of B and B is considered a descendant of A.

Definition 2.8.2 (Forward edge). A forward edge is an edge whose target block is a descendant of the source block. In contrast to the usual graph-theory definition, a CFG forward edge is also allowed to target a direct descendant of the source block.

Definition 2.8.3 (Retreating edge). A retreating edge is an edge whose target block is an ancestor of the source block. Retreating edges indicate loops, but they do not determine how many entry points a loop may have.

Definition 2.8.4 (Back edge). A back edge is a retreating edge whose destination block dominates the source block. Back edges indicate single-entry loops. Back-edges are a subset of retreating edges.

Example showcasing the different edge types

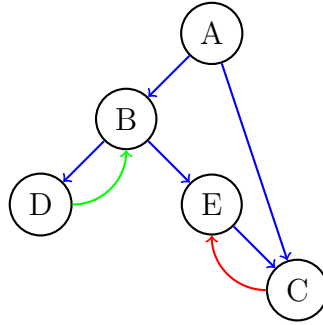


Figure 4: Example of a CFG with different edge types. Forward edges are depicted in blue, retreating edges in red and back edges in green.

2.8.3 Reducibility

CFGs can be classified as reducible or irreducible.

Definition 2.8.5 (Reducibility). A CFG is said to be reducible if all its retreating edges are back edges. All CFGs that are not reducible are considered to be irreducible.

This is equivalent to saying that all loops in a reducible CFG only have a single entry point, which is the loop header. Irreducible CFGs usually result from aggressive optimizations, but certain language constructs, such as manual goto statements, exceptions and jumps, can also produce them. Irreducible CFGs can be converted to reducible CFGs by applying transformations, such as node duplication or introducing a dispatcher block [40, p. 684][42, 44].

As an example, figure 5 depicts an irreducible CFG. The irreducibility of this CFG comes from the fact, that the $B \rightarrow D \rightarrow E \rightarrow B$ loop can be entered both via $A \rightarrow B$ and via $A \rightarrow C \rightarrow D$. Having multiple different entry points to the same loop makes the CFG irreducible. Two common ways to eliminate this irreducibility are node duplication or inserting a dispatcher block.

Example CFG

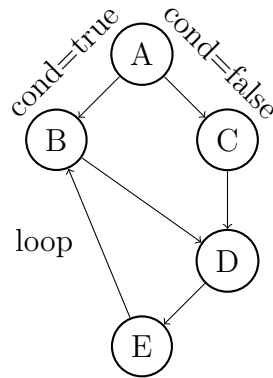


Figure 5: Example of a CFG with an irreducible loop.

Node Duplication This approach duplicates the nodes that provide extra loop entries such that all branches enter the loop at the same entry point. The resulting reducible CFG is depicted in figure 6. No extra runtime cost is introduced, but code size could increase exponentially in the worst case.

Node Duplication

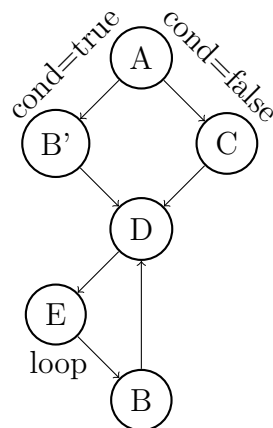


Figure 6: Transformed version of the CFG in figure 5 into a reducible CFG using node duplication.

Dispatcher Block Using a dispatcher block introduces a single-entry block called a dispatcher that acts as a state machine and routes control to the appropriate successor. The resulting reducible CFG is depicted in figure 7. Code size increases only slightly (linear in the worst case), but a small runtime cost is introduced.

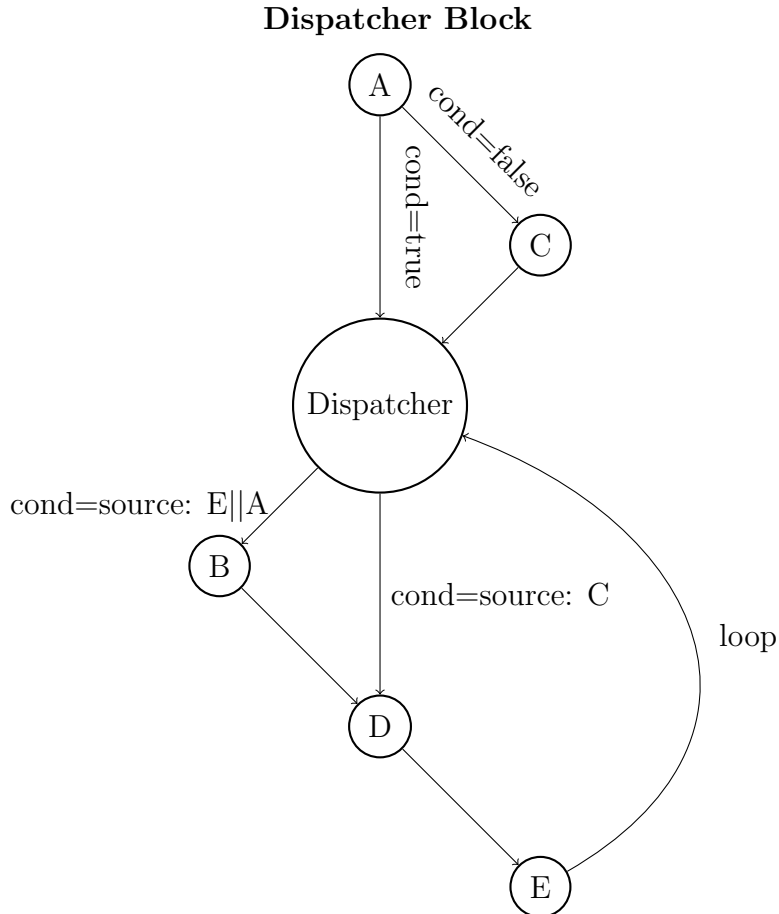


Figure 7: Transformed version of the CFG in figure 5 into a reducible CFG using a dispatcher block.

2.8.4 Relevance of Reducibility

Currently, SmallJ does not have any language constructs that can produce irreducible CFGs and the current compiler does not perform any optimizations that can produce them either, so irreducible CFGs are currently not a concern for this project. The only way for SmallJ to produce irreducible CFGs would be for someone to write SmallJ bytecode manually, but it is generally discouraged from writing any kind of intermediate code manually.

Still, it is important to be aware of irreducible CFGs and the fact that they can arise in future versions of SmallJ, implying that the architectural decisions should be made with this possibility in mind.

2.9 Lifting Algorithms

Lifting algorithms reconstruct structured control flow from CFGs. The reason an indirect compilation requires a lifting step is the mismatch between SmallJ bytecode’s `GOTO`, `IF_TRUE` and `IF_FALSE` jump instructions and WASM’s control flow constructs (if/else and block/loop). In essence, such lifting algorithms are a decompilation technique that tries to recover high-level structured control flow from low-level unstructured jumps. The two main approaches are relooper and stackifier, which reflect different philosophies.

2.9.1 Relooper

Relooper is a more straightforward, greedy approach that takes a CFG as input and reconstructs structured control flow by emitting nested loop/if constructs and explicit labels for otherwise unstructured edges. It was popularized in the Emscripten toolchain as a practical way to lift LLVM CFGs to a structured target language without arbitrary gotos. For WASM backends, this addresses the core mismatch between arbitrary CFG jumps and WASM’s structured control flow constructs. Strengths of relooper-style algorithms are their robustness, i.e. they work on any CFG, and their guarantee of producing valid structured output. However, relooper-style algorithms can produce unnatural, harder-to-optimize code, often emitting labels and a dispatcher (a switch-based state machine), which typically yields slower, larger and less elegant output than stackifier-style algorithms. [41, 42, 43].

2.9.2 Stackifier

Stackifier-style algorithms are based on ordering basic blocks and introducing nested block scopes using node dominators so that forward and backward edges can be represented via scoped branch targets. In practice, these approaches are closely related to classical control flow structuring and region-based reconstruction techniques from decompilation literature. Compared to relooper-style shaping, stackifier-style constructions often produce more direct structured code, that matches WASM’s structured control flow, on reducible CFGs. However, irreducible CFGs must be transformed into reducible ones before stackifier-style algorithms can be applied, since they

only operate on reducible CFGs. LLVM, as an example, uses reloop-style dispatcher blocks to transform irreducible CFGs into reducible ones before applying a stackifier-style algorithm to restore structured control flow [42, 43, 44].

2.9.3 Comparison

Compared to stackifier, reloop can be seen as a more primitive approach. It requires no pre-pass to eliminate irreducibility and works on any CFG, but it typically emits suboptimal and slower code. Stackifier, on the other hand, is a more refined algorithm that yields cleaner and potentially faster code. If irreducibility elimination is done in a dispatcher block style, the approach is somewhat similar to reloop. Thus, depending on the irreducibility elimination strategy, there can be an overlap between stackifier and reloop [44, 45].

2.9.4 Example

Figure 8 shows an example control flow graph that highlights a case where a reloop-style lifting produces a dispatcher-based state machine while a stackifier-style lifting can emit more natural nested control constructs [42].

Confusing CFG for Reloop

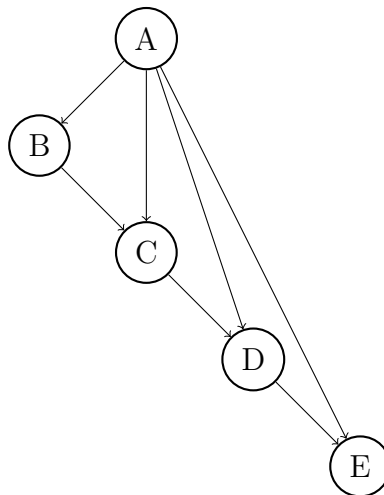


Figure 8: Example of a control flow graph that causes a reloop-style algorithm to fall back to a dispatcher-based state machine. The depicted CFG models a switch-case language construct [42].

Listing 13 shows the pseudo-code generated by a reloop-style algorithm

for the CFG in figure 8. The algorithm emits a dispatcher-based state machine using a label variable and switch statements to route control flow, which is less natural and more verbose than the original switch-case structure.

```

1 label = 0
2
3 switch A()
4   case 0
5     B()
6     label = 3
7     break
8
9   case 1
10    label = 3
11    break
12
13  case 2
14    label = 4
15    break
16
17 if label equals 3
18   C()
19   label = 4
20
21 if label equals 4
22   D()
23
24 E()

```

Listing 13: Pseudo-code generated by reloop [42].

Listing 14 shows the pseudo-code generated by a stackifier-style algorithm for the same CFG of figure 8. The algorithm emits more natural nested control constructs, closely matching the original switch-case structure of the CFG, without the need for an explicit dispatcher or label variable.

```

1 d:do{
2   e:do{
3     c:do{
4       switch A()
5         case 0
6           break
7
8         case 1
9           break c
10
11        case 2
12          break d
13
14        default

```

```

15         break e
16
17     B()
18 } while 0
19     C()
20 } while 0
21     D()
22 } while 0
23 E()

```

Listing 14: Pseudo-code generated by stackifier [42].

2.10 Summary

This chapter gave an introduction to the existing SmallJ compiler and WebAssembly. The SmallJ section provided an overview of the SmallJ language, its bytecode and the existing compiler architecture. The WebAssembly section provided an introduction to WebAssembly, its design and its execution model as well as various WASM concepts that are of relevance for this project.

The analysis of existing WASM compilers revealed that there is no consensus on the best architecture for a WASM compiler for a language with a managed runtime, such as SmallJ. The main architectural decisions that need to be made are whether to compile directly from SmallJ to WASM or to go through the existing bytecode intermediate representation, whether to compile user programs ahead-of-time to WASM or to compile an entire runtime to WASM and whether to use an existing backend or not. The different architectures have different advantages and disadvantages, but the indirect AOT compilation approach seems to be the best fit for this project, as it offers a cleaner architecture, better optimization potential and better maintainability, while the drawbacks of a more complex lifting step and a longer compilation pipeline are acceptable for this project.

Furthermore, the analysis of control flow graphs and lifting algorithms revealed that the lifting step is a complex but well-studied problem with different approaches, such as relooper and stackifier, that have different trade-offs.

3 Design

This chapter describes the design of the WASM backend for the SmallJ compiler. The findings from chapter 2 are used to derive the architectural decisions and implementation concepts that form the basis of the new backend.

The chapter is structured as follows. First, the most important architectural decisions are discussed and justified. Afterwards, the compiler pipeline and the responsibilities of its individual stages are presented. Finally, the chapter describes the selected WASM embedding environments and their role in executing the generated modules.

3.1 Architectural Decisions

Based on the analysis in chapter 2 the decision was made, that the indirect AOT compilation approach is the most suitable architecture for the SmallJ compiler. The IR-based design best balances implementation effort and long-term maintainability. It enables reuse and centralized optimizations (implement once on the IR and benefit all backends), reduces potential duplication of language patterns and produces a cleaner separation of concerns. Resulting in a more modular and maintainable architecture. This choice does have costs. The pipeline is longer and a lifting step is required to recover structured control flow, which implies implementing a complex lifting algorithm. Although external backends offer mature optimization infrastructure, they come at the cost of additional integration effort and introduce external dependencies. Binaryen, while excellent for WASM generation and post processing, provides an IR that is almost identical to WASM and therefore does not meaningfully reduce the burden of lowering to WASM. Additionally, Binaryen only supports WASM as a target.

Considering the project scope and goals, indirect AOT via an IR provides the best compromise between implementation complexity, optimization potential and maintainability.

3.1.1 Lifting Step

The decision to compile indirectly via bytecode implies that a lifting step is required. In consideration of the analysis of lifting algorithms in section 2.9, stackifier emerged as the best option for the SmallJ compiler. Since SmallJ, in its current state, does not have any language constructs that can produce irreducible CFGs, stackifier's limitation to only handle reducible CFGs is not a concern. For the same reason, reloader's main advantage of being more

robust to irreducible CFGs does not provide any benefit while it's tendency to produce less readable code would only introduce drawbacks.

As stated in section 2.8.4, the only way to produce irreducible CFGs would be to write SmallJ bytecode manually, which is not recommended, so the decision was made to not support code that generates irreducible CFGs. Due to this decision, there is no need to implement transformation logic for irreducible CFGs, which significantly reduces the implementation effort for the lifting step. However, the CFGs should still be analyzed for irreducibility and an error should be thrown if an irreducible CFG is encountered, to prevent the generation of invalid WASM code.

3.1.2 Produces Value

In WebAssembly, if-else constructs that leave a return value on the stack must be explicitly annotated with a result type, i.e. `if (result i32)`. This annotation enforces that both the then and else branches leave exactly one value of the specified type on the stack.

The logic for determining whether an if-else block leaves a value on the stack is simple, but it is also somewhat fragile and may fail for manually written bytecode. It relies on the way SmallJ currently lowers logical operators such as `&&` and `||` into bytecode. In the current compiler pipeline, a value-producing if-else only arises when such a condition is first lowered into an if-else construct and then followed by the conditional block. Figure 9 illustrates this lowering step. Although the approach is not fully general, it is sufficient for the intended use case and provides a pragmatic solution for compiling valid SmallJ programs correctly to WASM. The implementation is described more in detail within section 4.2.2.

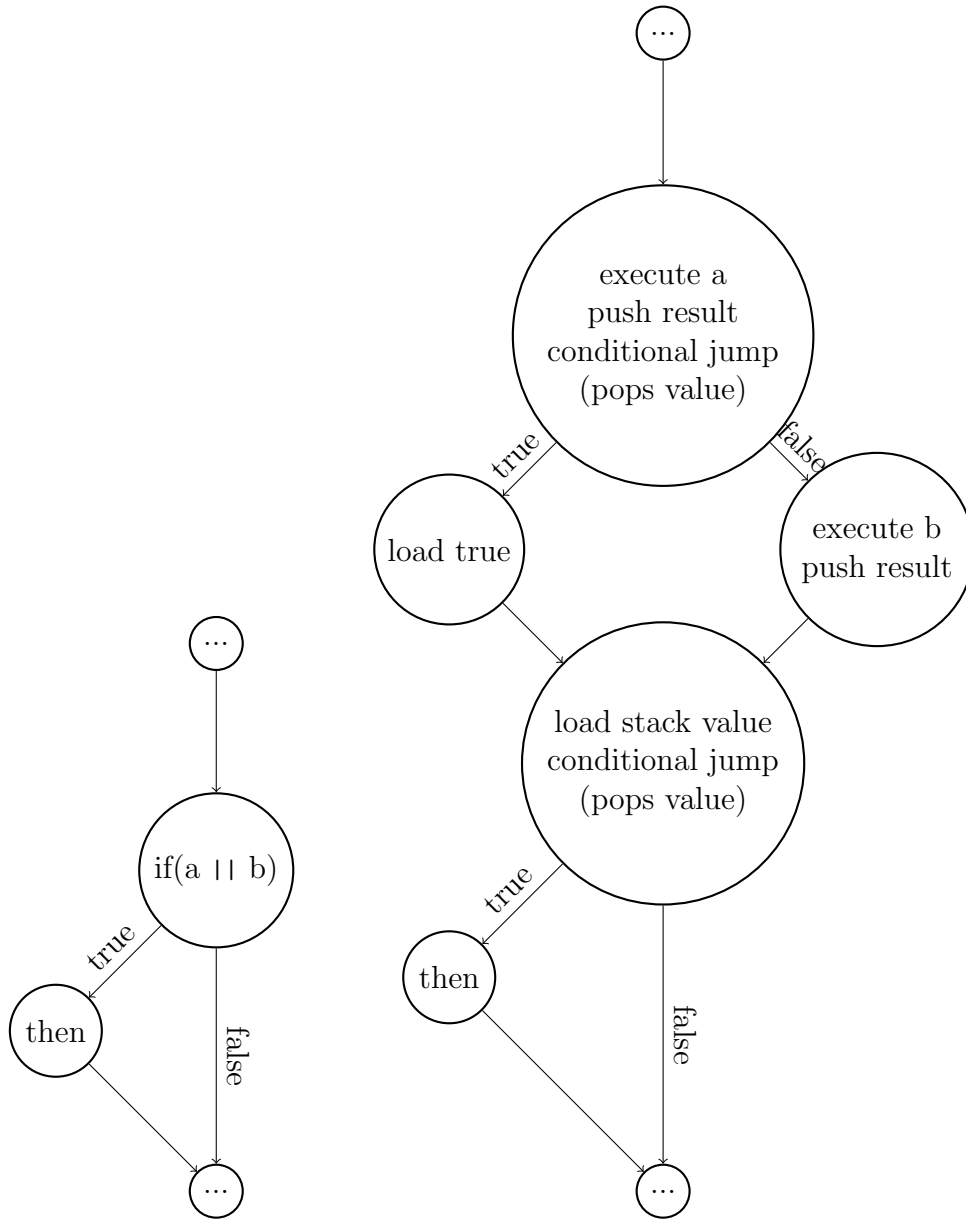


Figure 9: Example showing how a SmallJ condition containing a logical or (`||`) is lowered to bytecode. The graph on the left shows the original SmallJ control flow, while the graph on the right shows the corresponding simplified flow after lowering to bytecode. The conditions `a` and `b` can represent any expressions that evaluate to a boolean value, for example `i == 1`.

3.1.3 SmallJ to WASM Mapping of OO-Features

The analysis of the WASM GC proposal in section 2.2.8 revealed that the GC proposal provides features that can be used to implement object-oriented features of SmallJ in a natural way. The GC proposal not only introduces the concept of struct types, which can be used to represent SmallJ classes, but also arrays. WASM's subtypable structs even allow to model class hierarchies and inheritance. Additionally, the GC proposal provides support for reference types and managed memory allocation as well as garbage collection.

For dynamic dispatch, WASM's function tables, as described in section 2.2.5, are a natural fit. They allow indirect function calls based on an index, which can be used to model a vTable.

Considering these features, the decision was made to adopt the WASM GC proposal to implement SmallJ's object-oriented features in a natural and efficient way. The most promising approach is to represent SmallJ classes as WASM struct types, map SmallJ arrays directly to WASM arrays, use subtypable structs to model class inheritance and model a global vTable as a function table for dynamic dispatch. As in SmallJ bytecode, the first argument of each method will be a reference to the receiver object, which will be used to access the fields of the struct. In order to perform indirect calls, structs will have to hold indices into the function table.

This design allows for a natural implementation with little to no overhead. In contrast, implementing a custom object model on top of WASM, without using the GC proposal, would be possible but would require significantly more effort and would introduce more overhead. Considering the available time and resources, a custom implementation would most likely just be a worse implementation of an existing solution.

3.1.4 String Handling

The takeaway from section 2.2.7 is that strings in WASM are far from trivial to implement compared to higher level languages. The lack of a built-in string type requires careful consideration of how to represent strings. The following two designs were considered:

Manual Allocation and Management of String Data This design constructs a pool of strings at compile-time. Each unique string literal is collected from the SmallJ source code and stored once in the pool. This already allows for interning of string literals, where identical string literals share the same memory location and therefore save memory. All strings from this pool can then be encoded accordingly (i.e. Unicode) and stored in the

WASM module’s linear memory either as null-terminated or length-prefixed strings. The WASM code can then reference these strings by their offset in the linear memory.

This design shifts the majority of the complexity to compile-time, resulting in low runtime overhead since the strings are pre-allocated and can be accessed efficiently via their offsets. Since the WASM module can export its linear memory to the embedding environment, this design also allows the embedding environment to place dynamically generated strings (i.e. ones read by the `readString()` instruction) into the linear memory at runtime. However, this implementation has the significant drawback of requiring manual memory management for strings. This would, in turn, require a custom garbage collection solution for the strings in the linear memory, which would significantly increase the implementation effort and complexity of the runtime. The WASM-GC proposal cannot be utilized for this, since it firstly only supports GC structs and arrays and secondly operates on a separate heap and not on the linear memory.

Wrapping Strings in WASM-GC Structs In order to facilitate the WASM-GC proposal strings could be represented as GC structs that hold the string length in an `i32` field and the string data as a reference to a GC managed byte array. This design would also build a pool of unique strings and encode them at compile-time, again allowing for interning of string literals. However, instead of storing the strings from the pool in the linear memory, each string would generate a WASM data segment and a global null reference that will later be set to the string struct. A data segment is a sequence of bytes that is stored in the module binary and can be used at runtime. It is not stored in the modules linear memory, it is like a statically embedded “data blob” that can be accessed via a global reference. At startup, an initialization function would then have to allocate a string struct for each data segment and set the global reference to point to the allocated struct.

Since the GC heap can only be managed by the GC itself, I/O operations and dynamically generated strings would require the linear memory to be used as a buffer for sharing string data between the embedding environment and the GC managed heap. This, in addition to the initialization function, would require two more helper functions. One for creating string structs from string data in the linear memory and one for copying the data from a string struct into the linear memory. This design benefits from the GC proposal, which handles string memory management, eliminating the need for custom memory management and garbage collection of strings. On the other hand, it introduces more runtime overhead due to the need for additional helper

functions and an extra level of indirection when accessing string data.

Design Decision Considering the trade-offs, the decision was made to continue with the second design, which wraps strings in WASM-GC structs. Although it introduces some runtime overhead, the use of the GC proposal still significantly reduces the implementation effort and complexity of the runtime compared to implementing a custom memory management solution.

3.2 Architectural Overview

Figure 10 shows the system context of the SmallJ compiler. Since the entire compiler is seen as one software system, the addition of the WASM backend does not change anything about the system context.

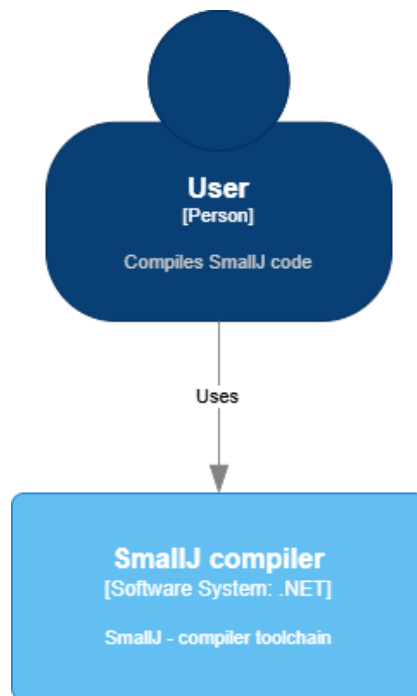


Figure 10: System context view of the SmallJ compiler.

The container view, as shown in figure 11, shows two new additions with a WASM runtime and an external dependency to the Wasmtime nuget package. The WASM code generation is integrated into the existing compiler container and does, therefore, not appear as a separate container in the diagram.

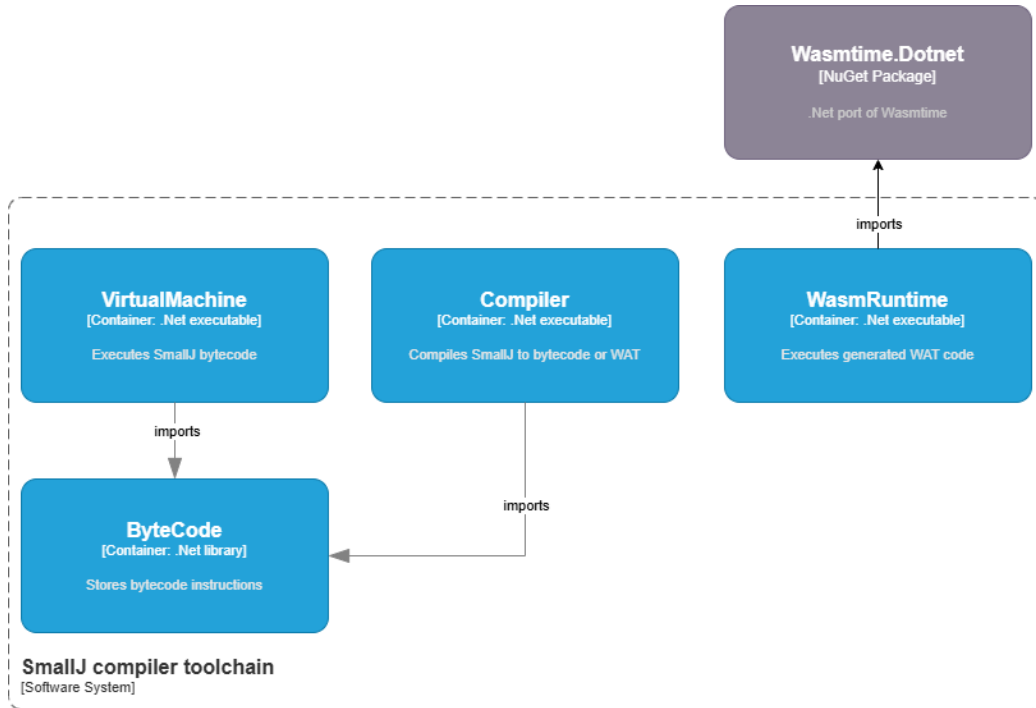


Figure 11: Container view of the SmallJ compiler.

The compiler component view in figure 12 shows the addition of the WASM code generation component as the new `WasmGeneration` component. The compilation pipeline goes from top to bottom, with each component being called by the `Program` class. `Lexing`, `Parsing`, `Checking` and `Generation` remain unchanged, with the last one generating SmallJ bytecode. For WASM generation, the new `WasmGeneration` component is called after the bytecode generation and takes the generated bytecode as input to compile it to WASM. Further, the diagram introduces the new `ControlFlow` component, which gets called by the `WasmGeneration` component in order to restore the structured control flow that was lost during compilation to bytecode. The `ControlFlow` component can optionally also be called by `Program` class to render CFGs for the bytecode methods.

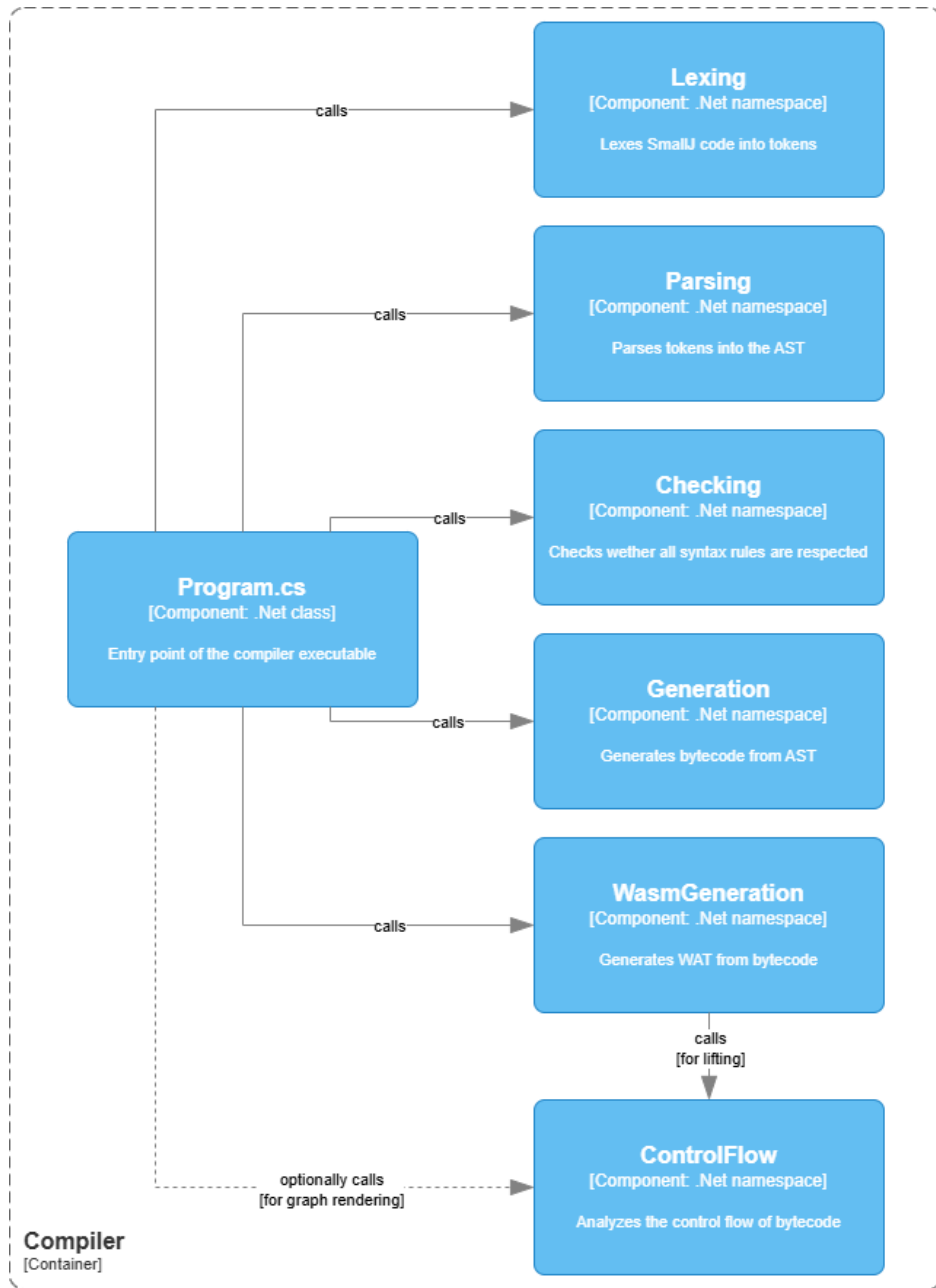


Figure 12: Compiler component view of the SmallJ compiler.

Figure 13 shows the runtime component view, with the `WasmRuntime` being the main entrypoint for executing the generated WASM code. The `WasmRuntime` component calls a `WasmRunner` component, which instantiates a WASM runtime using the external `Wasmtime` dependency that executes the generated WASM module. The `WasmEmbeddingException` component is a base

exception that gets extended by a `WasmInputException` and a `WasmHaltException` in order to enable custom and graceful error handling for the embedding environment. Exceptions of this type get caught and handled appropriately by the `WasmRuntime` component. Because the current version of the Wasmtime nuget package does not expose the API for enabling the WASM-GC proposal, the `WasmtimeProposalEnabler` component is required to set the `use GC proposal` config flag manually via reflection.

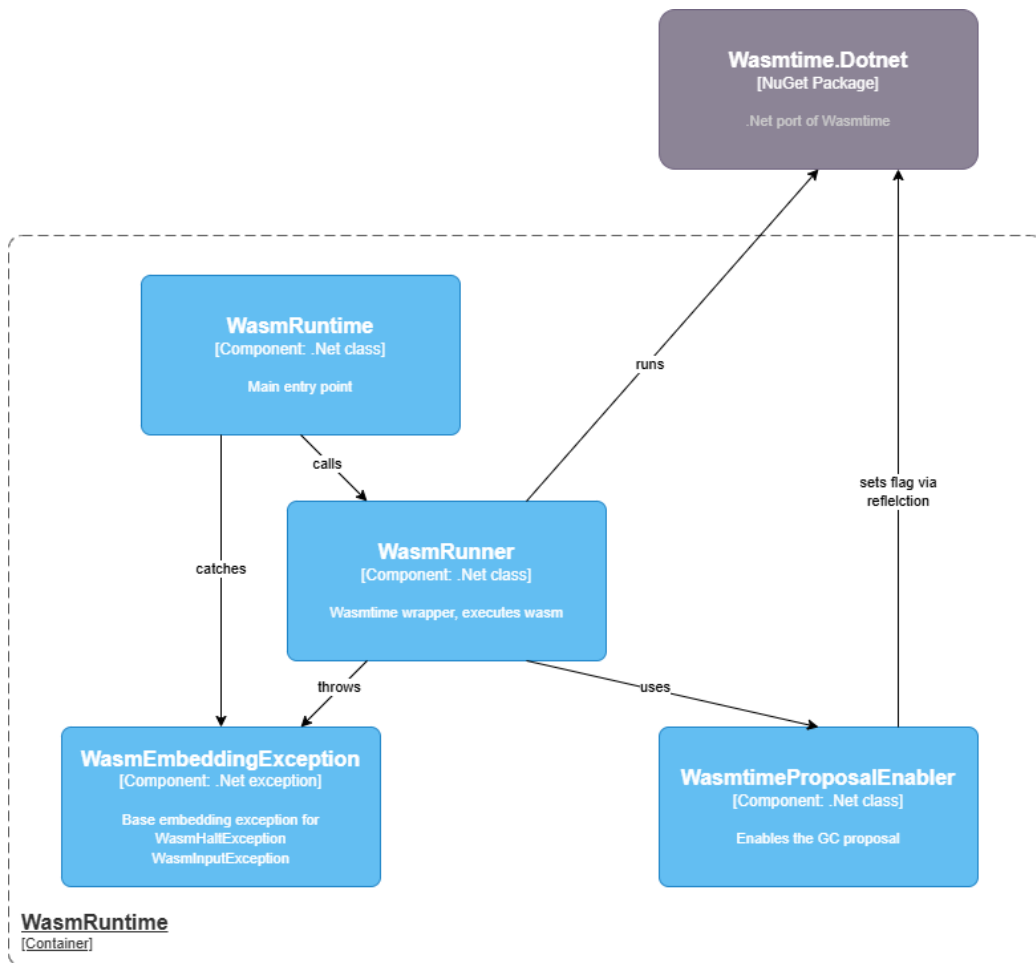


Figure 13: Runtime component view.

3.3 Compiler Pipeline

Figure 14 depicts the overall pipeline of the SmallJ compiler, including the new WASM generation. Up until the bytecode generation, the pipeline remains the same as the original SmallJ compiler pipeline. The WASM gener-

ation is a separate pipeline step, that follows the bytecode generation. This WASM step takes the generated bytecode as input and compiles it to WASM. It contains all the logic for compiling bytecode to WASM, including:

- A type mapper to map SmallJ types to WASM types
- Logic to build a vTable and an according WASM function table for dynamic dispatch
- Emitting logic to emit WASM instructions based on the bytecode and the lifted control flow
- Module building logic to build a complete WASM module with the emitted instructions, including handling of imports, exports and other module components

The control flow analysis and lifting logic is not part of the core pipeline, because it is a separate concern that is only relevant for the WASM generation step, while being an optional feature for bytecode generation in order to render CFGs of the bytecode methods.

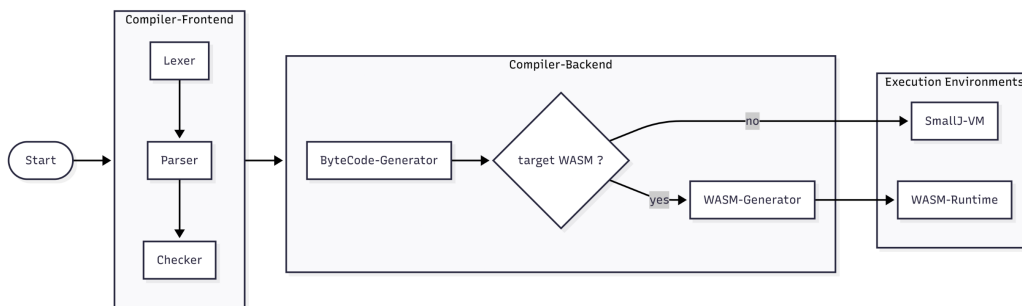


Figure 14: Visualization of the compiler pipeline with the WASM backend.

This pipeline design allows for both compilation to WASM and bytecode in a modular way. The WASM generation step can only be invoked, when WASM generation is desired, otherwise the pipeline can just omit the last step, resulting in a normal bytecode compilation.

3.4 WASM Embedding

As mentioned in section 2.2.2, there are multiple embedding environments where the generated WASM code can be executed, such as web browsers, Node.js or standalone runtimes like Wasmtime. In order to allow fast and easy development and manual end-2-end testing of the WASM backend, the decision was made to implement a minimal Node.js based embedding at first, similar to the one in listing 1. Once the WASM backend is in a stable state, a .NET embedding using the Wasmtime nuget package [46] can be implemented in order to allow better integration into the SmallJ ecosystem, easier automated end-2-end testing and a more streamlined execution workflow by eliminating the need to manually parse the `.wat` file into a `.wasm` binary.

Another rationale behind this decision is that a Node.js embedding is a single and very lightweight JavaScript file, at the cost of being more fragile and brittle, especially when it comes to the implementation of the `readString` and `writeString` I/O operations, which require the embedding to perform some manual memory management on the WASM string buffer linear memory. A .NET embedding, on the other hand, is a separate .NET project with a larger implementation footprint, but also the more robust and maintainable embedding in the long run since the I/O operations can be implemented in C# rather than in JavaScript.

Having both a Node.js and a .NET embedding allows for a good balance between fast development at first and a robust embedding in the long run. Additionally, it provides flexibility for users of the SmallJ compiler, as they can choose the embedding environment that best suits their preferences.

3.5 Summary

This chapter outlined the architectural decisions and the reasoning behind them. To balance implementation effort, optimization potential and maintainability, the WASM backend is implemented as an indirect AOT compiler via bytecode. For the lifting step, the stackifier algorithm is used, as it is sufficient for the intended use case and provides a pragmatic solution for correctly compiling valid SmallJ to WASM. The implementation of object-oriented features relies heavily on the WASM-GC proposal, which allows for a natural and efficient mapping of SmallJ's OO features to WASM. For string handling, the decision was made to wrap strings in WASM-GC structs, which reduces the implementation effort at the cost of some runtime overhead. A modular compiler pipeline supports both the generation of bytecode and WASM. The WASM embedding is implemented in Node.js for rapid initial development and also in .NET to achieve better integration and maintainability.

4 Implementation

This chapter presents implementation details that are of particular relevance and provides a comprehensive overview of the dependencies and technologies employed. It discusses the control flow graph (CFG) construction, the implementation of the lifting step, the generation of WebAssembly code, as well as the realization of object-oriented features and string handling. Furthermore, the chapter describes the utilized dependencies and technologies, explaining both their intended purpose and the rationale for their selection. Finally, the testing workflow is outlined, detailing the testing strategy, employed tools and validation procedures used to ensure the correctness and reliability of the implementation.

4.1 Control Flow Graph Construction

To lift the unstructured flow of the bytecode into structured form again, that conforms to WASM, a CFG-analysis is required. As a first step, the bytecode is partitioned into basic blocks, in order to later construct the CFG. A basic block is a sequence of instructions that can only be entered at its first instruction and exited at its last instruction.

The construction begins by identifying all entry points of basic blocks. Instruction index 0 is always considered an entry point, as it represents the program entry.

In addition, all jump targets of branch instructions (such as `GOTO`, `IF_TRUE` and `IF_FALSE`) are marked as entry points. Furthermore, the instruction immediately following a branch instruction is also marked as an entry point, provided the end of the bytecode has not been reached.

Once all entry points have been determined, the basic blocks are built. Each block starts at an entry point and ends immediately before the next entry point or at the final instruction of the bytecode.

Finally, the basic blocks are connected by resolving branch targets and adding edges to the CFG. Two kinds of control flow edges are distinguished: fallthrough edges and conditional edges. Conditional edges originate from `IF_TRUE` and `IF_FALSE` instructions and are explicitly classified according to the instruction type. This classification is required in a later lifting step to correctly determine the corresponding then and else blocks within an if-else (see section 4.2.2). Fallthrough edges arise either from an unconditional `GOTO` instruction or from the implicit control flow when a block is split due to an entry point occurring in the middle of an instruction sequence. Figure 15 depicts an example of a control flow graph created from source code.

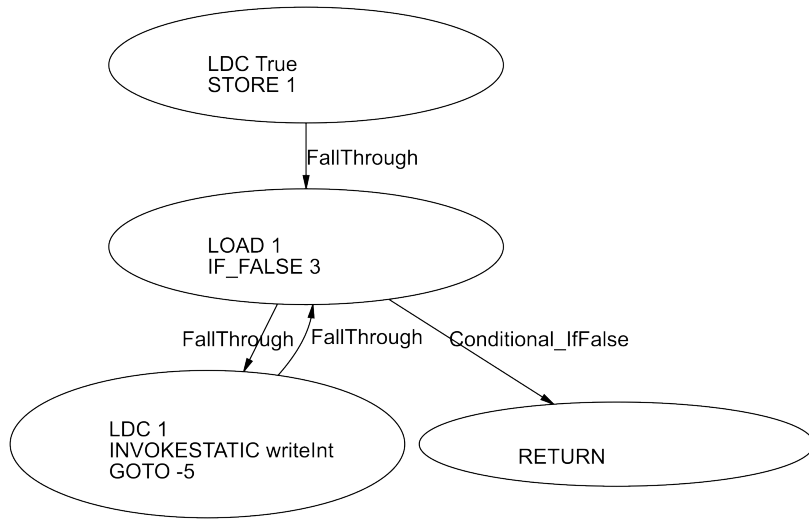


Figure 15: Sample CFG generated from bytecode, corresponding to a method containing a simple loop. The graph was generated using the custom-implemented graph rendering tooling, which produces SVG-based visualizations directly from the original source code.

4.2 Stackifier Implementation

This section details the implementation of the stackifier algorithm, which transforms an arbitrary Control Flow Graph into a structured representation suitable for direct translation into WebAssembly control flow constructs. To achieve this, the implementation operates in two distinct phases: an initial preprocessing and validation step, followed by the recursive application of the core stackifier algorithm.

4.2.1 Preprocessing

Before the stackifier algorithm can be applied to a control flow graph, some preprocessing steps are necessary to ensure that the graph is in the correct form.

Removal of Unreachable Nodes First, all unreachable nodes are removed from the graph. Unreachable nodes confuse the algorithms used afterwards and can lead to incorrect results. This is done by performing a depth-first search starting from the entry node and marking all reachable nodes. Then all unmarked nodes are removed from the graph.

Dominator Tree Construction The Lengauer-Tarjan[47] algorithm computes the dominator tree in two main phases. First, a depth-first search initializes DFS numbering. Second, the algorithm processes nodes in reverse DFS order. For each node v , it computes a semidominator (the predecessor with minimal DFS number) by evaluating all incoming edges using a link-eval union-find data structure with path compression. During this phase, tentative immediate dominators are stored in buckets associated with semidominator nodes. A final pass then corrects these dominators by propagating through the semidominator chain. The resulting dominator tree exposes nesting and loop headers which the stackifier algorithm needs to recover structured control flow. An example of a CFG and its dominator tree is shown in figure 16.

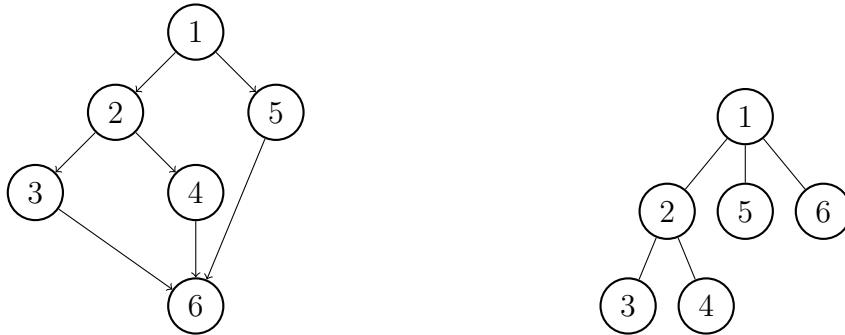


Figure 16: An example control flow graph (left) and its corresponding dominator tree (right).

Backedge Tagging Backedges are tagged beforehand so the stackifier can easily identify them. Specifically, every edge whose target node dominates its source node is marked as a backedge.

Irreducibility Check After preprocessing, the CFG is checked for irreducibility, two examples are shown in figure 17. If we remove all backedges from a CFG and it still contains cycles, the CFG is irreducible. To do this a depth-first search is performed, where all backedges are ignored. Nodes currently on the recursion stack are stored in a set. If the traversal reaches a node that is already in this set, the CFG contains a cycle. Irreducible CFGs are rejected and not processed any further. [40, p. 664]

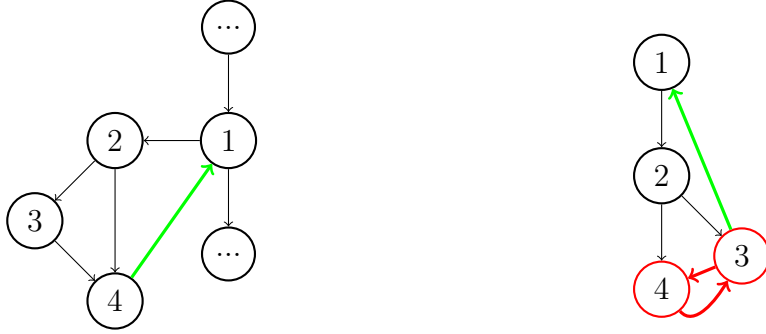


Figure 17: The CFG on the left contains no cycles after the backedges, marked in green, have been removed and is therefore reducible. In contrast, the CFG on the right still contains a cycle after the backedges have been removed and is therefore irreducible.

Loop Detection Using the previously tagged backedges, natural loops are identified. Each backedge target is treated as a loop header and for every backedge a reverse traversal over incoming edges starts at the backedge-source and gathers predecessors until the header is reached. The resulting loop region contains the header and the collected body nodes (excluding the header itself).

4.2.2 The Algorithm

The stackifier algorithm is then applied to the preprocessed CFG. The algorithm starts at the root of the dominator tree and recursively processes its child nodes. It checks whether the current node is a loop header, an if condition, or an if-else condition. If it is none of these, the algorithm assumes it is a basic block and places it into a sequence node. Structured nodes are built from the found components. Then, the child nodes that belong to the current scope are processed recursively, for example, the loop body. Nodes that are not within the current scope are deferred and processed later. After a structured node is built, the algorithm processes any trailing dominator children and deferred nodes and combines them into a sequence node, with the constructed node at the start and the trailing nodes appended afterwards. Nodes with more than two outgoing targets are not supported.

Child Ordering The children of a dominator-tree node are processed in a heuristic order. This ordering causes the nodes within the current scope to be processed first, before handling nodes that are more difficult to place, resulting in a more natural output of structured control flow. Additionally,

fewer nodes need to be deferred and revisited later, which simplifies the recursive construction.

In detail: First, the children that are direct CFG successors of the current node are considered (preserving the order of the parent’s outgoing edges). From that subset, children that have only one incoming edge within the CFG are placed at the beginning, backedges are ignored. The remaining CFG-successor children are appended afterwards. At last, all remaining dominator children are appended.

Loops The natural loops within the graph are calculated in advance, so the algorithm only checks if the current node is within the list of loop-headers. First, the loop header is converted into a structured node. The current loop region is pushed onto a stack of loop regions so the algorithm can track which nodes are within the current loop scope and which are not. Then the nodes within the loop body are processed recursively. If no body nodes are found, the loop is treated as an illegal self-loop and the algorithm aborts. Nodes not within the current loop scope are deferred and processed later. The rendered loop body and the header are then returned as a loop node. Finally, the current loop region is removed from the stack again.

Ifs A node is considered an if-condition when it has exactly two outgoing successors. Further, the two successor branches have to satisfy one of the following two conditions:

- One branch reaches the other without traversing any backedges (forming a triangle-shape).
- Exactly one branch is completely dominated by the condition node. Meaning, all nodes reachable from that branch’s entry are dominated by the condition node.

In the triangle case, the branch that can reach the other without backedges is taken as the then-block. In the second case, the dominated branch is treated as the then-block. In both cases, the then-block must be directly dominated by the condition node to be considered a valid if-condition. Examples for both cases are shown in figure 18. The then-block is processed recursively and the structured node (condition plus then-branch) is returned.

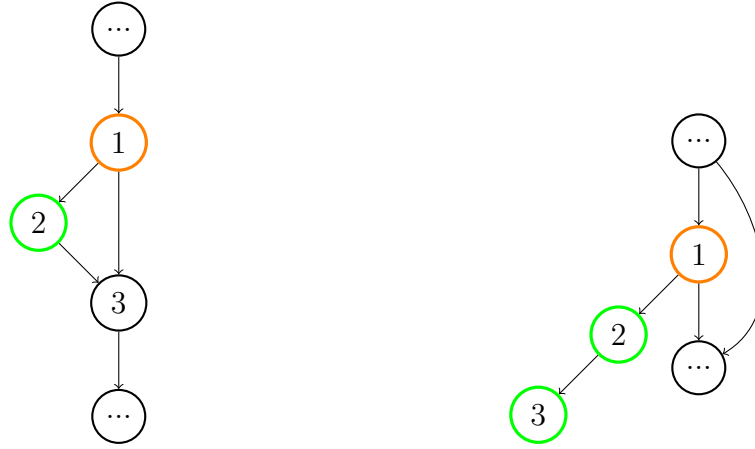


Figure 18: The two possible scenarios for an if, triangle-shape (left) and single dominated branch (right). Conditional blocks are marked orange and blocks within then-scope are marked green.

If-Else For a node to be treated as an if-else condition, it must have exactly two outgoing branches: one fall-through branch and one conditional branch. Both branch targets must be direct dominator-tree children of the condition node. Inside a loop scope, both branches must either stay within the current loop region or be completely dominated by the condition node. Meaning all nodes reachable from that branch's entry are dominated by the condition node. If the conditional instruction is `IF_TRUE`, its target becomes the then-block and the fall-through becomes the else-block. For `IF_FALSE`, this mapping is swapped. Examples are shown in figure 19. The produces-value flag is set when either branch ends with an `LDC` instruction, optionally followed by a `GOTO`. This marks that the resulting if-else node leaves a value on the stack, which is required during WebAssembly generation. This case appears for combined SmallJ conditions (conditions using logical AND/OR), which are lowered to an if-else shape that leaves a boolean value for a subsequent conditional step. Finally, the then and else blocks are processed recursively and the resulting structured node (condition plus branches) is returned.

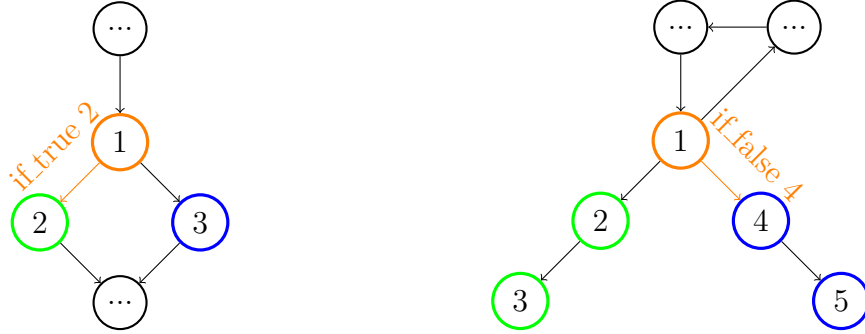


Figure 19: The two possible scenarios for an if-else, diamond-shape (left) or two separate, dominated branches (right). Conditional blocks are marked orange, blocks within then-scope are marked green and blocks within else-scope are marked blue.

4.2.3 Summary

This section highlighted the implementation of the stackifier algorithm for reconstructing structured control flow. The process combines a comprehensive preprocessing phase, which establishes block reachability, dominance relationships, loop structures and graph reducibility, with a recursive phase that applies the core stackifier algorithm to restructure the control flow. By identifying loops, conditional constructs and sequential nodes based on dominance and control flow structures, the algorithm produces a structured intermediate representation. This structure output serves as the direct foundation for the subsequent WASM code generation phase. Notably, any control flow graphs that violate the algorithm’s formulated structural constraints are explicitly rejected.

4.3 WASM Generation

This section first gives an overview of the backend’s structure as well as the implementation of specific OO features and later goes into more detail about how WASM code is generated from a SmallJ bytecode assembly and the structured control flow nodes.

The entire backend is structured into Layouts, Builders and Emitters. Layouts make up the structure of a WASM module, containing all the necessary information for generating corresponding WASM code. Builders are responsible for constructing Layouts from the SmallJ bytecode assembly, with each Layout having a corresponding Builder. Emitters generate WASM code from the Layouts. This Layout-Builder-Emitter architecture enables a clear separation of concerns, where the construction of the module’s structure is

decoupled from the actual code generation, allowing for better maintainability and extensibility. Further, it allows a hierarchical implementation, where multiple Layouts can compose one larger Layout and Builders and Emitters can delegate work to corresponding Builders and Emitters of sub-Layouts.

The entry point for WASM generation is the `WasmGenerator` class, which first calls the stackifier implementation to lift the control flow graph of each defined method to structured control flow nodes. Next, a `vTable` is built for each class, which is used to resolve method calls during code generation. After that, a `ModuleBuilder` is called to build a `ModuleLayout`, which finally is emitted into WASM code by the `ModuleEmitter`.

4.3.1 Layouts

Layouts are implemented as lightweight data structures containing all the information an Emitter needs in order to emit WASM code. At the implementation level, a `ModuleLayout` functions as the root Layout, storing references to each section-specific sub-Layout, which in turn, may reference further sub-Layouts. Figure 20 depicts the hierarchical structure of all Layouts and their relationships.

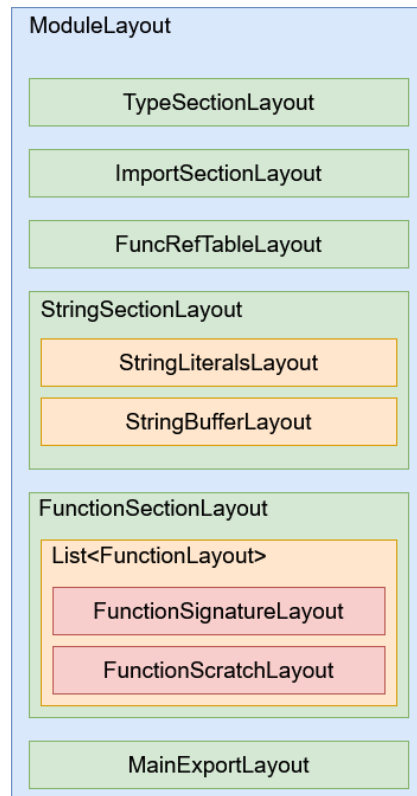


Figure 20: The layout hierarchy that makes up a complete WASM module. Each box represents a Layout. Layouts with matching colors appear on the same level of the hierarchy.

4.3.2 Builders

Builders are responsible for building Layouts based on the SmallJ bytecode assembly. The main entry point for building all Layouts is a `ModuleBuilder`. It delegates the building of the different sub-Layouts to their corresponding Builders, which are responsible for building their respective Layouts. Figure 21 shows a sequence diagram of the building process.

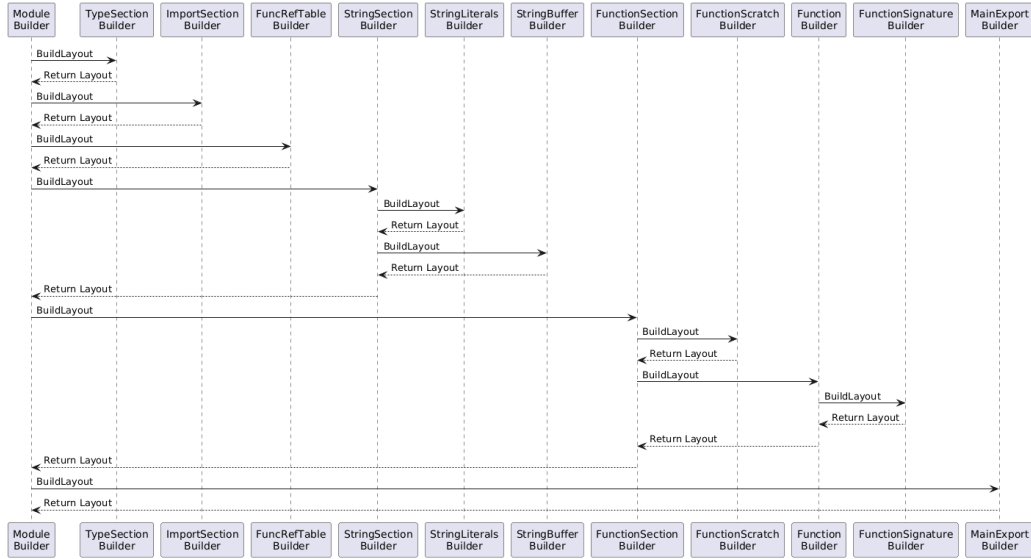


Figure 21: Sequence diagram showing the building process and the delegation of work from the `ModuleBuilder` to the different sub-Layout Builders in order to build a complete `ModuleLayout`.

All Builders are implemented using a static factory method pattern, where one static `BuildLayout` factory method is publicly exposed. This static factory method calls a private constructor and subsequently a private `Build[LayoutName]Layout` instance method. As a result, the Builder's public API is limited to the static factory method, while the actual implementation of the creation logic is encapsulated within the private instance method. This gives the Builder a static interface, while it is internally implemented as a stateful instance.

4.3.3 Emitters

Emitters are implemented as static classes, each providing an `Emit` method that takes a `Layout` and a `Writer` object and emits the corresponding WASM code. The `Writer` object is a `StringBuilder` wrapper that provides helper methods for emitting WASM code, such as indentation management, new line handling and methods for emitting specific WASM instructions. As in the Builders, the `ModuleEmitter` serves as the main entry point for code emission by delegating the emission of each `Layout` to its corresponding Emitter. A `Writer` object is passed down from the `ModuleEmitter` to the various sub-Emitters, such that a final `GetContent` call on the `Writer` returns the complete emitted WASM code for the entire module.

Most Emitters operate on dedicated Layouts, however there are exceptions. Rather than emitting from their own Layouts, `StructuredNodeEmitter` and `InstructionEmitter` generate WASM code directly from the structured control flow nodes of the lifted CFG and the original SmallJ bytecode instructions stored in the `FunctionLayout`. Figure 22 illustrates the emission process and the delegation structure among the Emitters.

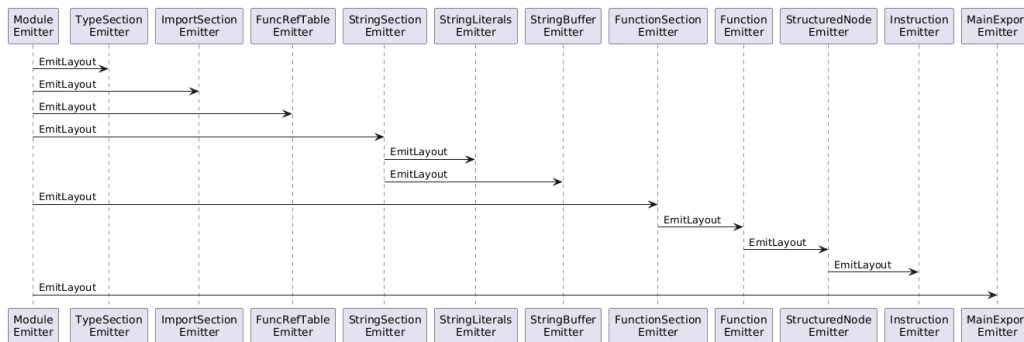


Figure 22: Sequence diagram showing the emission process, `ModuleEmitter` delegates work to the different sub-Layout Emitters in order to emit a complete `ModuleLayout`.

4.4 Object-Oriented Features

The implementation of all OO features relies heavily on the WASM-GC proposal and the reference types proposal. SmallJ classes, including inheritance chains, are directly mapped to WASM-GC subtypable struct types. Each field of a SmallJ class is mapped directly to a corresponding field in the associated WASM-GC struct type. The possible field types are `i32` for primitive SmallJ types, such as `int` and `boolean` and `ref` for SmallJ reference types, such as `string` and user-defined classes. Each method in a SmallJ class maps to an `i32` field in the corresponding WASM-GC struct type. Those fields are used to store indices into a global function reference (`funcref`) table, which, as will be described in section 4.4.1 in more detail, enables virtual method dispatch.

Inheritance is modeled using WASM-GC subtyping. Derived SmallJ classes are mapped to subtype structs of the corresponding basetype struct. Consequently, a reference to a subtype struct can be used whenever a reference to its basetype struct is expected. `InstanceOf` checks and casts map to the WASM-GC `ref.test` and `ref.cast` instructions respectively, which check and cast references based on the defined subtyping hierarchy.

SmallJ arrays are mapped to WASM-GC array types, with the element type, again, being an `i32` for primitive types and `ref` for reference types.

4.4.1 Virtual Method Dispatch

The vTable building process follows the same principles as in the SmallJ bytecode runtime, where the vTable of a class is built based on the vTable of its parent class, with overridden methods replacing the corresponding entries and new methods being appended at the end. A `vTableBuilder` is responsible for building all vTables based on the SmallJ assembly before the generation of WASM code starts. Once all vTables are built, the Builder returns a `vTableResolver`, which is used during code generation to determine the correct vTable for a given class and the correct vTable slot for a given method.

The WASM-GC proposal does not provide built-in support for virtual method dispatch. Therefore, it is implemented explicitly using a global `funcref` table and dispatch vectors embedded in the WASM-GC structs.

Every WASM-GC struct type contains one `i32` field per method in the corresponding SmallJ class. These fields are in the same order as the SmallJ class's vTable slots, meaning field position 1 corresponds to vTable slot 1, field position 2 corresponds to vTable slot 2 and so on.

Additionally, there is one global `funcref` table that contains references to all defined functions in the WASM module. The `i32` fields in the WASM-GC structs, that correspond to vTable slots, then get populated with indices into this global `funcref` table at creation of struct instances. This encodes the virtual method bindings directly into the struct and allows to indirectly call functions through the `funcref` table by looking up the index in the struct field corresponding to the vTable slot.

As an example, let `s` be the slot index in a vTable and `t` the index of a target method in the `funcref` table. At code generation time, `s` is known because it corresponds to the vTable slot of the method being called, which can be looked up in the `vTableResolver`. Since the `funcref` table Layout is built before WASM code generation, `t` can also be looked up at code generation time. WASM code generation for instantiating a struct can then store value `t` at field position `s` in the new struct. Then, a virtual call can generate WASM code to read value `t` from the field at position `s` in the struct, which can then be used as an index to call the correct method implementation through an indirect call via the `funcref` table.

Although this implementation enables correct virtual method resolution, it alone is not sufficient for correctly implementing virtual method calls. This is due to the mismatch, that in SmallJ, when a derived class overrides a virtual method from a base class, the overriding method implicitly assumes

that the `this` reference is of the derived type, which allows it to access fields that are only defined in the derived class. However, since WASM is strictly typed, an indirect function call through a `funcref` table not only expects the index of the target function, but also its type. This, in combination with the fact that the `this` reference is always passed as the first argument and therefore is part of the functions type, means that all functions overriding the same corresponding vTable slot must have the same type. Consequently, because WASM does not support any kind of covariant parameter types, all overriding functions are forced to accept a `this` reference of the function's base type, which prevents them from accessing fields that are only defined in the derived class.

The following four phases of the code generation process solve this issue:

- **Enforcing the base type:** When generating the signature of an overriding function, the `this` parameter is set to the base type of the corresponding vTable slot, which is looked up using the `VTableResolver`.
- **Allocating a this-alias:** When the class containing the overriding method differs from the type of the `this` reference in the function signature, a dedicated scratch local variable of the base type is allocated, which serves as an alias for the `this` reference.
- **Receiver cast prologue:** At the beginning of the function body, a prologue is emitted that down-casts the `this` reference to the derived type using a `ref.cast` instruction and stores the result in the previously allocated `this-alias`.
- **Redirecting receiver loads:** If the `this-alias` scratch variable exists, all loads of parameter index 0, which is always the `this` reference, are redirected to the `this-alias` variable, which has the correct derived type and allows access to all fields of the derived class.

The combined implementation of the `funcref` table indirect calls and the `this`-aliasing finally allows to correctly perform virtual method calls, while still adhering to the strict typing requirements of WASM. Further, the implementation isolates all the complexity related to virtual method dispatch such that the rest of the code generation does not have to be concerned about these details.

4.5 Strings

Strings are implemented as a custom WASM-GC struct type, with one `i32` field for the string length and one `ref` field for the string data, which is stored

as a reference to a byte array. Every generated WASM module contains a “string section”, which is made up of the following components:

- One data segment containing the UTF-8 encoded string data for each unique string literal used in the source code
- One global reference for each unique string literal, which is, at first, set to a null reference
- A `$_init_strings` helper function, that gets called at module startup and:
 - Constructs a WASM-GC struct for each string data segment
 - Sets the corresponding global reference to point to the corresponding string struct
 - Drops the original data segment since it is no longer needed
- A `$_string_buffer` memory declaration, used as a buffer for sharing string data between the embedding environment and the WASM module
- A `$_string_to_buffer` helper function for copying strings from the according string struct into the shared memory buffer
- A `$_buffer_to_string` helper function for creating new string structs from string data in the shared memory buffer

The string buffer memory is dynamically sized to always be large enough to hold the largest string that may be shared between the embedding environment and the WASM module. During compilation, the byte length of the largest string literal in the source code is calculated and the string buffer memory is initialized with the next multiple of the WASM page size (64KB) that can fit this largest string. The embedding environment reads the initial size of the string buffer memory when the module is loaded. If the embedding environment then wants to share a larger string, it can grow the string buffer memory at runtime in order to fit the new largest string.

The memory buffer only supports sharing one string at a time, since all copy operations to and from the buffer start at memory offset 0 and overwrite the previous content. However, the only operations that require the string buffer are the static `readString` and `writeString` functions, which only operate on one string at a time. Therefore, this implementation is sufficient for the required string handling functionality, while allowing a simple implementation that only requires string lengths to be passed around instead of lengths and offsets.

4.6 Specific Builder-Layout-Emitter Implementations

This section details the specific implementation of all Builder-Layout-Emitter components.

4.6.1 Module

`ModuleLayout` is essentially a container for all the Layouts of the next lower level of the hierarchy, while `ModuleBuilder` and `ModuleEmitter` can be seen as delegators for the building and emitting of these Layouts respectively.

4.6.2 TypeSection

A `TypeSectionLayout` contains a list of `TypeSectionDefinition` records and a `HeapTypeIndexResolver`. A `TypeSectionDefinition` is an abstract record extended by:

- A `CharArrayTypeDefinition` record for character arrays used in string handling
- A `StringTypeDefinition` record used as the struct type for the custom string implementation
- A `ClassTypeDefinition` record for each SmallJ class in the entire SmallJ assembly
- An `ArrayTypeDefinition` record for each SmallJ array type in the entire SmallJ assembly
- A `FunctionTypeDefinition` record for each method in the entire SmallJ assembly

A `TypeSectionBuilder` collects all types from a SmallJ assembly and builds the corresponding records, populating them with the necessary information for generating the WASM type definitions. Each collected type is assigned a key that is based on the type's identity and is mapped to the WASM type definition's index in a `HeapTypeIndexResolver`. This allows to look up the correct type indices throughout the building and emitting process.

The `TypeSectionEmitter` emits WASM code for each `TypeSectionDefinition` record in the list based on the records specific type.

4.6.3 ImportSection

An `ImportSectionLayout` contains a list of `ImportSectionDefinition` records, which represent the functions that are imported from the embedding environment. This List is populated by a `ImportSectionBuilder` and emitted by the `ImportSectionEmitter`. The currently used import functions are the same as in the existing SmallJ bytecode runtime, which are:

- `halt` for aborting execution of the program
- `writeInt` for writing an integer to the embedding environment's output stream
- `writeString` for writing a string to the embedding environment's output stream
- `readInt` for reading an integer from the embedding environment's input stream
- `readString` for reading a string from the embedding environment's input stream

4.6.4 FuncRefTable

A `FuncRefTableLayout` contains a list of all SmallJ bytecode methods and a dictionary mapping each method's identity to an index alongside a helper method for looking up these indices. A `FuncRefTableBuilder` collects all methods from a SmallJ bytecode assembly and builds a `FuncRefTableLayout`. The final WASM table instruction is emitted by the `FuncRefTableEmitter` according to the Layout.

4.6.5 StringSection

A `StringSectionLayout` is, again, a container for a `StringLiteralsLayout` and a `StringBufferLayout`, with a `StringSectionBuilder` and the `StringSectionEmitter` being delegators for building and emitting these sub-Layouts respectively.

A `StringLiteralsLayout` contains a list of `StringLiteralDefinition` records and the byte length of the largest string literal defined in the source code. A `StringLiteralDefinition` record contains the string literal's value as defined in the SmallJ source code, a name for the global reference that will point to the string struct and a name for the data segment that will contain the string's UTF-8 encoded byte data. All unique string literals are collected from the SmallJ assembly by a `StringLiteralsBuilder`, which creates

a `StringLiteralDefinition` for each of them and calculates the byte length of the largest string literal. By only collecting unique string literals, the Builder already interns them. Based on the Layout, the `StringLiteralsEmitter` emits the global string references and the data segments for the string literals, as well as a helper function for initializing the string structs and setting the global references at module startup.

A `StringBufferBuilder` calculates the minimum number of memory pages required for the string buffer's linear memory, based on the maximum byte length of any string literal computed by the `StringLiteralsBuilder` and builds a `StringBufferLayout` containing that value. The final string buffer memory declaration is emitted by the `StringBufferEmitter`, as well as helper functions for copying strings to and from the string buffer.

4.6.6 FunctionSection

A `FunctionSectionLayout` is a collection of `FunctionLayouts`, which, in turn, contain a `FunctionSignatureLayout` and a `FunctionScratchLayout` alongside other information such as the function's original bytecode and the lifted structured control flow nodes. A `FunctionSectionBuilder` builds a `FunctionLayout` for each method in a SmallJ assembly by delegating to a `FunctionScratchBuilder` and a `FunctionBuilder`. A `FunctionScratchBuilder` analyzes a function's bytecode for instructions that require scratch variables and returns an according `FunctionScratchLayout`. Scratch variables are necessary when a bytecode instruction's WASM equivalent presents some kind of mismatch. This is the case whenever a piece of information is implicitly present in the SmallJ bytecode runtime, but needs to be made explicit in WASM or whenever the order of operands on the stack differs between SmallJ bytecode and WASM. With the current SmallJ bytecode revision, scratch variables for WASM code generation are required in exactly four scenarios. `FunctionScratchLayout` defines a common record type that captures the information required to generate scratch variables and from which four specialized record variants are derived, one for each scenario:

- **VirtualCallScratch:** For storing the receiver's and the call arguments' local variable indices
- **ThisAliasScratch:** For storing the casted this-alias local variable index
- **PutFieldScratch:** For storing the local variable index of the value to be put into a field

- **NewArrayScratch:** For reordering the operands on the stack, such that they adhere to the order expected by the WASM `array.new` instruction

A `FunctionBuilder` delegates to a `FunctionSignatureBuilder` in order to build a `FunctionSignatureLayout` and returns that Layout alongside the bytecode as a `FunctionLayout`. A `FunctionSignatureBuilder` collects method signature information from a SmallJ method, such as name, parameters and return type, and builds a `FunctionSignatureLayout` containing that information.

In order to emit all WASM functions, the `FunctionSectionEmitter` iterates over all `FunctionLayouts` and calls the `FunctionEmitter` for each of them, which first emits the function signature based on the `FunctionSignatureLayout`, then checks if a `ThisAliasScratch` is present in the `FunctionScratchLayout` and, if so, emits the receiver cast prologue. Finally, the `FunctionEmitter` calls the `StructuredNodeEmitter` with the `InstructionEmitter`'s `Emit` method as delegate argument in order to emit the function body according to the lifted control flow and the original bytecode.

4.6.7 MainExport

The final step of the building process is to build a `MainExportLayout`, which defines a wrapper function for the main function. This Layout is built by a `MainExportBuilder`, with the `MainExportEmitter` responsible for emitting it. This wrapper function is necessary, because the WASM struct corresponding to the SmallJ Main class needs to be instantiated before its main function can be called, which is exactly what the main wrapper function does.

4.7 Changes to the Bytecode Generator

One big mismatch between SmallJ bytecode and WASM is that the SmallJ bytecode runtime can resolve types at runtime by looking up descriptors, while WASM requires all types to be resolved at compile time. To bridge this gap, the bytecode generator was extended to perform additional type resolution and inference during code generation. Specifically, this lead to the following two changes to the bytecode generator:

- The `ALOAD` and `ASTORE` instructions now include the array element type as an operand, which is used to emit the correct WASM instruction for loading from and storing to an array.
- The `CMPEQ` and `CMPNE` instructions were extended to include whether a reference or value comparison should be performed, which is necessary to emit the correct WASM instruction for comparisons.

Since these instructions used to have null operands, the changes have no effect on the existing SmallJ runtime. It simply ignores the operands and behaves as before, while the WASM backend can use the additional information for correct code generation.

4.8 WASM Runtime

The Node.js runtime is a small JavaScript file that loads the generated WASM module, registers the required host functions and executes the main function. The host functions are implemented in JavaScript and provide the necessary functionality for the WASM module to interact with the embedding environment, such as reading and writing to the console. The runtime dynamically grows the string buffer memory of the module whenever a string with a larger byte length than the current buffer size needs to be shared between the embedding environment and the WASM module.

The .NET runtime is implemented in C# using the `wasmtime` package to load and execute the generated WASM module. Similar to the Node.js runtime, it registers the required host functions and provides the necessary functionality for the WASM module to interact with the embedding environment. It handles the dynamic growth of the string buffer memory in the same way as the Node.js runtime. In addition, the .NET runtime also provides two custom exceptions for invalid input and calling the `halt` function.

Both runtimes are functionally equivalent and can be used interchangeably to execute the generated WASM modules, with the only exception being, that the Node.js runtime requires the user to manually parse the generated WAT file to a binary WASM file, while the .NET runtime can directly load the generated WAT file and internally handles the parsing to binary WASM.

4.9 Dependencies

This section highlights the dependencies the codebase relies on and why they were selected.

4.9.1 Existing Codebase

Since the task was to extend the existing compiler, from the Compiler Construction course, with a WebAssembly backend, the implementation relies heavily on the existing codebase provided by the course. This includes the existing Compiler front- and backend as well as the ByteCode Library. Additionally, a lot of the existing tests were migrated to the new WASM embedding, to easily test the functionality of the WASM backend.

4.9.2 C#/.NET

The existing compiler and runtime are implemented in C# using the .NET framework. In order to remain fully compatible with the existing codebase, it was a core requirement to use C#/.NET as well. This enabled the reuse of the project's existing build and test infrastructure and allowed for seamless integration and extension of the original codebase.

4.9.3 NUnit

NUnit is a well established testing framework for the .NET ecosystem, providing a comprehensive feature set and strong IDE/CI integration. It met all testing requirements, so there was no reason to switch. Since the project's existing unit and integration tests were already written in NUnit, adopting a different framework would have been counterproductive.

4.9.4 Moq

Moq was used to create mock implementations of interfaces within the unit tests, providing lightweight and controllable substitutes for complex components. This made it straightforward to construct arbitrary control flow graphs and write focused, deterministic tests for CFG construction and the stackifier algorithm, all with minimal setup.

4.9.5 Microsoft Automatic Graph Layout (MSAGL)

MSAGL was used to render control flow graphs as SVG visualizations. These visualizations were integrated into unit tests for manual inspection and debugging of generated graphs. In addition, a separate utility was developed that compiles SmallJ source code and renders the resulting control flow graphs.

4.9.6 Wasmtime

Wasmtime serves as the WebAssembly runtime, used to execute the generated WASM modules. The project depends on the official .NET/nuget port of Wasmtime, which integrates directly into the C# codebase and make it straightforward to run compiled WASM code and write integration tests. To support SmallJ I/O calls, for example `ReadString` and `WriteString`, a small set of host functions is registered. Wasmtime accepts WAT and translates it into WASM and implements all the WebAssembly proposals needed. Although,

support for the GC proposal exists it has to be enabled via a reflection-based workaround.

4.10 Testing

This section summarizes the testing strategy for the compiler and runtime. All automated tests are executed on every CI pipeline run and must pass for the pipeline to succeed. These automated checks are complemented by manual smoke tests that verify the overall functionality before merging to master.

4.10.1 Standard Tests

All standard automated tests are located within the `SmallJToWasmTesting` project. The suite primarily tests the control flow logic and the generation of WAT code. The tests within the `ControlFlow` folder focus on CFG creation, dominator and backedge detection, loop identification and the lifting transformation of graphs into structured nodes. All graph logic is written against interfaces, which makes it straightforward to construct a mock graph with the exact shape needed for a test. The tests within `WasmGeneration` specifically verify the backend's emission of language features: arithmetic operations, arrays, comparisons, function emission, inheritance and other object-oriented patterns, as well as string handling. Small, focused snippets are compiled to WAT and the produced WAT code is inspected.

4.10.2 Runtime Tests

Runtime tests compile SmallJ programs to WASM and execute the resulting modules using Wasmtime. These are black-box integration tests: a program is compiled and run (with the required host functions registered), its output and side effects are captured and assertions verify that the observed behavior matches the expected output. Runtime tests validate the full workflow, from parsing and code generation to module validation and execution.

4.10.3 Coverage Report

The coverage report is generated automatically in the CI pipeline after the test stage. By utilizing the Cobertura format, coverage results are consumed consistently by the pipeline and published as part of the generated coverage artifacts. Besides line and branch coverage, the report also includes additional quality indicators such as CRAP scores, which help to identify

risk hotpots within the codebase. At the end of the pipeline, the generated report is published to GitLab Pages.

4.11 Summary

This chapter covered the specific implementation details of the WASM backend, including the stackifier algorithm, the architecture of the backend and how WASM code is generated from SmallJ bytecode assembly and structured control flow nodes. Further, it described how object-oriented features and strings are implemented in the generated WASM code, as well as the changes that were made to the bytecode generator to support the requirements of WASM code generation. Finally, the dependencies used during implementation were identified and the testing strategy for both unit and integration tests was described.

5 Conclusion

This section summarizes the main results of the thesis and evaluates the developed WebAssembly backend for the SmallJ compiler. It revisits key design decisions made during the implementation, discusses the capabilities and current limitations of the solution and assesses whether the defined project goals were achieved. Finally, the section outlines potential directions for further extensions and development.

5.1 Results

This thesis developed a working WebAssembly backend for the SmallJ compiler together with a Wasmtime-based runtime embedding. The backend also provides control flow graph analysis and a lifting step to structure bytecode for WASM emission.

5.1.1 WASM Backend

The new backend takes SmallJ bytecode as input and compiles it to WAT. All bytecode produced by valid SmallJ code can be handled and translated into valid WAT. This includes:

- Basic arithmetic instructions
- Structured control flow (if, if-else and loops)
- Dynamic dispatching
- Heap memory management, based on the WASM GC proposal
- Custom string management

Further, the produced bytecode CFGs can be rendered into SVGs.

5.1.2 Runtime Embedding

A custom runtime embedding was created on top of Wasmtime. It translates the generated WAT code to WebAssembly and executes it. The embedding also provides the SmallJ runtime functions that are required by the generated code, for example `ReadString` and `WriteString`. As a result, programs can be executed either as standalone applications or directly within integration tests.

5.1.3 Control Flow Analysis

The compiler can generate control flow graphs from any SmallJ bytecode and export them as SVG images for visual inspection. The analysis recognizes structured constructs (loops and if-else branches) and computes properties such as reducibility and dominance.

5.1.4 Current Limitations

Not all arbitrary SmallJ bytecode can currently be compiled to WAT. Some restrictions still apply, namely:

- The bytecode must produce a reducible flow graph and must therefore not contain multi-entry loops. Irreducible control flow graphs cannot be generated from valid SmallJ code and are therefore not supported by the current implementation.
- Self loops are also not supported, since they can not be produced with valid SmallJ code and handling such constructs would raise non-trivial questions regarding their semantically correct treatment. Bytecode such as shown in listing 15 is rejected.

```
1 LDC 1  
2 GOTO -2
```

Listing 15: Self loop example.

- The logic for determining whether an if-else block leaves a value on the stack is based on the current compilation strategy of SmallJ's `&&` and `||` operators to bytecode. While this approach is sufficient and correct for all bytecode generated by the current SmallJ compiler, it relies on implicit assumptions about the lowering strategy and could fail if bytecode were written manually or if the compilation scheme were changed. Nevertheless, the implemented solution is simple, straightforward and effective for its intended use.

5.2 Evaluation

The developed backend fulfills all goals and requirements defined in the assignment (A). All features were thoroughly tested to ensure that they are fully functional and meet the specified requirements. Correctness and readiness were verified through automated unit and integration tests, end-to-end runs in the Wasmtime embedding, manual system tests and extensive code

reviews. Although implementing the lifting step proved to be more challenging than anticipated, the indirect, IR-based approach was nevertheless the right choice given the project’s goals. Adopting the WASM reference types and GC proposal substantially reduced implementation complexity and saved considerable development effort.

5.2.1 Success

This project can be considered successful because it provides direct value for the Compiler Construction course and removes the current “VM inside a VM” setup. It demonstrates how a stack-based language can be compiled to an alternative intermediate representation, how the related control flow challenges can be addressed and how a WebAssembly-based execution environment can replace the original VM. In addition, the implemented solution satisfies the defined project criteria.

5.3 Outlook

There are several directions in which the current results could be extended. A possible next step would be to add support for irreducible flow graphs by introducing a transformation after the reducibility check that converts them into reducible form using node duplication or a dispatcher block. The logic for determining whether an if-else block produces a value could also be made more robust so that it remains stable even if the current lowering strategy from SmallJ compilation strategy changes in the future. In addition, the work on control flow analysis could form the basis for further optimizations in the SmallJ compiler and could also be used in the Compiler Construction course when optimizations are covered. A further possible extension of the project would be the implementation of a custom virtual machine for executing WebAssembly.

List of Abbreviations

AOT	Ahead-Of-Time
CFG	Control Flow Graph
GC	Garbage Collector
I/O	Input/Output
IR	Intermediate Representation
JIT	Just-In-Time
JRE	Java Runtime Environment
JVM	Java Virtual Machine
OO	Object-Oriented
OST	Ostschweizer Fachhochschule
SSA	Static Single Assignment
VM	Virtual Machine
WASM	WebAssembly
WAT	WebAssembly Text Format

Bibliography

- [1] Departement Informatik, “Compilerbau,” OST - Ostschweizer Fachhochschule, 2025. [Online]. Available: https://studien.ost.ch/allModules/40663_M_CoBau.html. [Accessed: Feb. 17, 2026].
- [2] L. Bläser, *Modern Runtime System and Compiler Design*. Independently published, 2020. [Online]. Available: <https://compiler-book.ch/>. [Accessed: Feb. 24, 2026].
- [3] WebAssembly Contributors, “WebAssembly,” WebAssembly, 2026. [Online]. Available: <https://webassembly.org/>. [Accessed: Feb. 24, 2026].
- [4] Node.js Contributors, “Node.js with WebAssembly,” Node.js, 2026. [Online]. Available: <https://nodejs.org/en/learn/getting-started/nodejs-with-webassembly>. [Accessed: Feb. 24, 2026].
- [5] Bytecode Alliance, “wasmtime,” GitHub, 2026. [Online]. Available: <https://github.com/bytecodealliance/wasmtime>. [Accessed: Feb. 24, 2026].
- [6] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, D. Gohman, L. Wagner, A. Zakai, M. Holman, and J. F. Bastien, “Bringing the Web up to Speed with WebAssembly,” in *PLDI 17: ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017.
- [7] R. Battagline, *The Art of WebAssembly: Build secure, portable, high-performance applications*. No Starch Press, Inc., 2021.
- [8] B. Sletten, *WebAssembly: The Definitive Guide. Safe, Fast and Portable Code*. O’Reilly Media, Inc., 2022.
- [9] MDN Web Docs Contributors, “WebAssembly,” MDN Web Docs, 2026. [Online]. Available: <https://developer.mozilla.org/en-US/docs/WebAssembly>. [Accessed: Apr. 22, 2026].
- [10] A. Rossberg, “WebAssembly Specification,” WebAssembly, 2026. [Online]. Available: <https://webassembly.github.io/spec/core/index.html>. [Accessed: Apr. 22, 2026].
- [11] Bytecode Alliance Contributors, “wasm-tools,” GitHub, 2026. [Online]. Available: <https://github.com/bytecodealliance/wasm-tools>. [Accessed: Apr. 25, 2026].

- [12] WebAssembly Contributors, “WebAssembly Proposals,” GitHub, 2026. [Online]. Available: <https://github.com/WebAssembly/proposals>. [Accessed: Apr. 28, 2026].
- [13] W3C, “WebAssembly Community Group,” W3C, 2026. [Online]. Available: <https://www.w3.org/community/webassembly/>. [Accessed: Apr. 28, 2026].
- [14] W3C, “WebAssembly Working Group,” W3C, 2026. [Online]. Available: <https://www.w3.org/groups/wg/wasm/>. [Accessed: Apr. 28, 2026].
- [15] WebAssembly Contributors, “Feature Status,” WebAssembly, 2026. [Online]. Available: <https://webassembly.org/features/>. [Accessed: Apr. 28, 2026].
- [16] WebAssembly Contributors, “GC Proposal for WebAssembly,” GitHub, 2026. [Online]. Available: <https://github.com/WebAssembly/gc>. [Accessed: Apr. 28, 2026].
- [17] WebAssembly Contributors, “Reference Types Proposal for WebAssembly,” GitHub, 2026. [Online]. Available: <https://github.com/WebAssembly/reference-types>. [Accessed: Apr. 28, 2026].
- [18] WebAssembly Contributors, “Function Reference Types Proposal for WebAssembly,” GitHub, 2026. [Online]. Available: <https://github.com/WebAssembly/function-references>. [Accessed: Apr. 28, 2026].
- [19] WebAssembly Contributors, “Type Imports Proposal for WebAssembly,” GitHub, 2026. [Online]. Available: <https://github.com/WebAssembly/proposal-type-imports>. [Accessed: Apr. 28, 2026].
- [20] S. Sanderson, “Blazor: a technical introduction,” Steve Sanderson’s Blog, 2017. [Online]. Available: <https://blog.stevensanderson.com/2018/02/06/blazor-intro/>. [Accessed: Feb. 27, 2026].
- [21] M. De Icaza, “Hello WebAssembly,” Mono, 2017. [Online]. Available: <https://www.mono-project.com/news/2017/08/09/hello-webassembly/>. [Accessed: Feb. 27, 2026].
- [22] S. Gingter, “Understanding and Controlling the Blazor WebAssembly Startup Process,” thinkecture, 2023. [Online]. Available: <https://www.thinkecture.com/en/blazor/understanding-and-controlling-the-blazor-webassembly-startup-process/>. [Accessed: Feb. 27, 2026].

- [23] R. Strahl, “Re-Assembling the Web with Web Assembly and Blazor,” in *CODE Magazine*, Sep./Oct. 2018. [Online]. Available: <https://www.codemag.com/article/1809061/re-assembling-the-web-with-web-assembly-and-blazor>. [Accessed: Feb. 27, 2026].
- [24] Microsoft Learn Contributors, “ASP.NET Core Blazor WebAssembly build tools and ahead-of-time (AOT) compilation,” Microsoft Learn, 2026. [Online]. Available: <https://learn.microsoft.com/en-us/aspnet/core/blazor/webassembly-build-tools-and-aot>. [Accessed: Feb. 27, 2026].
- [25] .NET runtime lab Contributors, “.NET Runtime - NativeAOT-LLVM,” GitHub, 2026. [Online]. Available: <https://github.com/dotnet/runtime-lab/tree/feature/NativeAOT-LLVM>. [Accessed: Feb. 27, 2026].
- [26] Google j2cl Contributors, “Getting Started for J2CL/Wasm (Experimental),” GitHub, 2026. [Online]. Available: <https://github.com/google/j2cl/blob/master/docs/getting-started-j2wasm.md>. [Accessed: Feb. 27, 2026].
- [27] TeaVM Contributors, “TeaVM: Build Fast, Modern Web Apps in Java,” TeaVM, 2026. [Online]. Available: <https://teavm.org/>. [Accessed: Feb. 27, 2026].
- [28] R. Garg, “How to Compile Java to WASM,” Baeldung, 2024. [Online]. Available: <https://www.baeldung.com/java-wasm-web-assembly>. [Accessed: Feb. 27, 2026].
- [29] A. Andreev, “New WebAssembly backend,” GitHub, 2024. [Online]. Available: <https://github.com/konsoletyper/teavm/discussions/984>. [Accessed: Feb. 27, 2026].
- [30] cheerpJ Contributors, “Architecture: How does CheerpJ work?,” cheerpj, 2026. [Online]. Available: <https://cheerpj.com/docs/explanation/architecture>. [Accessed: Feb. 27, 2026].
- [31] A. Pignotti, A. Bates, and S. De Rossi, “Deep dive into CheerpJ 3.0: A WebAssembly Java Virtual Machine for the browser,” cheerpj, 2023. [Online]. Available: <https://cheerpj.com/docs/blog/cheerpj-3-deep-dive>. [Accessed: Feb. 27, 2026].
- [32] GraalVM Contributors, “GraalWasm Documentation,” GraalVM, 2026. [Online]. Available: <https://www.graalvm.org/webassembly/docs/>. [Accessed: Feb. 27, 2026].

- [33] GraalVM Contributors, “GraalVM Implementation of WebAssembly,” GraalVM, 2026. [Online]. Available: <https://www.graalvm.org/22.1/reference-manual/wasm/>. [Accessed: Feb. 27, 2026].
- [34] JWebAssembly Contributors, “JWebAssembly,” GitHub, 2023. [Online]. Available: <https://github.com/i-net-software/JWebAssembly>. [Accessed: Feb. 27, 2026].
- [35] A. Kelley, “Zig Compiler Internals,” Zig SHOWTIME, 2020. [Online]. Available: <https://www.youtube.com/watch?v=8MbREuiLQrM>. [Accessed: Mar. 4, 2026].
- [36] LLVM Admin Team, “LLVM,” LLVM, 2026. [Online]. Available: <https://llvm.org/>. [Accessed: Mar. 4, 2026].
- [37] Bytecode Alliance, “Craneflirt,” Craneflirt, 2026. [Online]. Available: <https://craneflirt.dev/>. [Accessed: Mar. 4, 2026].
- [38] Binaryen Contributors, “Binaryen,” Binaryen, 2026. [Online]. Available: <https://github.com/WebAssembly/binaryen>. [Accessed: Mar. 9, 2026].
- [39] B. A. Cota, D. G. Fritz, and R. G. Sargent, “Control Flow Graphs as a Representation Language,” Simulation Research Group, Syracuse University, 1994.
- [40] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Pearson, 2006.
- [41] A. Zakai, “Emscripten: An LLVM-to-JavaScript Compiler,” in *SPLASH 11: Conference on Systems, Programming and Applications*, 2011.
- [42] Y. Iozzelli, “Solving the structured control flow problem once and for all,” learningtech, 2019. [Online]. Available: <https://labs.learningtech.com/blog/control-flow>. [Accessed: Mar. 17, 2026].
- [43] “WebAssembly Troubles part 2: Why Do We Need the Relooper Algorithm, Again?,” troubles.md, 2019. [Online]. Available: <http://troubles.md/why-do-we-need-the-relooper-algorithm-again/>. [Accessed: Mar. 17, 2026].
- [44] LLVM Admin Team, “WebAssemblyFixIrreducibleControlFlow.cpp,” LLVM, 2026. [Online]. Available: https://llvm.org/doxygen/WebAssemblyFixIrreducibleControlFlow_8cpp.html. [Accessed: Mar. 23, 2026].

- [45] LLVM Admin Team, “WebAssemblyCFGStackify.cpp,” LLVM, 2026. [Online]. Available: https://llvm.org/doxygen/WebAssemblyCFGStackify_8cpp.html. [Accessed: Mar. 23, 2026].
- [46] bytecodealliance, “wasmtime,” nuget.org, 2025. [Online]. Available: <https://www.nuget.org/packages/Wasmtime>. [Accessed: Mar. 23, 2026].
- [47] T. Lengauer and R. E. Tarjan, “A Fast Algorithm for Finding Dominators in a Flowgraph,” Stanford University, 1979. [Online]. Available: <https://www.cs.princeton.edu/courses/archive/spr03/cs423/download/dominators.pdf>. [Accessed: Mar. 24, 2026].
- [48] Kommission Lehre und Weiterbildung, “Leitlinie zum Umgang mit KI-basierten Hilfsmitteln in Lehre und Weiterbildung,” OST - Ostschweizer Fachhochschule, 2024. [Online]. Available: <https://ostch.sharepoint.com/teams/TS-StudiengangInformatik/Freigegebene%20Dokumente/Studieninformationen/Leitlinie%20KI-Einsatz%20in%20Lehre%20und%20Weiterbildung.pdf>. [Accessed: Feb. 17, 2026].
- [49] Departement Informatik, “Umgang mit KI bei schriftlichen Arbeiten,” OST - Ostschweizer Fachhochschule, 2024. [Online]. Available: <https://ostch.sharepoint.com/teams/TS-StudiengangInformatik/Freigegebene%20Dokumente/Studieninformationen/D-I-Umgang-mit-KI-bei-schriftlichen-Arbeiten.pdf>. [Accessed: Feb. 17, 2026].
- [50] Conventional Commits contributors, “Specification,” Conventional Commits, 2025. [Online]. Available: <https://www.conventionalcommits.org/en/v1.0.0/#specification>. [Accessed: Feb. 17, 2026].
- [51] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, 2009.

List of Figures

1	Example of an <code>add</code> instruction on a given stack. The left side depicts the stack before the <code>add</code> instruction is executed (at instruction index <code>i=0</code>). The right side depicts the stack after the <code>add</code> instruction is executed (at instruction index <code>i=1</code>). . . .	6
2	Visualization of different compiler architectures as flow chart. . . .	24
3	Example of a simple control flow graph for a program with an if-else instruction. Block A contains the program up until the if-else statement. Block B contains the instructions of the true branch (then block). Block C contains the instructions of the false branch (else block). Block D contains the program after the if-else statement.	26
4	Example of a CFG with different edge types. Forward edges are depicted in blue, retreating edges in red and back edges in green.	27
5	Example of a CFG with an irreducible loop.	28
6	Transformed version of the CFG in figure 5 into a reducible CFG using node duplication.	28
7	Transformed version of the CFG in figure 5 into a reducible CFG using a dispatcher block.	29
8	Example of a control flow graph that causes a relooper-style algorithm to fall back to a dispatcher-based state machine. The depicted CFG models a switch-case language construct [42].	31
9	Example showing how a SmallJ condition containing a logical <code>or ()</code> is lowered to bytecode. The graph on the left shows the original SmallJ control flow, while the graph on the right shows the corresponding simplified flow after lowering to bytecode. The conditions <code>a</code> and <code>b</code> can represent any expressions that evaluate to a boolean value, for example <code>i == 1</code>	36
10	System context view of the SmallJ compiler.	39
11	Container view of the SmallJ compiler.	40
12	Compiler component view of the SmallJ compiler.	41
13	Runtime component view.	42
14	Visualization of the compiler pipeline with the WASM backend.	43
15	Sample CFG generated from bytecode, corresponding to a method containing a simple loop. The graph was generated using the custom-implemented graph rendering tooling, which produces SVG-based visualizations directly from the original source code.	46

16	An example control flow graph (left) and its corresponding dominator tree (right).	47
17	The CFG on the left contains no cycles after the backedges, marked in green, have been removed and is therefore reducible. In contrast, the CFG on the right still contains a cycle after the backedges have been removed and is therefore irreducible.	48
18	The two possible scenarios for an if, triangle-shape (left) and single dominated branch (right). Conditional blocks are marked orange and blocks within then-scope are marked green.	50
19	The two possible scenarios for an if-else, diamond-shape (left) or two separate, dominated branches (right). Conditional blocks are marked orange, blocks within then-scope are marked green and blocks within else-scope are marked blue.	51
20	The layout hierarchy that makes up a complete WASM module. Each box represents a Layout. Layouts with matching colors appear on the same level of the hierarchy.	53
21	Sequence diagram showing the building process and the delegation of work from the <code>ModuleBuilder</code> to the different sub-Layout Builders in order to build a complete <code>ModuleLayout</code>	54
22	Sequence diagram showing the emission process, <code>ModuleEmitter</code> delegates work to the different sub-Layout Emitters in order to emit a complete <code>ModuleLayout</code>	55

List of Tables

1	Comparison of compiler architectures. Each architecture is awarded a grade from 1 to 5 per criterion with 1 being the best and 5 the worst grade.	25
---	---	----

List of Code Listings

1	JavaScript WASM embedding example.	7
2	WASM function import example.	7
3	Comparison of WASM syntax styles.	8
4	Example of if/else statement in WASM.	9
5	Example of a while loop in WASM.	10
6	Example of a function header in WASM.	11
7	Example of a function table in WASM.	12
8	Example of memory allocation by the embedding environment.	13

9	Example of memory operations.	14
10	Example of string handling.	16
11	Example of string handling in embedding environment.	16
12	Example of type declarations.	19
13	Pseudo-code generated by relooper [42].	32
14	Pseudo-code generated by stackifier [42].	32
15	Self loop example.	68

Appendix

A Assignment

The following pages contain the assignment for this bachelor thesis as it was given to us at the beginning of the project.

Assignment for Bachelor Thesis “SmallJ to WASM Compiler”

Sven Elvedi and Timo Strolz

1. Supervisor and Expert

This bachelor Thesis will be developed for the Compiler Construction module at OST internally. It will be supervised by Thomas Corbat [REDACTED], OST. An expert independent of OST will examine the thesis and will be present at the final presentation.

- Guido Zraggen, Google [REDACTED]

2. Students

This project is conducted in the context of the module “Bachelor-Arbeit” in the department “Informatik” by

- Sven Elvedi [REDACTED]
- Timo Strolz [REDACTED]

3. Introduction

Students attending the course Compiler Construction in the Computer Science department at OST develop a complete compiler and a virtual machine for the programming language SmallJ, which was invented by Luc Bläser specifically for this course [SMJ]. During the semester they implement parts of the complete compiler front-end, including lexer, parser and semantic checker as well as the backend for generating the intermediate language SmallJ Bytecode. The project concludes with the students implementing a virtual machine for executing SmallJ Bytecode that can even compile some methods to native machine code as Intel64 instructions with a Just-in-Time compiler.

SmallJ is a subset of Java with a focus on interesting language features when implementing a compiler and a corresponding runtime. The language features basic concepts of structured programming, object-oriented capabilities with virtual dispatching and dynamic heap allocation. Some capabilities are simplified to make the implementation a little more straight forward but still requires meaningful insights into real-world application of the necessary techniques and concepts.

SmallJ Bytecode targets a stack-based virtual machine and is also a subset with slight simplification of the Java Bytecode. WebAssembly (WASM) [WEA], like SmallJ Bytecode is also an intermediate language for a stack-based virtual machine. It has slightly different properties and capabilities, like built-in structured control flow, but lacks string representation.

For students attending the Compiler Construction course it would be interesting to see how the compilation to a different target intermediate representation, like WebAssembly, would look like, what similarities exist, what aspects are different and how the challenges can be solved.

4. Goals of the Project

This bachelor thesis aims to develop a new backend for the SmallJ compiler to target Web Assembly as an alternative intermediate language to SmallJ Bytecode. It is to be determined during the project which parts of SmallJ are even feasible to be implemented within the context of this bachelor thesis. The students can also decide on which features to work on first. Examples for some of the aspects to be realized:

- Basic arithmetic instructions are expected to be straight-forward to implement and serve as simple starting point.
- Structured control flow has different instructions in WASM than in SmallJ Bytecode and needs a different kind of mapping than using primitive goto instruction.
- Dynamic dispatching and heap memory management will provide more significant challenge to be solved.

In case of very quick progress on the compiler backend the project scope could be extended to implementing a virtual machine for WASM execution.

The compiler backend needs to be implemented in the context of the existing SmallJ compiler project in .Net with C#. The project template from the Compiler Construction course serves as starting point for the WASM backend.

5. Documentation

This project must be documented according to the guidelines of the "Informatik" department [SGI]. This includes all analysis, design, implementation, project management, etc. sections. All documentation is expected to be written in English. The project plan also contains the documentation tasks. All results must be complete in the final upload to the archive server [AVT]. One printed copy of the documentation must be handed in (color, two-sided). Additional copies can be requested by the expert.

6. Important Dates

16.02.2026	Start of the bachelor thesis.
08.06.2026	Hand-in of the abstract to the supervisor for checking [ABS]. Information about accessing the corresponding web tool will be given by the department office. Hand-in (by email) of the A0 poster to the supervisor.
12.06.2026 – 17:00 o'clock	Final hand-in of the report
TBD - June 2026	Presentation and Oral Exam

7. Evaluation

A successful term project counts as 12 ECTS points. The estimated effort for 1 ECTS is 30 hours. (See also the module description [6]). The supervisor will be in charge of all the evaluation of the project.

Criterion	Weight
1. Organization, Execution	1/6
2. Report (Abstract, Management Summary, technical and personal reports) as well as structure, visualization, and language of the whole documentation	1/6
3. Content	3/6
4. Final Presentation of the results and discussion	1/6

Furthermore, the general regulations for term projects of the department “Informatik” apply.

8. References

[SMJ] <https://compiler-book.ch>

[WEA] <https://webassembly.org/>

[SGI] <https://ostch.sharepoint.com/:f:/r/teams/TS-StudiengangInformatik/Freigegebene%20Dokumente/Studieninformationen/Studien-%20und%20Bachelorarbeiten>

[ABS] <https://abstract.ost.ch/>

[AVT] <https://avt.i.ost.ch> (only available internally)

Rapperswil, 16. February 2026

Thomas Corbat

Lecturer

Ostschweizer Fachhochschule

B SmallJ Language Report

The following pages contain the SmallJ language report as it was given to us last semester during the Compiler Construction course.

The Programming Language SmallJ

Revision 2025-09-14

Language Report

1 Introduction

SmallJ is a simplified Java-like programming language. It constitutes the source language to be implemented during the assignments of the compiler design course. This language report version comprises all the principal concepts of a modern object-oriented programming language.

The syntax is in the *Extended Backus Naur Formalism (EBNF)*, following the original Wirth notation.

Object-oriented features are specifically highlighted, as they will be discussed later in the course.

2 Program

A program comprises a set of classes.

Program = { Class }.

Example:

```
class FirstClass {
    void main() {
        SecondClass obj;
        obj = new SecondClass();
        obj.x = 3;
        obj.print();
    }
}

class SecondClass {
    int x;
    void print() {
        write(x);
    }
}
```

The method with identifier `main` defines the program start and has no parameters and no return type. Exactly one method with this identifier must be declared in one class of the program.

Object-Orientation

On program start, an initial object of the class with the `Main` method is created and the `Main` method is invoked for that object.

3 Declarations

A declaration specifies a class, field, variable, or method, with an associated identifier. Every identifier in the program refers to a declaration, except reserved keywords.

3.1 Classes

A class contains field and method declarations.

```
Class = "class" Identifier [ "extends" Identifier ] "{"  
      { Variable | Method }  
      "}".
```

Example:

```
class Range {  
    int left;  
    int right;  
    void increaseRight() { ... }  
    void decreaseLeft() { ... }  
    int distance() { ... }  
}
```

Object-Orientation

A class can inherit from a base class, such that it automatically contains all fields and methods of its direct base class. Inheritance is transitive, i.e. a class also inherits from indirect base classes. Inheritance must be acyclic. There is no implicit root class. Only single-inheritance is supported.

The inheriting class may contain methods and with identical head declarations than its base classes. In this case, the body of the method is redefined for the inheriting class (overriding). The inheriting class can also declare variables with equal identifiers than in its base classes (hiding base class variables).

No constructors are supported.

Example:

```
class BoundedRange extends Range {  
    int min;  
    int max;  
    void fitToBoundaries() { ... }  
    void increaseRight() { ... }  
    void decreaseLeft() { ... }  
}
```

3.2 Variables

Fields are variables associated with an object, whereas local variables and parameters belong to a method.

Variable	= Type Identifier ";"
----------	-----------------------

Fields and local variables are always initialized with the default value of the declared type.

Object-Orientation

Fields are associated with an object of the class. There exist no static fields.

3.3 Methods

Methods can have a return type or, if none, the keyword void.

Method	= MethodHead MethodBody.
MethodHead	= Type Identifier "(" ParameterList ")".
ParameterList	= [Parameter { "," Parameter }].
Parameter	= Type Identifier.
MethodBody	= StatementBlock.

Examples:

```
void printSeries(int from, int to) {  
    ...  
}  
  
boolean isPrime(int number) {  
    ...  
}
```

Only value parameters are supported, i.e. arguments are passed by copying.

Object-Orientation

Methods are always associated with an object of the class, i.e. there exist no static methods, except for predefined methods.

The following methods are predefined and static:

- void halt(string message): stop program execution with the error message.
- void writeInt(int integer): standard output of the integer value.
- void writeString(string text): standard output of the string with an implicit new line at its end.
- int readInt(): reads and returns an integer from standard input.
- string readString(): reads a text line as a string in standard input. The resulting string has no new line character at its end.

4 Types

The following types are supported:

- Predefined primitive types with copy semantics: values are directly contained in variables and copied on assignments:
 - int: the integer numbers between -2^{31} and $2^{31}-1$

- `boolean`: the Boolean values `true` and `false`
- Type `string` for a reference to an immutable character sequence.
 - The type includes `null` as a reference to nothing.
 - Equality operators compare the string references.
- Arrays are series of elements of a defined type. Elements are indexed by 0 to the length - 1. Only single-dimension and non-nested arrays are supported.
 - Array types have reference semantics: only references are stored in variables; the array itself is not copied on assignments, only the references to it. As for equality operators, the references are compared.
 - Arrays can be created as part of the program execution.
 - An array type includes the `null` keyword as a special value, a reference to nothing.
 - The length of an array `x` can be accessed by the read-only expression `x.length`.

Type = Identifier ["[" "]"].

Examples:

- `int`
- `boolean`
- `string`
- `int[]`
- `Range`
- `BoundedRange[]`

3

Object-Orientation

One or multiple objects can be created from a class. Objects have reference semantics like arrays.

The class is a type, with the following values:

- `null`, a special keyword denoting a reference to nothing.
- A reference to an object of that class.
- A reference to an object of directly or indirectly inheriting class (type polymorphism).

A type `T1` is compatible to a type `T2` if:

- `T1` and `T2` are identical, or,
- `T1` is a directly or indirectly inheriting class of `T2`.

Arrays are not considered as objects. Array types must be equal to be compatible (no covariance).

The default initialization for variables depends on the type:

- `int`: `0`
- `boolean`: `false`
- `string`, arrays, and objects: `null`

5 Statements

The following statements with the usual semantics are provided.

- Empty statement
- Local variable declaration
- Assignments
- If-Statement
- While-statement
- Method calls
- Return-statement

```
StatementBlock  = "{" { Statement } "}".  
Statement      = ";" | Variable | Assignment | IfStatement  
               | WhileStatement | CallStatement | ReturnStatement.
```

5.1 Empty Statement

The sole semicolon denotes the empty statement.

Example (two empty statements):

```
;;
```

5.2 Local Variable Declarations

Local variables are declared within a method body.

Example:

```
int x;
```

5.3 Assignments

The type of the expression must be compatible to the type of the designated memory location.

```
Assignment = Designator "=" Expression ";".
```

5.4 If-Statement

An if-statement has an optional else. Braces for the if- and else-block are mandatory.

```
IfStatement = "if" "(" Expression ")" StatementBlock  
            [ "else" StatementBlock ].
```

Examples:

```
if (x < 3) {  
    ...  
}
```

```
if (b) {  
    ...  
} else {  
    ...  
}
```

5.5 While-Statement

Braces are also mandatory.

`WhileStatement = "while" "(" Expression ")" StatementBlock.`

Example:

```
while (x < 10) {  
    x = x + 1;  
}
```

5.6 Method Calls

Only methods with no return type (void) can be called as a statement.

`CallStatement = MethodCall ";"`

Example:

```
Write(x);
```

Object-Orientation

The target object of the invoked method is specified by a designator. The designator is omitted, if the target object is the same as of the caller (this).

Examples:

```
otherRange.increaseLeft();  
decreaseRight();
```

5.7 Return-Statement

For void methods, a return statement without expression can be used to exit the method. Otherwise, a return statement with the result expression is required.

`ReturnStatement = "return" [ Expression ] ";"`

Examples:

```
return x + y;  
return;
```

6 Expression

The supported operators are:

- General compare operators: == (equal), != (unequal)
- Arithmetic compare operators: <, <=, >, >=
- Binary logical operators: && (conjunction), || (disjunction)
- Binary arithmetic operators: +, -, *, /, % (modulo)
- Unary logical operator: ! (not)
- Unary arithmetic operators: + and -, e.g. -3

General compare operators can be applied to any type-compatible values, logical operations can be applied to values of type `boolean`, while arithmetic operations can be applied to type `int`. Compare and

logical operators yield an expression of type `boolean`, while arithmetic operators yield type `int`. Binary logical operators `&&` and `||` effect conditional evaluation.

Object-Orientation

The following operators are additionally supported:

- Type test `x instanceof T`: This compare operator evaluates to true if `x` is not null and the class of the referenced object (dynamic type) of `x` is compatible to type `T`.
- Type cast `(T)x`: This unary operator raises a runtime error if `x` is not null and the dynamic type of `x` is incompatible to type `T`. Otherwise, it yields expression `x`. The type cast has type `T`.

Type tests and type casts can only be applied for `T` being a class type.

Operator precedence has the following order (first means stronger):

- Explicit parentheses `()` or unary operators
- `*`, `/`, `%`
- Binary `+`, `-`
- Compare operators
- `&&`
- `||`

Expression	= LogicTerm { " " LogicTerm }.
LogicTerm	= LogicFactor { "&&" LogicFactor }.
LogicFactor	= SimpleExpression { CompareOperator SimpleExpression }.
CompareOperator	= "==" "!=" "<" "<=" ">" ">=" "instanceof".
SimpleExpression	= Term { ("+" "-") Term }.
Term	= Factor { ("*" "/" "%") Factor }.
Factor	= Operand UnaryExpression "(" Expression ")" TypeCast.
UnaryExpression	= ("!" "+" "-") Factor.
TypeCast	= "(" Identifier ")" Designator.

7 Operands

An operand is a constant integer number, true, false, null, a designator, an object creation, an array creation, or a method call.

A designator refers to a memory location, such as a variable or an array element.

Object-Orientation

A designator can be used to specify the associated object of a field. If the designator is omitted, the field is associated with the object of the current method (`this`).

The read-only operand with the keyword `this` represents the reference to the current object.

Operand	= Literal Designator ArrayCreation ObjectCreation MethodCall.
Literal	= Integer String.

Designator	= Identifier
	Designator "." Identifier
	Designator "[" Expression "]"
	MethodCall.

Examples:

```
x
a[12]
a[i * 3 + 1]
otherRange.left
this.y
```

The array creation yields a reference to a new array of the specified length, with all its elements being default initialized.

ArrayCreation	= "new" Identifier "[" Expression "]"
---------------	---------------------------------------

Examples:

```
new int[100]
new boolean[k]
```

Object-Orientation

The object creation yields a reference to a new object of the specified class, with all fields being default initialized (no constructor).

ObjectCreation	= "new" Identifier "(" ")"
----------------	----------------------------

Examples:

```
new Range()
new BoundedRange()
```

A method call as part of an expression is restricted to methods with a return type (not void).

MethodCall	= Designator "(" ArgumentList ")"
ArgumentList	= [Expression { "," Expression }]

Examples:

```
readInt()
otherRange.distance()
distance()
```

8 Scopes

A declaration belongs to a scope, that is either delimited by the directly surrounding class or method, or otherwise, the global scope of a program. All declarations in the same scope must have distinct identifiers.

Accessibility depends on the scopes:

- A declaration is accessible inside its own scope. Declarations of the global scope are accessible throughout the whole program.
- If a declaration is accessible inside a scope, it becomes also accessible within the inner (nested) scopes, unless its identifier is reused by a new declaration inside the inner scope.
- Local variable declarations are only accessible in statements occurring past the declaration. Local variable identifiers are unique within the entire method scope.

9 Symbols

Usual lexical symbols can be used: identifiers and numeric constants, as well as predefined symbols. White spaces separate symbols, except comments. The program is case-sensitive.

Identifiers begin with a letter.

Identifier	= Letter { Letter Digit }.
------------	------------------------------

Integers are written in decimal form.

Integer	= Digit { Digit }.
Letter	= "A" .. "Z" "a" .. "z".
Digit	= "0" .. "9".

A string literal is written within double quotes, e.g. "Hello!". No string escaping is supported. A string cannot span multiple lines. The supported character encoding depends on the system.

String	= "" { Character } "".
Character	= <i>any character except "</i> .

Examples:

```
string message;  
message = "Hello!";  
writeString("Welcome to Compiler Design");
```

10 Comments

Block comments are enclosed in /* and */ and can be inserted before or after any symbol. Nesting of block comments is not supported. Moreover, line comments // last until the new line character (\n).

11 Reserved Keywords

boolean class else extends false if instanceof int string this true new null
return void while

12 Syntax Summary

```
Program      = { Class }.  
Class        = "class" Identifier [ "extends" Identifier ] "{"  
              { Variable | Method }  
              }".  
Variable     = Type Identifier ";".
```

Method	= MethodHead MethodBody.
MethodHead	= Type Identifier "(" ParameterList ")".
ParameterList	= [Parameter { "," Parameter }].
Parameter	= Type Identifier.
MethodBody	= StatementBlock.
Type	= Identifier ["[" "]"].
StatementBlock	= "{" { Statement } "}".
Statement	= ";" Variable Assignment IfStatement WhileStatement CallStatement ReturnStatement.
Assignment	= Designator "=" Expression ";".
IfStatement	= "if" "(" Expression ")" StatementBlock ["else" StatementBlock].
WhileStatement	= "while" "(" Expression ")" StatementBlock.
CallStatement	= MethodCall ";".
ReturnStatement	= "return" [Expression] ";".
Expression	= LogicTerm { " " LogicTerm }.
LogicTerm	= LogicFactor { "&&" LogicFactor }.
LogicFactor	= SimpleExpression { CompareOperator SimpleExpression }.
CompareOperator	= "==" "!=" "<" "<=" ">" ">=" "instanceof".
SimpleExpression	= Term { ("+" "-") Term }.
Term	= Factor { ("*" "/" "%") Factor }.
Factor	= Operand UnaryExpression "(" Expression ")" TypeCast.
UnaryExpression	= ("!" "+" "-") Factor.
TypeCast	= "(" Identifier ")" Designator.
Operand	= Literal Designator ArrayCreation ObjectCreation.
Literal	= Integer String.
Designator	= Identifier Designator "." Identifier Designator "[" Expression "]" MethodCall.
ArrayCreation	= "new" Identifier "[" Expression "]".
ObjectCreation	= "new" Identifier "(" ")".
MethodCall	= Designator "(" ArgumentList ")".
ArgumentList	= [Expression { "," Expression }].
Identifier	= Letter { Letter Digit }.
Integer	= Digit { Digit }.
Letter	= "A" .. "Z" "a" .. "z".
Digit	= "0" .. "9".
String	= "" { Character } "".
Character	= <i>any character except " or new line.</i>

C SmallJ Bytecode Reference

The following pages contain the SmallJ bytecode reference as it was given to us last semester during the Compiler Construction course.

SmallJ Bytecode Reference

Revision 2021-10-14

Instruction set

Opcode	Written out	Operand	Semantic
ldc	load constant	value: boolean, int or string	Push constant value on stack
aconst_null	constant null		Push null value on stack
iadd	integer add		Pop y, x int values and push $x + y$
isub	integer subtract		Pop y, x int values and push $x - y$
imul	integer multiply		Pop y, x int values and push $x * y$
idiv	integer divide		Pop y, x int values and push x / y
irem	integer remainder (modulo)		Pop y, x int values and push $x \% y$
ineg	integer negate		Pop int value and push negative value
bneg	boolean negate		Pop boolean and push negated value
cmpeq	compare equal		Pop y, x type-compatible values and push $x == y$
cmpne	compare not equal		Pop y, x type-compatible values and push $x != y$
icmplt	integer compare less than		Pop y, x int values and push $x < y$
icmple	integer compare less equal		Pop y, x int values and push $x \leq y$
icmpgt	integer compare greater than		Pop y, x int values and push $x > y$
icmpge	integer compare greater equal		Pop y, x int values and push $x \geq y$
if_true	conditional branch if true	offset to target (label)	Pop boolean value and, if true, branch to target
if_false	conditional branch if false	offset to target (label)	Pop boolean value and, if false, branch to target
goto	goto (unconditional branch)	offset to target (label)	Branch to target
instanceof	instanceof operator (type test)	class symbol	Pop instance reference and push true if instance is type or subtype of class, otherwise false
checkcast	check cast (type cast)	class symbol	trigger runtime error if instance reference on top of stack is not a type or subtype of class
load	load parameter or local variable	index "this": 0 params: 1..p locals: p+1..	Push argument or local variable value on stack
store	store to parameter or local variable	index params: 1..p locals: p+1..	Pop value from stack and store into argument or params (assignment to index 0 = "this" not allowed)
getfield	get field (read field)	field symbol	Pop instance reference from stack and push value of field on stack

putfield	put field (write field)	field symbol	Pop value and instance reference from stack, store value in instance field
aload	array load (read array element)		Pop index and array reference from stack and push value of the array element at index
astore	array store (write array element)		Pop value, index and array reference from stack and store value into the array element at index
new	new object	class symbol	Create a new class instance and push its reference on stack
arraylength	array length		Pop array reference from stack and push array length
newarray	new array	array symbol	Pop length from stack, create new array and push its reference on stack
invokestatic	invoke static method (built-in call)	method symbol	Pop argument (if any), call built-in method (static), leave result on stack (if not void)
invokevirtual	invoke virtual method	method symbol	Pop arguments (reverse order) and target object reference from stack, virtual call to method, leave result on stack (if not void)
return	return		Pop method return value from stack (if not void), return to caller

Notes:

- Stack here refers to the evaluation stack of the virtual processor and not to the method call stack.
- Type, method, or field symbols can be used as operands in the bytecode assembler. These are patched to metadata references in the generator completion step.
- Labels can be used for branches in the bytecode assembler. These are patched to relative offsets by the assembler completion step.
- The VM loader patches metadata references to corresponding type and method descriptors of the VM.

Main differences to real Java bytecode:

- Features many more instructions.
- Boolean are represented as bytes with 0 == false.
- No bneg instruction.
- Comparison is combined with conditional branches, ifeq, ifne, iflt, ifle, ifgt, ifgl etc.
- Equal comparison is typed, if_icmpeq (int), if_acmpeq (references).
- No general standalone compare instructions such as icmpit etc.
- No boolean conditional branches if_true, if_false.